# Лабораторная работа №4.
# Хранимые процедуры и функции. Триггеры. Работа со связанными данными.

*Базы данных.*
*3 курс. 5 группа.*
*Кушнеров А.В. 2022-2023 г.*
*Уровень сложности: Средний*
*Формат работы: Индивидуальная по вариантам.*
*Срок выполнения:* ***2 недели.***

**Цель работы**

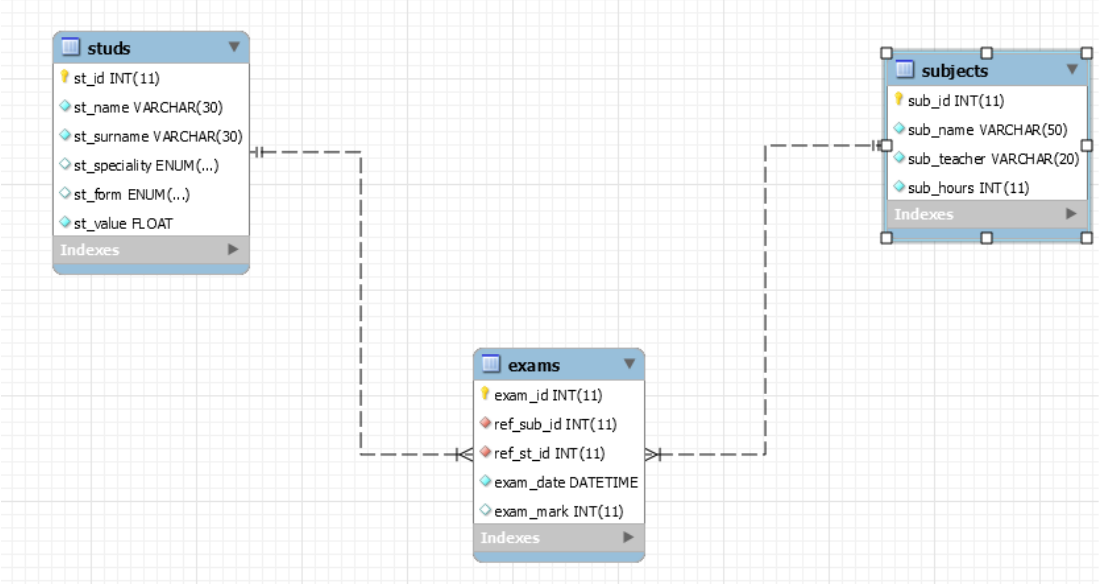
Изучить механизмы построения и применения хранимых процедур и триггеров языка SQL на примере СУБД MySQL. Освоить технологию курсора и её применение. Приобрести навыки работы с простейшими конструкциями программирования в MySQL.

**Минимальные теоретические сведения**

*Хранимая процедура* – набор SQL-инструкций, который размещён на сервере БД. Основное назначение хранимых процедур – повторное и оптимальное использование кода. Однако, их применение также можно обосновать соображениями безопасности (транзактности) и требований бизнес-логики приложения. Для создания процедуры используют оператор CREATE PROCEDURE (ALTER/DROP PROCEDURE для изменения/удаления).

```
CREATE PROCEDURE sp_name ([parameter[, ...]])
    [characteristic ...] routine_body
```

Как видим, у хранимых процедур есть важная особенность – они могут иметь входные параметры. Эти параметры. Также тело процедуры помещается в специальный блок BEGIN…END. Для дальнейшей работы с примерами нам понадобится БД mmf2020, схема которой представлена ниже.




***Пример 1:*** Рассмотрим код простейшей процедуры.

```
DELIMITER //
CREATE PROCEDURE givestuds()
BEGIN
    SELECT * FROM studs;
END//
DELIMITER ;

CALL givestuds();
```

Видим, что данная процедура не принимает на вход параметров и возвращает результат простого SELECT-запроса. ОСОБОЕ ВНИМАНИЕ! Сложность при построении хранимых процедур возникает с символом « ; ». Синтаксис требует заканчивать им строки процедуры, но в тоже время этот символ

заканчивает запрос. Поэтому при создании хранимых процедур, хранимых функций, триггеров мы временно меняем символ конца запроса на \\ с помощью атрибута DELIMITER. Для вызова хранимой процедуры используют оператор CALL.

***Пример 2:*** Рассмотрим пример посложнее. В процедурах мы вполне можем использовать некоторый набор входных параметров. Построим процедуру, возвращающую список студентов, обучающихся на определённом курсе.

```
DELIMITER //
CREATE PROCEDURE givestudsbycourse(IN course INT)
BEGIN
    SELECT * FROM studs
    WHERE st_course=course;
END//
DELIMITER ;
```

Обратите внимание, на модификатор IN перед названием входной переменной. Он показывает, что значение это переменной видно лишь в теле процедуры. Иными, словами это классический шаблон входного аргумента. С другой стороны, MySQL даёт разработчику возможность передать выходной параметр по ссылке с помощью модификатора OUT. Например.

```
DELIMITER //
CREATE PROCEDURE studsquant(OUT res INT)
BEGIN
    SELECT COUNT(*) INTO res  FROM subjects;
END//
DELIMITER ;

CALL studsquant(@xx);

SELECT @xx;
```

В данном случае, результат выполнения процедуры будет передан в переменную, поступившую на вход. Такой подход иногда называют передачей результата по ссылке.

Среда разработки позволяет использовать логические конструкции такие как IF…ELSE или CASE. Они позволяют нам «ветвить» логику нашей процедуры. Рассмотрим пример.

***Пример 3:*** Построим процедуру, добавляющую оценку за экзамен. Важно заметить, что с помощью оператора IF мы проверяем, есть ли укзанный на входе студент и существует ли такой предмет. Оператор DECLARE используют для объявления локальных переменных процедуры.

```
DELIMITER //
CREATE PROCEDURE passexam(IN stud_id INT, IN sub_value VARCHAR(50),IN ex_date datetime, IN ex_mark INT)
BEGIN
    DECLARE id,qua INT;
    SELECT sub_id INTO id FROM subjects
    WHERE sub_name=sub_value;
    SELECT COUNT(*) INTO qua FROM studs
    WHERE st_id=stud_id;
    IF(qua=1 && id>0) THEN
        INSERT INTO exams
        (ref_sub_id,ref_st_id,exam_date,exam_mark)
        VALUES
        (id,stud_id,ex_date,ex_mark);
    ELSE
        SELECT NULL;
    END IF;
END//
DELIMITER ;
```

Как вы могли заметить, для записи результата запроса в переменную мы ловко используем конструкцию SELECT… INTO. Однако, попытка записать таким образом многострочный результат приведёт вас к фиаско. Для записи результат запроса в отдельную таблицу и для других целей, сопряжённых с пробегом по таблице принято использовать *курсоры*.

*Курсор* – именованная область памяти, хранящая результаты SELECT-запроса. Рассмотрим пример.

***Пример 4.*** Запишем результаты некоторого запроса в новую таблицу с помощью курсора. Для объявления курсора используют оператор DECLARE. FETCH – извлечение одной строки курсора с переходом на следующую.

```
CREATE TABLE res_table
(
res_name varchar(30),
res_surname varchar(30)
);
DELIMITER //
CREATE PROCEDURE cursorexample()
BEGIN
    DECLARE n,sn VARCHAR(50);
    DECLARE is_end INT DEFAULT 0;

    DECLARE studscur CURSOR FOR SELECT st_name,st_surname FROM studs;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET is_end=1;

    OPEN studscur;
    curs : LOOP
        FETCH studscur INTO n,sn;
        IF is_end THEN
            LEAVE curs;
        END IF;
        INSERT INTO res_table VALUES (n,sn);
    END LOOP curs;

    CLOSE studscur;
END//
DELIMITER ;
```

Тут остановимся на нескольких моментах. Для отслеживания окончания строк курсора мы объявили специальный обработчик CONTINUE HANDLER FOR NOT FOUND SET. Для прохода по строкам нам нужен цикл. MySQL позволяет использовать ряд циклических конструкций. В данном случае, использован оператор LOOP он повторяет команды из своего тела пока не сработает оператор LEAVE. И последнее, работа с курсором – работа с потоком памяти, значит его обязательно нужно открыть и закрыть.

Наряду с хранимыми процедурами разработчик может использовать и *хранимые функции.* Хранимая функция, может вызываться непосредственно без оператора CALL и всегда возвращает одно значение. Такие функции удобны для применения «налету» в запросах. Смотрим пример.

***Пример 5:*** Создадим хранимую функцию для вычисления средней оценки студента и используем её.

```
DELIMITER //
CREATE FUNCTION average_mark(id INT)
RETURNS FLOAT
BEGIN
    DECLARE averagemark FLOAT;
    SELECT AVG(exam_mark) INTO averagemark FROM exams
    WHERE ref_st_id=id;
    RETURN averagemark;
END//
DELIMITER ;

SELECT * FROM exams
WHERE exam_mark>=average_mark(1622161);

SELECT average_mark(1622161);
```

Важно! Сервер БД может не позволить вам создать хранимую функцию, подозревая её в том, что она NOT DETERMENISTIC. Успокоить его поможет следующая команда.

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

*Триггеры* – хранимые процедуры, которые выполняются автоматически при наступлении некоторого события. На самом деле таких событий три (INSERT, UPDATE, DELETE). Триггеры служат для автоматической проверки или преобразования информации сервером. Рассмотрим пример.

***Пример 6:*** Создадим простейший триггер. Он будет помечен модификатором AFTER INSERT и будет считать все пересдачи студентов в автоматическом режиме.

```
SET @fails=0;

DELIMITER //
CREATE TRIGGER fail_count AFTER INSERT ON exams
FOR EACH ROW
BEGIN
    IF NEW.exam_mark<4 THEN
        SET @fails = @fails+1;
    END IF;
END//
DELIMITER ;

CALL passexam(1622161,'Geometry',now(),2);

SELECT @fails;
```

Для доступа к новому значению используют оператор NEW. Для доступа к предыдущему значению (речь о UPDATE и DELETE) используют оператор OLD.

Для проверки модифицируемых значений перед изменением используют ключевое слово BEFORE.

Такие триггеры называют проверочными. Приводим код триггера, автоматически выставляющего начальное значение стипендии для бюджетников и платников.

```
DELIMITER //
CREATE TRIGGER st_check BEFORE INSERT ON studs
FOR EACH ROW
BEGIN
    CASE NEW.st_form
        WHEN 'budget' THEN
            SET NEW.st_value=100;
        WHEN 'paid' THEN
            SET NEW.st_value=0;
    END CASE;
END//
DELIMITER ;
```

Обратите также внимание на оператор управления логикой – CASE. Порой следует предусмотреть возможность полной отмены вставки. Ниже представлен пример триггера, не допускающего вставку определённых значений в базу данных.

```
DELIMITER //
CREATE TRIGGER sub_check BEFORE INSERT ON subjects
FOR EACH ROW
BEGIN
    IF NEW.sub_name='Databases' THEN
        signal sqlstate '45000';
    END IF;
END//
DELIMITER ;
```

Ранее мы рассмотрели возможность физически связать данные по внешнему ключу. С помощью оператора CONSTRAINT разработчик имеет возможность ограничить значения того или иного столбца, который является внешним ключом. Возникает закономерный вопрос, что происходит с связанными данными при попытке удалить родительскую строку? Рассмотрим пример.

***Пример 7:*** Создадим для примера тестовую БД с двумя таблицами, которые связаны внешним ключом.

```
create database lr4;

use lr4;


create table factory
(
 f_id int4 primary key,
 f_name text not null,
 f_adress text
);

create table cars
(
car_id int4 primary key,
car_name text,
car_factory int4,
constraint cn1 foreign key (car_factory) references factory(f_id)
);
```

Видим таблицы с заводами и автомобилями. Логично, что автомобиль ссылается на завод-производитель. Заполните таблицы данными на ваше усмотрение.

Попробуем удалить данные из таблицы с производителями.

```
delete from factory
where f_name = 'Ford';
```

В результате этой попытки вы получите сообщение об ошибке. Проблема в том, что в таблице с машинами есть записи, которые ссылаются на завод Ford. Суть в том, что при определении CONSTRAINT cn1 мы неявно указали опцию ON DELETE RESTRICT (она устанавливается по умолчанию). Она запрещает удалять строки, которые имеют ссылки в других таблицах. Переопределим CONSTRAINT.

```
alter table cars
drop foreign key cn1;

alter table cars
add constraint cn1  foreign key (car_factory)
references factory(f_id) on delete cascade;
```

Теперь мы явно указали опцию ON DELETE CASCADE. Если теперь снова попытаться удалить строки из таблицы с производителями, то увидим, что вслед за строкой таблицы удалятся и все строки, которые на неё ссылаются в таблице с автомобилями. Таким образом мы настроили удаление связанных данных. Принцип работы опции ON DELETE SET NULL рассмотрите самостоятельно. Также отметим, что за обновление связанных данных отвечает аналогичная опция ON UPDATE.

**Задания для самостоятельной работы.**

1. Реализуйте на своём сервере учебную БД mmf2020 согласно схеме и примерам. Заполните её.
2. Модифицируйте учебную БД mmf2020 до следующих возможностей.
   2.1. Хранение информации о всех оценках студента в семестре.
   2.2. Хранение информации о посещаемости студента на каждом занятии.
   2.3. Хранение информации об общественной нагрузке и активности студента.
   2.4. Хранение информации об оплате за обучение для платников, с возможностью рассрочки.
   2.5. Хранение информации о здоровье студента.
   2.6. Опции удаления и обновления связанных данных.
   2.7. Опции проверки данных при вставке.
3. Добавьте в БД раздельные концепции зачётов и экзаменов. Предусмотрите с помощью триггеров, хранимых процедур и функций бизнес-логику, аналогичную логике на нашем факультете.

4. Реализуйте следующие хранимые процедуры или функции.
   4.1. Процедура для повышения всех стипендий на некоторое количество процентов.
   4.2. Функция, вычисляющая среднюю оценку на экзамене у определённого преподавателя.
   4.3. Процедура для начисления надбавок общественно активным студентам. Критерий начисления надбавок, должен быть привязан к некоторому числовому параметру.
   4.4. Процедуры для вывода топ-5 самых успешных студентов факультета, топ-5 «двоечников», топ-5 самых активных. Результаты курсором записать в новые таблицы.
   4.5. Процедура для отчисления проблемных студентов. Подумайте о проверке условий отчисления.
   4.6. Функция вычисляющую самую популярную оценку на факультете (в группе).
   4.7. Процедура для вычисления процента пропущенных занятий для студентов определённой группы.
   4.8. Процедура для вычисления самых лояльных и предвзятых преподавателей на факультете.
   4.9. Процедура для выдачи бонусов студентам. Принимаю на вход некоторый период времени начисляет надбавку к стипендии студентам, родившимся в этот период. Чем старше студент, тем больше надбавка.
   4.10. Модифицируйте функцию из 3.9 таким образом, чтобы студентам, родившимся в определённый день недели надбавка выдавалась в трёхкратном размере.
   4.11. Процедура для вывода, ожидаемой оценки для студента на предстоящем экзамене. Ожидаемая оценка прогнозируется исходя из лояльности преподавателя, успешности студента и любых других параметров по вашему желанию.
5. Реализуйте следующие триггеры.
   5.1. Триггер для автоматического изменения размера стипендии в зависимости от успеваемости.
   5.2. Триггер для автоматического снижения оплаты при успешной успеваемости.
   5.3. Проверочные триггеры для таблицы со студентами и предметами. На ваш вкус. Их можно придумать 100.
   *5.4. Триггер для автоматического перевода студента на следующий курс при успешной сессии.* *
   5.5. Триггер помечающий потенциально проблемных студентов специальным модификатором.
   5.6. Триггер, не допускающий перевода на следующий курс студента с проблемами по линии здоровья.
   5.7. Триггер для хранения истории изменений определённых полей таблиц.

Литература:

1. Кузнецов, Симдянов – MySQL 5.0.
2. Линн Бейли – Изучаем SQL.
3. Кронке – Теория и практика построения баз данных.
4. Коннолли, Берг – Базы данных.