Advanced topic in SQL

Objectives

- Prepared statements
- Store Procedures
- Triggers
- Data Modeling (UML, ER Diagram)

Database Experience

Rather, optional topic covered by be from my experience of working with database. When working with database, there are a couple thing you have to know. This database is the single truth (usually) for your business. And usually business wants more data like when did users login, when did users do transactions the most often. To do so, you usually need to log such data but may not be within the same database table. You may have a audit data table.

Prepared Statements

Prepared statements is used to create queries that contains placeholder that can be replaced with variables ?.
In example:

```sql
SELECT *
FROM Artists
WHERE ArtistID = ?;
```

Where ? can be used to replace as any value you plug in later as following example:

# Start by creating prepared statement
PREPARE stmt FROM 'SELECT *
FROM Artists
WHERE ArtistID = ?';

# In MySQL you can use following syntax to create variable
SET @aid = '1';

# Execute statement with variable
EXECUTE stmt USING @aid;

DEALLOCATE PREPARE stmt;

Store Procedures

Store Procedures allows SQL developers to define function like "procedure" that can contains some application specific logics inside to allow code reusability and utilize the database server to do more.

Defining procedure

In specific, following example creates a GetAllArtists procedure.

```sql
# We start by defining the delimiter so that the SQL command doesn't end with ";
DELIMITER //
CREATE PROCEDURE GetAllArtists()
BEGIN
  SELECT * FROM Artists;
END //
DELIMITER ;
```

Debugging procedure

After we have defined the store procedure, we can then use the following command to find out all procedures in the database.

```sql
SHOW PROCEDURE STATUS WHERE db = 'lyric';
```
And if you want to find out the detail of store procedure, you can do following:

```sql
SHOW CREATE PROCEDURE GetAllArtists;
```

Variables

Of course that the store procedure doesn't only allow SQL developers to define a reusable SQL query. In additional to defining the procedure that runs an arbitrary query, it also gives the ability to define variables like:

```sql
# Syntax to create new variable
DECLARE variable_name datatype(size) DEFAULT default_value;

# Example of createing varialbe and plug SQL result in
DECLARE total_products INT DEFAULT 0;

SELECT COUNT(*) INTO total_products
FROM products
```

Procedure parameter modes
In parameter, we have three modes (IN, OUT and INOUT). We will see how to use mode in few examples below.

**IN**

To use the variable as IN mode example as above, you can follow this example:

```sql
DELIMITER //
CREATE PROCEDURE GetArtistsByCity(IN cityName VARCHAR(25))
BEGIN
SELECT *
FROM Artists
WHERE city = cityName;
END //
DELIMITER ;

Then you can call procedure above like:

CALL GetArtistsByCity('London');
CALL GetArtistsByCity('Alverez');
```

**OUT**

Out is used when we want to get specific result out from Query not just getting table

```sql
DELIMITER $$
CREATE PROCEDURE CountArtistsByCity(IN cityName VARCHAR(25),
OUT total INT)
BEGIN
SELECT count(*) INTO total
FROM Artists
WHERE city = cityName;
END$$
DELIMITER ;

Then you can get result by calling procedure and select from result value:

CALL CountArtistsByCity('London', @total);
SELECT @total;
```

**INOUT**

INOUT mode allows SQL to define mutable variables so that variables can be given to procedure and mutated in the procedure and get updated values outside of it.

```sql
DELIMITER $$
CREATE PROCEDURE set_counter(INOUT count INT(4), IN inc INT(4))
BEGIN
SET count = count + inc;
END$$
DELIMITER ;
```
SET @counter = 1;
CALL set_counter(@counter, 1); -- 2
CALL set_counter(@counter, 1); -- 3
CALL set_counter(@counter, 5); -- 8
SELECT @counter; -- 8

**Triggers**

To create such table and maintain data you can use **Trigger**.

To create a trigger you can follow the following syntax:

```
CREATE TRIGGER `event_name` {BEFORE|AFTER} {INSERT|UPDATE|DELETE}
ON `table_name`
FOR EACH ROW BEGIN
  -- trigger body
  -- this code is applied to every
  -- insert, update, delete row (according to above)
END;
```

Example:

```
CREATE TABLE Audit(
  ArtistID int NOT NULL,
  ChangeTimeStamp DateTime DEFAULT CURRENT_TIMESTAMP
);
CREATE TRIGGER `artist_audit` AFTER INSERT
ON `Artists`
FOR EACH ROW INSERT INTO audit (ArtistId) VALUES (NEW.ArtistID);
```

To drop trigger

```
DROP TRIGGER `artist_audit`;
```

If the trigger body becomes complicated and you need more line, you will need to set up delimiter and change it back accordingly

```
delimiter //
CREATE TRIGGER `artist_audit` AFTER INSERT
ON `Artists`
FOR EACH ROW BEGIN
  INSERT INTO audit (ArtistId) VALUES (NEW.ArtistID);
END; //
delimiter;
```

**Data Modeling**

In MySQL, I'd summarize the data modeling down to a few concept.

1. Normalization
2. One to many vs one to one vs many to many
3. Think ahead on queries

**Normalization**

It’s a process of organizing columns and tables to reduce data redundancy and improve data integrity.
In other word, you want to store as little duplicated data as possible. This concept is the starting of relations!

To do data normalization, you will need to start defining data entities.

Let's take an example project called gradebook -- a student grade sheet project.

This project will probably have a couple entities like students, teachers, grades and classes.

Within each entities, we have to define their attributes.

```plaintext
Student {
  id,
  name,
}

Teacher {
  id,
  name,
}

Class {
  id,
  name,
}

Grade {
  classId,
  teacherId,
  studentId,
  name,
  grade,
}
```

Start with basic and add more attributes/entities based on requirement!

**One to many vs one to one vs many to many**

When one to one relationship is formed between entities. You will probably want to store the relationship in same table.
For example, when teacher has id and name (think of id and name having one to one relationship). They are usually stored under the same table.

When one to many relationship, it is usually stored in additional table like class to grade.

And when many to many relationship happens, they are also usually stored under a different table!

**Think ahead!**

From here, you want to think about the database queries (e.g. what queries you most likely will call).

Let’s say in gradesheet project, we will be calling this grade table a lot to get student grade because this student grade is showing up on the homepage of the student.

Then you might want to consider to index this table first to ensure the read performance. Second of all is how can you optimize the data?