ECE30030/ITP30010 Database Systems

Stored Procedures, Functions, Triggers

Chapter 5.3 - 5.4

Charmgil Hong

charmgil@handong.edu

Spring, 2025

Handong Global University

Integrity Constraints

- There are several different ways to protect a database from corruption:
  - Datatypes for the individual columns
  - Primary key and other uniqueness constraints
  - Referential integrity constraints
    - Implement relationships between tables
    - Ensure that enumerated values are valid
    - Implement referenced data

- Major benefits in doing all of this in the database
  - There is no way to “back door” the database
  - All processes run on the same database server, which saves on network traffic

Basic Programming Structures

- Stored Procedures
  - Blocks of code stored in the database that are pre-compiled
  - They can operate on the tables within the database and (indirectly) return scalars or results sets
- Functions
  - Can be used like a built-in function to provide expanded capability to the SQL statements
  - They can take any number of arguments and return a single value
- Triggers
  - Kick off in response to standard database operations on a specified table
  - Can be used to automatically perform additional database operations when the triggering event occurs

Agenda

- Stored Procedures
- Functions
- Triggers

Glossary

- Database catalog: A database instance consists of metadata in which definitions of database objects are stored
  - Base tables
  - Views (virtual tables)
  - Synonyms
  - Value ranges
  - Indexes
  - Users and user groups

Stored Procedures in MySQL

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

  CREATE PROCEDURE <proc-name>

  (param_spec$_1$, param_spec$_2$, …, param_spec$_n$)

  BEGIN

  -- execution code

  END;

  where each param_spec is of the form:

  [**IN** | **OUT** | **INOUT**] <param_name> <param_type>

  - **IN** mode: Allows you to pass values into the procedure
  - **OUT** mode: Allows you to pass value back from procedure to the calling program

Stored Procedures in MySQL

- Example
  - **CREATE PROCEDURE** dept_count_proc(**IN** *dept_name* **VARCHAR**(20),
    OUT *d_count* **INTEGER**)

    BEGIN
    SELECT COUNT(*) **INTO** *d_count*
    FROM *instructor*
    WHERE *instructor.dept_name* = *dept_name*;
    END;

Stored Procedures in MySQL

- Procedures can be invoked using the **CALL** command, followed by the procedure name, and the arguments
- To remove a procedure, use the **DROP PROCEDURE** command

- Example: To invoke *dept_count_proc*()
  - **CALL** *dept_count_proc*('Physics', @n_physics_inst);

- Example: To remove *dept_count_proc*()
  - **DROP PROCEDURE** *dept_count_proc*;

More about Stored Procedures

- One can declare variables in stored procedures
- Can have any number of parameters
- Each parameter must specify whether it is **IN**, **OUT**, or **INOUT**
  - The typical argument list will look like:
    OUT ver_param VARCHAR(25), **INOUT** incr_param INT …
  - Be careful of output parameters for side effects
  - VARCHAR declarations for the parameters have to specify the maximum length
  - The individual parameters can have any supported MySQL datatype
- One can use flow control statements (conditional **IF-THEN-ELSE** or loops such as **WHILE** and **REPEAT**)

IF

- Note that <condition> is a generic Boolean expression, not a condition in the MySQL sense of the word

IF <condition> **THEN**

<statements>

ELSEIF <condition> **THEN**

<statements>

ELSE

<statements>

END IF

  - There can be any number of **ELSEIF** clauses for an **IF** statement

CASE

- Two different syntaxes
  - Syntax 1

CASE <expression>
    WHEN <value> **THEN**
        <statements>
    WHEN <value> **THEN**
        <statements>
    …
    ELSE
        <statements>
END CASE;

CASE

- Two different syntaxes
  - Syntax 2

```
CASE
    WHEN <condition> THEN
        <statements>
    WHEN <condition> THEN
        <statements>
    …
    ELSE
        <statements>
END CASE;
```

LOOP

- Syntax 1
  - [begin_label:] **LOOP**
    <statement list>
    END LOOP [end_label]

- Syntax 2
  - [begin_label:] **REPEAT**
    <statement list>
    UNTIL <search_condition>
    END REPEAT [end_label]

- 'begin_label' must match with 'end_label'
  - Both are optional

WHILE

- Syntax
  - [begin_label:] **WHILE** <condition> **DO**
    <statements>
    END WHILE [end_label]

Loop Control Flow

- **ITERATE** <label> – start the loop again
  - Can only be issued within **LOOP**, **REPEAT**, or **WHILE** statements
  - Works much like the “continue” statement in Java or C++

- **LEAVE** <label> – jumps out of the control construct that has the given label
  - Can only be issued within **LOOP**, **REPEAT**, or **WHILE** statements
  - One can jump out to the out of the outermost loop as desired
    - Can be used at any level of nesting
  - Works much like the “break” statement in Java or C++

Stored Procedure Examples

- An example procedure (src: https://www.javatpoint.com/mysql-procedure)
  - **DELIMITER** &&
    CREATE PROCEDURE get_merit_student()
    BEGIN
      SELECT * **FROM** student_info **WHERE** marks > 70;
      SELECT COUNT(stud_code) **AS** TOT_STUD **FROM** student_info;
    END &&
    DELIMITER ;

  - When **CALL** get_merit_student();

student_info

stud_id	stud_code	stud_name	subject	marks	phone
1	101	Mark	English	68	4124487743
2	102	Joseph	Physics	70	4125143329
3	103	John	Maths	70	4126558833
4	104	Barack	Maths	90	8147434412
5	105	Rinky	Maths	85	4123302197
6	106	Adam	Science	92	8147836587
7	107	Andrew	Science	83	8143395789
8	108	Brayan	Science	85	4125819673
10	110	Alex	Biology	67	8149981030

Result

```
mysql> CALL get_merit_student();
+---------+-----------+-----------+---------+-------+------------+
| stud_id | stud_code | stud_name | subject | marks | phone      |
+---------+-----------+-----------+---------+-------+------------+
|       4 |       104 | Barack    | Maths   |    90 | 8147434412 |
|       5 |       105 | Rinky     | Maths   |    85 | 4123302197 |
|       6 |       106 | Adam      | Science |    92 | 8147836587 |
|       7 |       107 | Andrew    | Science |    83 | 8143395789 |
|       8 |       108 | Brayan    | Science |    85 | 4125819673 |
+---------+-----------+-----------+---------+-------+------------+
5 rows in set (0.00 sec)

+----------+
| TOT_STUD |
+----------+
|        9 |
+----------+
1 row in set (0.00 sec)
```

Delimiter

- Delimiter: a character or string of characters, which is used to complete an SQL statement
  - By default, SQL is using semicolon (;) as the delimiter
  - This may cause problem in stored procedure because a procedure can have many statements
    - One may designate another character or string of character as the delimiter:
      - ```
        DELIMITER &&
        CREATE PROCEDURE get_merit_student ()
        BEGIN
            SELECT * FROM student_info WHERE marks > 70;
            SELECT COUNT(stud_code) AS Total_Student FROM student_info;
        END &&
        DELIMITER ;
        ```
        ```

Stored Procedure Examples

- An example procedure (src: https://www.javatpoint.com/mysql-procedure)
  - **DELIMITER** &&
    CREATE PROCEDURE *get_student* (**IN** *var1* **INT**)
    BEGIN
        SELECT * **FROM** *student_info* **LIMIT** *var1*;
        SELECT COUNT(*stud_code*) **AS** TOT_STUD **FROM** *student_info*;
    END &&
    DELIMITER ;

  - When **CALL** *get_student* (4);

```
mysql> CALL get_student(4);
+---------+-----------+-----------+---------+-------+------------+
| stud_id | stud_code | stud_name | subject | marks | phone      |
+---------+-----------+-----------+---------+-------+------------+
|       1 |       101 | Mark      | English |    68 | 4124487743 |
|       2 |       102 | Joseph    | Physics |    70 | 4125143329 |
|       3 |       103 | John      | Maths   |    70 | 4126558833 |
|       4 |       104 | Barack    | Maths   |    90 | 8147434412 |
+---------+-----------+-----------+---------+-------+------------+
4 rows in set (0.00 sec)

+----------+
| TOT_STUD |
+----------+
|        9 |
+----------+
1 row in set (0.00 sec)
```

Stored Procedure Examples

- An example procedure (src: https://www.javatpoint.com/mysql-procedure)
  - **DELIMITER** &&
    CREATE PROCEDURE *display_max_mark* (**OUT** *highestmark* **INT**)
    BEGIN
        SELECT MAX(*marks*) **INTO** *highestmark* **FROM** *student_info*;
    END &&
    DELIMITER ;

  - **CALL** *display_max_mark*(@*M*);
    SELECT @*M*;

```
mysql> CALL display_max_mark(@M);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @M;
+------+
| @M   |
+------+
|   92 |
+------+
1 row in set (0.00 sec)
```

Stored Procedure Examples

- An example procedure (src: https://www.javatpoint.com/mysql-procedure)
  - **DELIMITER** &&
    CREATE PROCEDURE *display_marks* (**INOUT** *var1* **INT**)
    BEGIN
        SELECT *marks* **INTO** *var1* **FROM** *student_info* **WHERE** *stud_id* = *var1*;
    END &&
    DELIMITER ;

  - **SET** @*M* = '3';
    CALL *display_marks*(@*M*);
    SELECT @*M*;

```
mysql> SET @M='3';
Query OK, 0 rows affected (0.00 sec)

mysql> CALL display_marks(@M);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @M;
+------+
| @M   |
+------+
|   70 |
+------+
1 row in set (0.00 sec)
```

Procedure Calling Anther Procedure

- **CREATE PROCEDURE** innerproc(**OUT** param **INT**)
  BEGIN
    INSERT INTO sometable
    SELECT LAST_INSERT_ID() **INTO** param
  END

- **CREATE PROCEDURE** outerproc(**OUT** param **INT**)
  BEGIN
    CALL innerproc(@a)
    SELECT @a **INTO** param
  END

DDL in Procedure

- **DELIMITER //**
  CREATE PROCEDURE make_table()
  BEGIN
  CREATE TABLE DemoTable (
  Id **INT NOT NULL AUTO_INCREMENT PRIMARY KEY**,
  FirstName **VARCHAR**(20),
  LastName **VARCHAR**(20)
  );
  END//
  DELIMITER ;

- **CALL** make_table();

Stored Procedures in MySQL

- Use **SHOW PROCEDURE STATUS** to display the list of stored procedures that you have created

	Db	Name	Type	Definer	Modified	Created
1	sakila	film_in_stock	PROCEDURE	dbuser@%	2021-03-10 05:49:15	2021-03-10 05:49:15
2	sakila	film_not_in_stock	PROCEDURE	dbuser@%	2021-03-10 05:49:15	2021-03-10 05:49:15
3	sakila	rewards_report	PROCEDURE	dbuser@%	2021-03-10 05:49:15	2021-03-10 05:49:15
4	university_small	dept_count_proc	PROCEDURE	dbuser@%	2021-05-08 16:53:59	2021-05-08 16:53:59

Agenda

- Stored Procedures
- **Functions**
- Triggers

Functions

- Functions: User-defined routines that can act just like a function defined in the database
  - Take arguments and return a single output
  - Syntax:
    - **CREATE FUNCTION** <name> (<arg1> <type1>, [<arg2> <type2> [,…]) **RETURNS** <return type> [**NOT DETERMINISTIC** | **DETERMINISTIC**]
    - **DETERMINISTIC** means that the output from the function is strictly a consequence of the arguments
    - The arguments are immutable (cannot change), and the new values passed back to the caller
  - Follow that with **BEGIN** … **END** and you have a function

Functions

- Functions are declared using the following syntax:

  FUNCTION <function-name> (param_spec$_1$, …, param_spec$_k$)

  RETURNS <return_type> [**NOT DETERMINISTIC** | **DETERMINISTIC**]

  BEGIN

  -- execution code

  END;

  where param_spec is:

  <param_name> <param_type>

- Requires an ADMIN privilege to create functions on MySQL

Functions

- Example
  - **CREATE FUNCTION** *dept_count*(*dept_name* **VARCHAR**(20))
    RETURNS INTEGER
    BEGIN
    DECLARE *d_count* **INTEGER**;
    SELECT COUNT(*) **INTO** *d_count*
    FROM *instructor*
    WHERE *instructor.dept_name* = *dept_name*
    RETURN *d_count*;
    END;

  - The function *dept_count*() can be used to find the department names and budget of all departments with more than 12 instructors
    - **SELECT** *dept_name*, *budget*
      FROM *department*
      WHERE *dept_count*(*dept_name*) > 12

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called table functions
- *E.g.*, return all instructors in a given department
  - **CREATE FUNCTION** *instructor_of*(*dept_name* **CHAR**(20))
    RETURNS TABLE(
    ID **VARCHAR**(5),
    name **VARCHAR**(20),
    dept_name **VARCHAR**(20),
    salary **DECIMAL**(8, 2))
    RETURN TABLE
    (**SELECT** *ID, name, dept_name, salary*
    FROM *instructor*
    WHERE *instructor.dept_name = dept_name*)
  - Usage: **SELECT** * **FROM TABLE** (*instructor_of*('Music'));

Function Examples

- An example function (src: https://www.javatpoint.com/mysql-functions)
  - **DELIMITER** $$
    CREATE FUNCTION get_designation_name(d_id **INT**)
    RETURNS VARCHAR(20) **DETERMINISTIC**
    BEGIN
    DECLARE de_name **VARCHAR**(20);
    SELECT name **INTO** de_name **FROM** designation **WHERE** id=d_id;
    RETURN de_name;
    END $$
    DELIMITER ;
  - **SELECT** *id*, *get_designation_name*(*d_id*) **AS** DESIGNATION, *name*
    FROM *staff*;

staff

id	d_id	name
1	4	Sueun
2	2	Jihyun
3	5	Juwon
4	1	Harim
5	4	Dahee
6	3	Dokyeong

designation

id	name
1	Trainee Engineer
2	Project Lead
3	Project Manager
4	System Analyst
5	Program Manager

Function result

id	DESIGNATION	name
1	System Analyst	Sueun
2	Project Lead	Jihyun
3	Program Manager	Juwon
4	Trainee Engineer	Harim
5	System Analyst	Dahee
6	Project Manager	Dokyeong

Function Examples

- An example function (src: https://www.mysqltutorial.org/mysql-stored-function/)
  - **DELIMITER** $$
    CREATE FUNCTION *CustomerLevel*(*credit* **DECIMAL**(10,2))
    RETURNS VARCHAR(20) **DETERMINISTIC**
    BEGIN
    DECLARE *customerLevel* **VARCHAR**(20);
    IF *credit* > 50000 **THEN**
    SET *customerLevel* = 'PLATINUM';
    ELSEIF (*credit* >= 50000 **AND** *credit* <= 10000) **THEN**
    SET *customerLevel* = 'GOLD';
    ELSEIF *credit* < 10000 **THEN**
    SET *customerLevel* = 'SILVER';
    END IF;
    RETURN (*customerLevel*);
    END$$
    DELIMITER ;

Function Examples

- An example function (src: https://www.mysqltutorial.org/mysql-stored-function/)
  - Calling a function
    SELECT *customerName*, *CustomerLevel*(*creditLimit*)
    FROM *customers*
    ORDER BY *customerName*;

Data

customerName	creditLimit
Alpha Cognac	61100.00
American Souvenirs Inc	0.00
Amica Models & Co.	113000.00
ANG Resellers	0.00
Anna's Decorations, Ltd	107800.00
Anton Designs, Ltd.	0.00
Asian Shopping Network, Co	0.00
Asian Treasures, Inc.	0.00
Atelier graphique	21000.00
Australian Collectables, Ltd	60300.00
Australian Collectors, Co.	117300.00

Function result

customerName	CustomerLevel(creditLimit)
Alpha Cognac	PLATINUM
American Souvenirs Inc	SILVER
Amica Models & Co.	PLATINUM
ANG Resellers	SILVER
Anna's Decorations, Ltd	PLATINUM
Anton Designs, Ltd.	SILVER
Asian Shopping Network, Co	SILVER
Asian Treasures, Inc.	SILVER
Atelier graphique	NULL
Australian Collectables, Ltd	PLATINUM
Australian Collectors, Co.	PLATINUM

Functions in MySQL

- Use **SHOW FUNCTION STATUS** to display the list of stored procedures that you have created

	Db	Name	Type	Definer	Modified	Created	Security_type
1	sakila	get_customer_balance	FUNCTION	dbuser@%	2021-03-10 05:49:15	2021-03-10 05:49:15	DEFINER
2	sakila	inventory_held_by_customer	FUNCTION	dbuser@%	2021-03-10 05:49:15	2021-03-10 05:49:15	DEFINER
3	sakila	inventory_in_stock	FUNCTION	dbuser@%	2021-03-10 05:49:15	2021-03-10 05:49:15	DEFINER

References

- MySQL Programming by Mimi Opkins
- *www.cse.msu.edu/~pramanik/teaching/courses/cse480/14s/lectures/12/lecture13.ppt by* Sakti Pramanik at Michigan State University
- MySQL Procedural Language by David Brown at California State University Long Beach

- http://dev.mysql.com/doc/
- https://dev.mysql.com/doc/refman/8.0/en/sql-compound-statements.html

Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database
  - *A.k.a.* event-condition-action rule (ECA rule)

- To monitor a database and take a corrective action when a condition occurs
  - *E.g.*, Charge $15 late fee if the credit card balance is not paid in time
  - *E.g.*, Limit the salary increase of an employee to no more than 5% raise

Triggers

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed
  - Specify the actions to be taken when the trigger executes

  - Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases

Triggers

- Syntax
  - **CREATE TRIGGER** *trigger-name*
    trigger-time trigger-event
    ON *table-name*
    FOR EACH ROW
    trigger-action;

    - *trigger-time* ∈ {**BEFORE**, **AFTER**}
    - *trigger-event* ∈ {**INSERT**, **DELETE**, **UPDATE**}

Triggers

- For a complete description, please see: https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html

CREATE
[**DEFINER** = { *user* | **CURRENT_USER** }]
TRIGGER *trigger_name*
trigger_time trigger_event
ON *tbl_name* **FOR EACH ROW**
[trigger_order]
trigger_body

trigger_time: { **BEFORE** | **AFTER** }
trigger_event: { **INSERT** | **UPDATE** | **DELETE** }
trigger_order: { **FOLLOWS** | **PRECEDES** }

Triggers

- Triggering event can be **INSERT**, **DELETE** or **UPDATE**
- Triggers on update can be restricted to specific attributes
  - *E.g.*, **AFTER UPDATE OF** *takes* **ON** *grade*
- Values of attributes before and after an update can be referenced
  - **REFERENCING OLD ROW AS**: for deletes and updates
  - **REFERENCING NEW ROW AS**: for inserts and updates

Triggers

- Triggers can be activated before an event, which can serve as extra constraints
  - Example: Convert blank grades to null
    - **CREATE TRIGGER** *setnull_trigger* **BEFORE UPDATE ON** *takes*
      REFERENCING NEW ROW AS *nrow*
      FOR EACH ROW
      WHEN (*nrow.grade* = ‘ ‘)
      BEGIN ATOMIC
      SET *nrow.grade* = *null*;
      END;

Triggers

- Example: To automatically maintain *credits_earned* value
  - **CREATE TRIGGER** *credits_earned*
    AFTER UPDATE OF *takes* **ON** (*grade*)
    REFERENCING NEW ROW AS *nrow*
    REFERENCING OLD ROW AS *orow*
    FOR EACH ROW
    WHEN *nrow.grade* <> 'F' **AND** *nrow.grade* **IS NOT NULL**
    AND (*orow.grade* = 'F' **OR** *orow.grade* **IS NULL**)
    BEGIN ATOMIC
    UPDATE *student*
    SET *tot_cred* = *tot_cred* + (**SELECT** *credits*
    FROM *course*
    WHERE *course.course_id* = *nrow.course_id*
    WHERE *student.id* = *nrow.id*
    END;

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **FOR EACH STATEMENT**, instead of **FOR EACH ROW**
  - Use **REFERENCING OLD TABLE** or **REFERENCING NEW TABLE** to refer to temporary tables (called *transition tables*) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not to Use Triggers

- Early days, triggers were used for:
  - Maintaining summary data (*e.g.*, total *salary* of each *department*)
  - Replicating databases by recording changes to special relations and having a separate process that applies the changes over to a replica

- Nowadays, there are better ways:
  - Databases provide materialized views to maintain summary data
  - Databases provide built-in support for replication

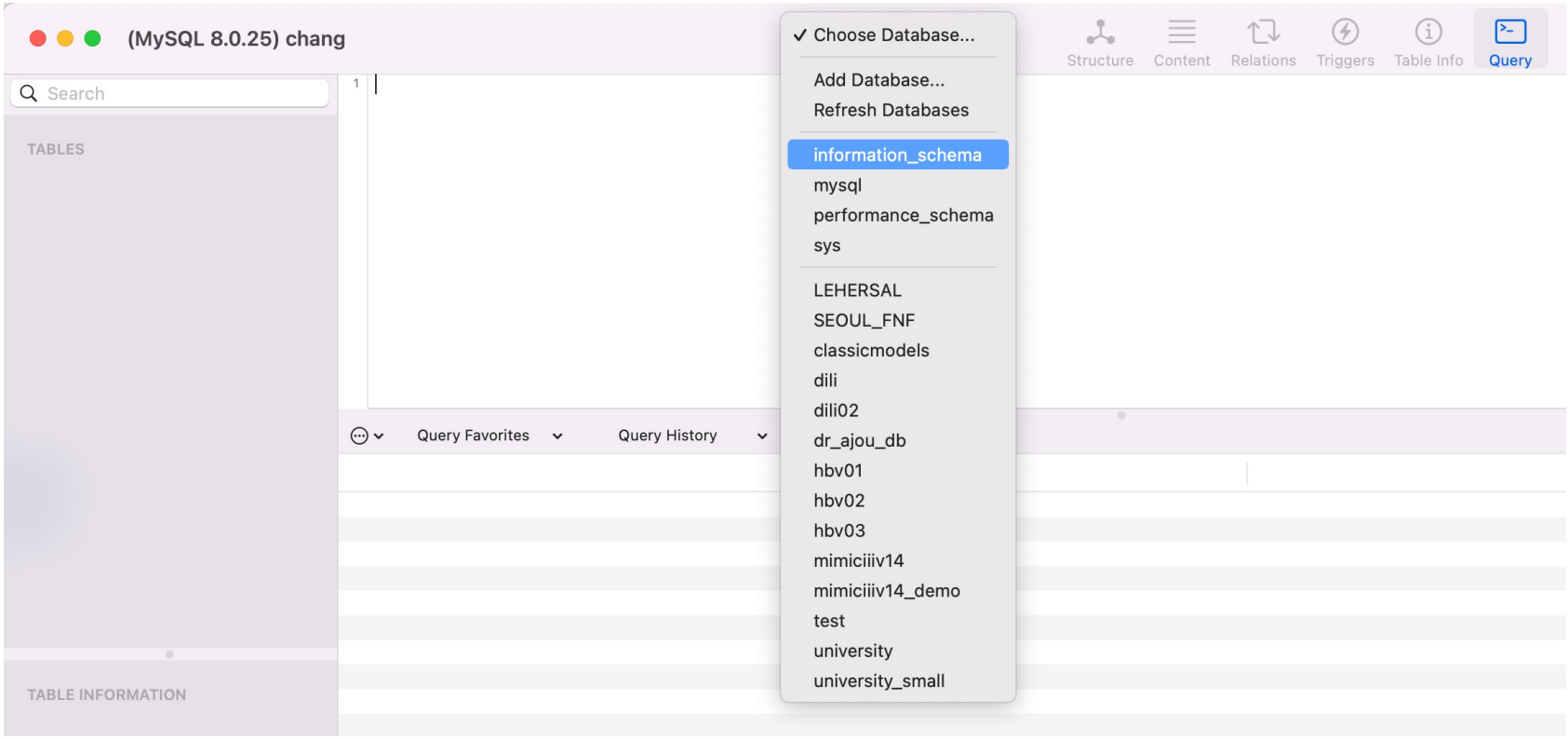
When Not to Use Triggers

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions

- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Viewing Your Triggers

- Method 1: getting information from meta data
  - **SELECT * FROM** *information_schema.triggers*
  - Meta data: MySQL has a schema that has tables for all the information that is needed to define and run the data in the database



Viewing Your Triggers

- Method 1: getting information from meta data
  - **SELECT** * **FROM** *information_schema.triggers*
  - Meta data: MySQL has a schema that has tables for all the information that is needed to define and run the data in the database

- Method 2: using the **SHOW TRIGGERS** command
  - This shows only the triggers in your current database
  - **SHOW TRIGGERS** is not SQL

Remarks

- Naming convention: A good naming standard for a trigger is <table_name>_event
- Like a function or procedure, a trigger body needs a **BEGIN** … **END** unless it is a single statement trigger
- A trigger can only have one event
- If you have the same or similar task that must go on during INSERT and DELETE, then the best approach is *to have that task in a procedure or function* and then *call it from the trigger*

EOF

- Coming next:
  - Indexes