Complex Systems Theory and Practice

Lecture 3

Piotr Syga

2021-10-22

Creating a procedure

Simple example

```
DELIMITER $$
CREATE PROCEDURE counter (OUT result INT)
BEGIN
 SELECT COUNT(*) INTO result FROM Metals;
END $$
DELIMITER ;
```

Using a procedure

```
CALL counter(@res);
SELECT @res;
```

Removing a procedure

```
DROP IF EXISTS counter;
```

Creating a function

Simple example

```
CREATE FUNCTION strMerger (str varchar(10), s varchar(10))
 RETURNS VARCHAR(21) DETERMINISTIC
 RETURN CONCAT(str,' ',s);
```

Using a function

```
SELECT strMerger('complex','systems');
```

Important: Note the difference between variables(`var`) and user defined variables (`@var`).

Prepared statements

```
SET @col = "name";
SET @str = CONCAT("SELECT ", @col, " FROM Metals");
PREPARE stmt FROM @str;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

How to make a procedure that allows using @col as a input parameter?

Prepared statements

```
Delimiter $$
CREATE PROCEDURE uniCol (IN col VARCHAR(50))
BEGIN
SET @str = CONCAT("SELECT ", col, " FROM Metals");
PREPARE stmt FROM @str;
EXECUTE stmt;
END$$
DELIMITER ;
```

We can call the procedure by `CALL uniCol('name')`, `CALL uniCol('name, id')` or `SET @c = "name";CALL uniCol(@c)'`

Delaying the value reveal

Before we used a variable (or procedure parameter) to create a concatenated string, that was prepared to a complete query, we can establish the value later on as well:

```
SET @str = "SELECT * FROM Metals WHERE name=?";
PREPARE stmt FROM @str;
SET @n = "iron";
EXECUTE stmt USING @n;
```

Another example

Let us assume a table salary(workerID, position, wage).

```
DELIMITER $$
CREATE PROCEDURE aboveMean (IN worker INT, OUT above BOOLEAN)
BEGIN
DECLARE mean FLOAT DEFAULT 0;
DECLARE pay FLOAT DEFAULT 0;
SELECT AVG(wage) INTO mean FROM salary;
SELECT wage INTO pay FROM workerID=worker;
IF pay> mean THEN
SET above = TRUE;
ELSE
SET above = FALSE;
END IF;
END$$
DELIMITER ;
```

Another example

Let us assume a table salary(workerID, position, wage).

```
DELIMITER $$
CREATE PROCEDURE aboveMean (IN worker INT, OUT above BOOLEAN)
BEGIN
DECLARE mean FLOAT DEFAULT 0;
DECLARE pay FLOAT DEFAULT 0;
SELECT AVG(wage) INTO mean FROM salary;
SELECT wage INTO pay FROM workerID=worker;
IF pay> mean THEN
SET above = TRUE;
ELSE
SET above = FALSE;
END IF;
END$$
DELIMITER ;
```

Questions: Why vars instead of @vars have been used? How to make universal query for any column and a given fraction (or multiple) of the mean? Can you propose a similar procedure for other quartiles?

The idea

- functions that are called always when a given action takes place
- allows us to control the correctness of the data in a table (since `CHECK` does not work properly)
- verifying the consistency of the data between the tables
- automation of data modification
- backwards compatibility
- access control, logging of the actions

Trigger parameters

Trigger description:

time `BEFORE`, `AFTER`, (MSSQL: `INSTEAD OF`)

action `INSERT`, `UPDATE`, `DELETE`, (some dialects allow `INSERT OR UPDATE`)

Important

There can be only one trigger for a given action on a given table (independent of their time).
The triggers on different tables cannot cross, hence if on insert trigger on table A modifies table B, there cannot be a trigger on B that modifies A.

Inserts

```
CREATE TRIGGER name
AFTER INSERT ON table
FOR EACH ROW
(actions);
```

Optionally, depending on the dialect, you may have access to `OLD` and `NEW` tables or have to set an alias for them: `REFERENCING OLD ROW AS` oldAlias;

Inserts

Lets have two tables: sale_details(id, sale_person_id, sale_person_name, no_products_sold, sales_department) and total_sales(id, sales_department, total_sale)

```
DELIMITER //

CREATE TRIGGER after_insert_sale_details
AFTER INSERT ON sale_details
FOR EACH ROW BEGIN
IF(NEW.sales_department = "Kitchen Essentials")
THEN
INSERT INTO total_sales (id, sales_department, total_sale)
VALUES(1,NEW.sales_department, NEW.no_products_sold)
ON DUPLICATE KEY UPDATE total_sale = total_sale +
NEW.no_products_sold, sales_department = NEW.sales_department;
```

Inserts

```
ELSEIF
(NEW.sales_department = "Apparels")
THEN
INSERT INTO total_sales (id, sales_department, total_sale)
VALUES(2,NEW.sales_department, NEW.no_products_sold)
ON DUPLICATE KEY UPDATE total_sale = total_sale +
NEW.no_products_sold, sales_department = NEW.sales_department;
ELSEIF
(NEW.sales_department = "Medicines")
THEN
INSERT INTO total_sales (id, sales_department, total_sale)
VALUES(3,NEW.sales_department, NEW.no_products_sold)
ON DUPLICATE KEY UPDATE total_sale = total_sale +
NEW.no_products_sold, sales_department = NEW.sales_department;
END IF;
END;//
```

Inserts

INSERT INTO sale_details (id, sale_person_id, sale_person_name, no_products_sold, sales_department)
VALUES(1,"sd1","Henry",2000,"Kitchen Essentials"),
(2,"sd1","Henry",5000,"Apparels"), (3,"sd1","Henry",40,"Medicines"),
(4,"sd2","Richa",3000,"Kitchen Essentials"),
(5,"sd2","Richa",500,"Apparels"), (6,"sd2","Richa",50,"Medicines"),
(7,"sd3","Ved",100,"Kitchen Essentials"),
(8,"sd3","Ved",150,"Apparels"), (9,"sd3","Ved",1000,"Medicines");

What happens after inserting new values, e.g.

INSERT INTO sale_details (id, sale_person_id, sale_person_name, no_products_sold, sales_department) VALUES
(10,"sd3","Ved",1500,"Apparels"), (11,"sd3","Ved",150,"newDept");
How to modify the trigger so that it works properly also for new departments?

Update triggers

Let us add a new auxiliary table:

```
CREATE TABLE historyLogs(
id INT AUTO_INCREMENT,
spi1 VARCHAR(255) ,
spi2 VARCHAR(255) ,
spn1 VARCHAR(255),
spn2 VARCHAR(255),
ps1 INT,
ps2 INT,
sd1 VARCHAR(255),
sd2 VARCHAR(255),
modDate DATE
);
```

Update triggers

```
DELIMITER //
CREATE TRIGGER logging
AFTER UPDATE
ON sale_details FOR EACH ROW
BEGIN
IF (OLD.sale_details <> new.sale_details)
THEN
INSERT INTO historyLogs(spi1, spi2, spn1, spn2, ps1, ps2, sd1,
sd2,modDate)
VALUES(old.sale_person_id, NEW.sale_person_id,
old.sale_person_name, NEW.sale_person_name,
old.no_products_sold, NEW.no_products_sold, old.sales_department,
NEW.sales_department , sysdate());
END IF;
END//
DELIMITER ;
```

Question: Can we store only the changed values?

On delete trigger

```
DELIMITER //
CREATE TRIGGER after_sale_details_delete
AFTER DELETE
ON sale_details FOR EACH ROW
BEGIN
UPDATE total_sales SET total_sale = total_sale -
OLD.no_products_sold WHERE sales_department =
OLD.sales_department;
END; //
```

Note: In general case, one needs to be careful with foreign keys and indexes when deleting a row.

Idea

Example

Let us asume that we make a mobile payment from our account A to some other account B. Let us assume for brevity that all the logging issues are passed and that the accounts use only the queries we already learned. We click the payment button, but in the middle of transfer processing the connection is broken due to some internet issue.
What will happen with the data?

Idea

Let us look on some possibilities:

Event 1

- After transfer confirmation the money was removed from A
- The connection broke before the money was added to B

The result: We lost money, the operation has not been completed, so the money lost is an 'earning' by A's bank.

Idea

Let us look on some possibilities:

Event 2

- After transfer confirmation the money was added to B
- The connection broke before the money was removed from A

The result: We 'created' some money, quite a serious crime.

Idea

The solution:

The goal

We aim at *all-or-nothing* approach, making any operation only if all the declared operations may be completed, forcing the engine to process them 'at the same time'.

Execution – minimal working example

```
SET autocommit = 0;
START TRANSACTION;
SELECT @s:=balance FROM accounts WHERE acc=A;
UPDATE acc SET balance=@s-payment WHERE acc=A;
SELECT @s2:=balance FROM accounts WHERE acc=B;
UPDATE acc SET balance=@s2+payment WHERE acc=B;
COMMIT;
```

Multiple revoke points

MySQL

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

Important:

1. naming a savepoint does not start a transaction
2. reusing the same name overwrites the previous uses

What we cannot rollback

DDL, so `CREATE`, `DROP`, `ALTER`, `TRUNCATE`, `RENAME` of:

- database and users
- tables, views, indexes
- procedures, functions, triggers

cannot be rolled back, as DBE commits the current transaction **before** the DDL-containing query.

The basics

The idea

- 'saving' and 'ordering' of the values in a column (part of the column or multiple columns) in a given row, so we can avoid processing each of the rows
- the queries are made less time consuming, yet at the cost of memory usage
- the indexes are used automatically, if they can be used the optimizer decides to use them
- Note: similar at the first glance queries may be treated differently – most troublesome keywords: `LIMIT`, `LIKE` '%abc' vs 'abc%'

The basics

Creating an index

- There is always at least a single index in a table
- `CREATE INDEX` name_idx `USING` structure `ON` table (col1, col2);
- We can create an index on a part of a column, just as we limit the length of a char or varchar column.
- **Note:** Multicolumn indexes are using concatenation, hence the order is crucial. To use an index all caluses in the condition have to use indexes.

Structures: BTREE, RTREE (only some dialects), HASH
Note: We can select more options – a type of index (e.g., `FULLTEXT`, `SPATIAL`) or a creation algorithm (e.g., `INPLACE`, `COPY`)