Databaskonstruktion

Stored Procedures and Triggers

Stored Procedures

Code within the database, much like a function or method in a programming language

- Faster execution since part of the **optimization** can be performed before execution (used to be necessary on some servers such as Ms Sql Server)
- **Less data** needs to be sent between database server and client (probably not a major consideration in 2020-s)
- **New kinds of behavior** that is not possible without procedures or triggers (such as logging)
- Adds **security** since the developers control exactly what query that is to be executed rather than leaving that decision to a potentially insecure application

Delimiters

For the (some) sql clients we need delimiters to define where each procedure begins and ends

```
DELIMITER//

-- Procedure code!

// DELIMITER ;
```

Simple Procedure (optimize query)

The most basic kind of procedure optimizes code by making the query known to the database beforehand. No parameters and result of query is returned.

```
DELIMITER //

CREATE PROCEDURE GETAVGCOST()
BEGIN
   SELECT AVG(COST) FROM INVOICEROW;
END;
//

DELIMITER ;
```

Update Procedure (a bit like a like a method)

By updating the database with a procedure we guarantee that this is the exact code that is run (though well crafted rights we forbid update statement in application)

```
CREATE PROCEDURE SETCOSTPRODRULE(InInvoice integer,
   InRowno integer, InCost real)
BEGIN
    UPDATE INVOICEROW SET COST=InCost
     WHERE INVOICENO=InInvoice and NUMBER=InRowno;
END;

-- Execute procedure
CALL SETCOSTPRODRULE('1','2','300');
```

Rule Checking Procedure

If we can define our own error codes and error messages we can use the standard exception handling handle the application errors. E.g. “Error 30057 Too Many Rows in Invoice”

There is a statement called **signal** that aborts execution of procedure and produces an error message

This is however **not supported** by MySQL **in older versions** so this is not used in older applications

Rule Checking Procedure

If we can **enforce a rule** in a procedure we can guarantee that only correct data can be stored. We can get much **more complex behavior** than check constraints. We check **before** we perform the update.

```
CREATE PROCEDURE SETCOSTPRODRULE(InInvoice integer,InRowno integer, InCost real)
BEGIN
   IF (InCost<=0) THEN
       SIGNAL SQLSTATE ‘45000’ set message_text =‘Error cost must be positive’
   ELSE
       UPDATE INVOICEROW SET COST=InCost WHERE
        INVOICENO=InInvoice and NUMBER=InRowno;
   END IF;
END;
```

Logging (can not be bypassed by users)

Logging is most common use for procedures and triggers. The process for creating logs is as follows.

1. Decide what to log and when and then create a log table
2. Create trigger or procedure to update log

If a procedure is not used to update the data, a trigger is a **more general** way of taking care of logging

Making the **select a procedure** is the only way to log select statements

Select logging

We want to log which product the users have viewed
We basically add insert to select procedure. USER() and NOW() are predefined functions.

```
CREATE PROCEDURE GETCOSTPROD(prod VARCHAR(30))
BEGIN
    INSERT INTO INVOICEROWLOG(OPERATION,USERNAME,PRODUCT,OPTIME)
      VALUES ("SEL",USER(),prod,NOW());

    SELECT SUM(COST)
    FROM INVOICEROW
    WHERE PRODUCT=prod;
END;
```

Handling denormalization

This oversimplified code can only handle when unpaid invoices are paid if we pay already paid invoices the code does break.
It is also required that we update cost when paying invoice.

```
CREATE PROCEDURE SETCOSTPROD(InRowno integer, InCost real, paid bool)
BEGIN
   IF (NOT paid) THEN
       -- Update item to show new higher cost
       UPDATE INVOICEROW SET COST=InCost WHERE ROWNO=InRowno;
   ELSE
       -- Insert in destination table
       INSERT INTO PAIDINVOICEROW (SELECT * FROM INVOICEROW WHERE ROWNO=InRowno);
       -- Delete from original table
       DELETE FROM INVOICEROW WHERE ROWNO=InRowno;
       -- Update item to show new higher cost
       UPDATE PAIDINVOICEROW SET COST=InCost WHERE ROWNO=InRowno;
   END IF;
END;
```

Materialized views

If a view is used often and takes a lot of time to compute it is better to create a table for the view and use code to update that table
First, figure out which view you wish to optimize

CREATE VIEW CATEGORYSUM AS SELECT AVG(COST) FROM INVOICEROW GROUP BY PRODUCT;

Then, create the Materialized view table

```
CREATE TABLE CATEGORYSUM(
    PRODUCTNAME VARCHAR(30),
    COST        REAL,
    PRIMARY KEY (PRODUCTNAME)
) ENGINE=INNODB;
```

Finally remove the view and create the procedure or trigger (i.e. Original view definition is only there as documentation)

Simple materialized view

```
CREATE PROCEDURE SETCOSTPROD(InRowno integer, InCost real)
BEGIN

   DECLARE done INT DEFAULT 0;
   DECLARE PNAME VARCHAR(30);
   DECLARE PCOST REAL;
   DECLARE cur CURSOR FOR SELECT SUM(COST),PRODUCTNAME FROM INVOICEROW GROUP BY PRODUCTNAME;
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;
   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET done=1;

   UPDATE INVOICEROW SET COST=InCost WHERE ROWNO=InRowno;
   DELETE FROM CATEGORYSUM;

   OPEN cur;
   lbl: LOOP
       IF done=1 THEN LEAVE lbl;
       END IF;
       IF NOT done=1 THEN
           FETCH cur INTO PCOST,PNAME;
           INSERT INTO CATEGORYSUM VALUES(PNAME,PCOST);
       END IF;
   END LOOP;
   CLOSE cur;
END;
```

Smarter materialized view

```
CREATE PROCEDURE SETCOSTPROD(InRowno integer, InCost real)
BEGIN

   DECLARE done INT DEFAULT 0;
   DECLARE PNAME VARCHAR(30);
   DECLARE CNAME VARCHAR(30);
   DECLARE PCOST REAL;
   DECLARE cur CURSOR FOR SELECT SUM(COST),PRODUCTNAME FROM INVOICEROW WHERE PRODUCTNAME=CNAME GROUP BY PRODUCTNAME;
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;
   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET done=1;

   SELECT PRODUCTNAME INTO CNAME FROM INVOICEROW WHERE ROWNO=InRowNo;

   UPDATE INVOICEROW SET COST=InCost WHERE ROWNO=InRowno;
   DELETE FROM CATEGORYSUM WHERE PRODUCTNAME=CNAME;

   OPEN cur;
   lbl: LOOP
       IF done=1 THEN LEAVE lbl;
       END IF;
       IF NOT done=1 THEN
           FETCH cur INTO PCOST,PNAME;
           INSERT INTO CATEGORYSUM VALUES(PNAME,PCOST);
       END IF;
   END LOOP;
   CLOSE cur;
END;
```

Combined Procedures

Procedures often contain a combination of the simpler cases

By combining many tasks into the same procedure we gain additional performance due to the optimizations that the database server can make.

Rules
Logging
Updates / Insert / Select
Denormalization
Materialized View
...

Triggers

- Triggers are automatic stored procedures
- Triggers can do things that are not easy to achieve with procedures
- Triggers work even if the exact query is not known before creation of the trigger whereas in procedures we need to know the code at compile time
- If we introduce logging through the use of triggers it is guaranteed that no operation can be executed under the radar from the log
- One disadvantage is that in most servers we cannot introduce logging of select statements without procedures

Trigger Code

- A trigger can either execute **before** the insert/delete/update happens or **after** it
- Each table can only have **one trigger for each action** e.g. One single before insert.
- If the application needs e.g. two before insert triggers the code in each must be merged into a single trigger
- **Old and New** are both available for update triggers, **New** is not available for delete triggers and **Old** is not available for insert triggers

Log Trigger

```
CREATE TRIGGER LOGGTRIGGER AFTER INSERT
ON INVOICEROW
FOR EACH ROW BEGIN
   INSERT INTO INVOICEROWLOG(OPERATION,USERNAME,PRODUCT,OPTIME)
      VALUES ("INSERT",USER(),NEW.PRODUCTNAME,NOW());
END;
```

Rule Checking Trigger

```
CREATE TRIGGER INSERTCHECK BEFORE INSERT ON INVOICEROW
FOR EACH ROW BEGIN
 IF(NEW.COST<0) THEN
   SIGNAL SQLSTATE ‘45000’ set message_text =‘The cost cant be less than
zero.’
 END IF;
END;
```

Injections

SQL code in applications can be taken over by an attacker. But code in stored procedures is in most cases controlled by the person that designed the procedure.

This is why executing a procedure is more secure than using data direct from tables in an application.

In some very rare cases code must be built during execution. This is also possible in procedures or triggers but poses risks to the security model.

Prepared statement in stored procedure

In this specific case the prepared statement is not needed, and in this case it is recommended to use normal static select statements.

```
CREATE PROCEDURE emps_in_dept2 (in_dept_id VARCHAR(1000))
BEGIN
   SET @sql=CONCAT( "SELECT employee_id,firstname,surname FROM employees
WHERE department_id=",in_dept_id);
   PREPARE s1 FROM @sql;
   EXECUTE s1;
   DEALLOCATE PREPARE s1;
END;
```