

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ВЫСШИЙ КОЛЛЕДЖ ИНФОРМАТИКИ

Иваньчева Т.А.

Методическое пособие по языку SQL
(Диалект СУБД MySQL)
Часть 2

Новосибирск
2009

Составитель: Иваньчева Т.А.

МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО ЯЗЫКУ SQL (Диалект СУБД MySQL)

Методическое пособие по языку SQL содержит материал по диалекту языка SQL сервера баз данных `mysql`. С использованием практических примеров и задач рассмотрен язык определения данных, позволяющий создавать объекты баз данных, такие как таблицы, индексы, представления, хранимые процедуры, функции, триггеры. В пособии также рассмотрен язык манипулирования данными, позволяющий выполнять обработку данных. Большое внимание уделено рассмотрению нетривиальных, вложенных запросов.

Данное методическое пособие предназначено для поддержки лекционных и практических занятий по курсу «Базы данных» для студентов 3 курса средне-технического факультета ВКИ НГУ. Пособие может быть также полезно для проведения практических занятий по курсу «Разработка удаленных баз данных», а также может оказать помощь для студентов, выполняющих дипломный проект и использующих в качестве основной СУБД сервер баз данных `MySQL`.

Пособие является результатом обобщения многолетнего опыта подготовки специалистов по технологиям баз данных.

Данный материал опробован на лекционных и практических занятиях по курсу «Базы данных». Данное пособие содержит не только теоретический материал по диалекту языка SQL сервера `MySQL`, но и снабжено многочисленными задачами и практическими заданиями для закрепления этого материала.

Рецензент: Н.А. Аульченко

Ответственный за выпуск: А.А. Юшкова

Оглавление

Глава 6. Итоговые запросы, вложенные запросы	5
Итоговые запросы	5
Вложенные запросы	9
Скалярный подзапрос	9
Табличный подзапрос	9
Использование операций IN и NOT IN	10
Использование операций SOME, ANY и ALL	10
Использование EXISTS и NOT EXISTS	11
Контрольные вопросы	12
Глава 7. Задачи на составление запросов	14
Задания	15
Глава 8. Хранимые подпрограммы	24
Создание подпрограммы	25
Вызов процедуры	28
Вызов функции	30
Ограничения использования некоторых операторов в хранимых подпрограммах и триггерах	31
Локальные переменные	31
Операторы, используемые в теле хранимых процедур	32
Оператор ветвления	32
Оператор выбора	32
Оператор WHILE	33
Использование в циклах операторов LEAVE и ITERATE	34
Оператор REPEAT	36

Оператор LOOP	36
Просмотр списка хранимых процедур и функций	37
Удаление хранимых процедур и функций	39
Курсоры	39
Примеры использования хранимых процедур и функций ...	40
Контрольные вопросы и упражнения	49
Задания для самостоятельной работы	50
Глава 9. Триггеры	53
Создание триггера	53
Примеры использования триггеров	55
Удаление триггера	58
Просмотр существующих триггеров	58
Контрольные вопросы и упражнения	59
Задания для самостоятельной работы	60
Литература	62

Глава 6

Итоговые запросы, вложенные запросы

Итоговые запросы

Итоговые запросы – это запросы, которые позволяют подводить итоги по таблицам. Например:

1. Посчитать количество сотрудников в организации и фонд заработной платы.
2. Посчитать количество сотрудников и среднюю заработную плату по всем отделам организации.
3. Посчитать количество сделок по каждому клиенту.

Итоги подводятся либо по всей таблице, либо по группам. Для лучшего понимания итоговых запросов введем следующие определения:

1. **Поле (поля) группирования** – это одно поле или несколько полей, по которым данные объединяются в группы для подведения итогов. Например, в случае первого примера полей группирования не будет, т.к. итоги подводятся по всей таблице. В случае второго примера полем группирования будет поле НомерОтдела. В случае третьего примера полем группирования будет НомерКлиента.
2. **Итоговое поле** – это, как правило, числовое поле, по которому подводится итог.
3. **Итоговые (агрегатные) функции** задают те итоги, которые подводятся по итоговым полям. Например:
 - `sum(ИтоговоеПоле)` – сумма Итогового поля по группе;

- min(ИтоговоеПоле) – минимальное значение Итогового поля по группе;
- max(ИтоговоеПоле) – максимальное значение Итогового поля по группе;
- avg(ИтоговоеПоле) – среднее-арифметическое значение Итогового поля по группе;
- count(ИтоговоеПоле) – возвращает количество записей в группе, в которых значение Итогового поля не является пустым (NULL).
- count(*) - возвращает количество записей в группе.

Итоговые запросы выполняются с помощью оператора SELECT. Синтаксис оператора в этом случае имеет следующий вид:

```
SELECT <СписокЭлементов> FROM <СписокТаблиц >
      GROUP BY (<ПолеГруппир1, ПолеГруппир2,.. > )
      HAVING <УслНаГруппы >;
```

Где:

СписокЭлементов это список элементов через запятую, который может включать только следующие элементы:

1. Имена полей группирования, которые должны присутствовать во фразе GROUP BY.
2. Итоговые функции.
3. Константы.
4. Выражения из вышеперечисленных элементов.

СписокТаблиц это список таблиц, для которых выполняется операция подведения итогов.

ПолеГруппир1, ПолеГруппир2 – это поле(поля) группирования.

УслНаГруппы – условие, накладываемое на группы.

Фраза GROUP BY задает поле или поля группирования.

Фраза HAVING – задает условие, накладываемое на группы.

В чем отличие HAVING от WHERE? Ведь с помощью WHERE тоже можно отсечь ненужные записи. В HAVING можно

использовать агрегатные функции, а в WHERE нельзя.

Например, даны две таблицы:

Отделы(*НомОтдела, НазваниеОтдела)

Сотрудники(*НомСотр, ФИО, Зарплата, НомОтдела)

Где первичные ключи отмечены *(звездочками).

Требуется выполнить следующие запросы:

Запрос 1. Посчитать количество сотрудников в организации и общий фонд заработной платы по всей организации.

Решение. В этом случае итог подводится по всей таблице, поэтому фраза GROUP BY отсутствует. Запрос будет выглядеть так:

```
SELECT COUNT(*), SUM(Зарплата) FROM Сотрудники;
```

Запрос 2. Посчитать количество сотрудников в каждом отделе и среднюю заработную плату по каждому отделу. Результирующий набор данных должен возвращать таблицу следующей структуры:

Таблица 8

НомОтдела	Кол-во сотрудников	Средняя Зарплата

Решение. В данном случае итоги подводятся по таблице «Сотрудники». Требуется выполнить группирование данных по полю НомОтдела и подвести итоги по каждой группе. Запрос будет выглядеть так:

```
SELECT НомОтдела as “Номер отдела”,  
COUNT(*) as “Кол-во сотрудников”, AVG(Зарплата)  
as “Средняя зарплата” FROM Сотрудники  
GROUP BY (НомОтдела);
```

Запрос 3. Изменим предыдущий запрос следующим образом. Добавим в результирующий набор еще одно поле НазваниеОтдела.

Таким образом, требуется получить следующий результирующий набор:

Таблица 9

НомОтдела	Название Отдела	Кол-во сотрудников	Средняя Зарплата

Решение. Поле НазваниеОтдела находится в другой таблице. Поэтому требуется выполнить соединение таблиц по условию и над полученным соединением выполнить подведение итогов. Группировка будет выполняться по двум полям НомОтдела и НазваниеОтдела и эти же поля указываются в списке оператора SELECT. Запрос будет выглядеть так:

```
SELECT НомОтдела, НазваниеОтдела,
COUNT(*), AVG(Зарплата)
FROM
Сотрудники С INNER JOIN Отделы О
ON С. НомОтдела = О. НомОтдела
GROUP BY (НомОтдела, НазваниеОтдела) ;
```

Запрос 4. Изменим предыдущий запрос следующим образом: пусть требуется выбрать информацию только о тех отделах, в которых более 10 сотрудников.

Решение. В данном запросе следует использовать фразу HAVING, которая позволяет использовать агрегатные функции в условии, которое накладывается на группы. Запрос будет выглядеть так:

```
SELECT НомОтдела, НазваниеОтдела,
COUNT(*), AVG(Зарплата)
FROM
Сотрудники С INNER JOIN Отделы О
ON С. НомОтдела = О. НомОтдела
```


GROUP BY (НомОтдела, НазваниеОтдела)
HAVING (COUNT(*)>10);

Вложенные запросы

SELECT-предложение может содержать подзапрос. Такие подзапросы называют вложенными. Подзапрос может содержаться в условии во фразах WHERE и HAVING. Подзапросы бывают скалярные и табличные. Скалярный подзапрос возвращает одно значение. Табличный подзапрос может возвращать несколько строк, т.е. таблицу.

Скалярный подзапрос

Рассмотрим первый случай, когда подзапрос возвращает одно значение. В этом случае условие во фразе WHERE может иметь следующий вид:

<ИмяАтрибута> <Операция> <Подзапрос>

Где:

Операция это { =, >, <, >=, <=, <> }

Подзапрос это вложенный запрос.

Например, вывести фамилии сотрудников, у которых самая большая зарплата:

```
SELECT ФИО FROM Сотрудники  
WHERE Зарплата = (SELECT max(Зарплата) from Сотрудники);
```

Вложенный запрос возвращает ровно одно значение, поэтому правомерно применение операции равно (=).

Табличный подзапрос

Рассмотрим второй случай, когда подзапрос возвращает один столбец таблицы. В этом случае условие во фразе WHERE может

иметь следующий вид:

<ИмяАтрибута> { [NOT] IN | SOME | ALL } <Подзапрос>

Рассмотрим отдельно использование каждого из операторов [NOT] IN, SOME, ALL.

Пусть даны следующие таблицы:

1. Товары(*НомерТовара, Название, Цена)
2. Клиенты(*НомерКлиента, ФИО, Адрес)
3. Сделки(*НомерСделки, НомерКлиента, НомерТовара, Количество, ДатаСделки)

Назначения полей очевидны. В таблице «Товары» содержится описание вообще всех товаров. В таблице «Клиенты» содержится информация вообще обо всех клиентах. А в таблице «Сделки» содержится информация только о тех товарах, которые покупались клиентами и только о тех клиентах, которые покупали товар.

Для простоты будем использовать русские имена полей.

Использование операций IN и NOT IN

Оператор IN используется для сравнения некоторого значения со списком значений. При этом проверяется, входит ли это значение в список значений или не входит(в случае NOT IN).

Например, выбрать список товаров, которые покупались, т.е. таких, которые содержатся в таблице «Сделка». Запрос будет иметь следующий вид:

```
SELECT Товары.Название FROM Товары
WHERE Товары.НомерТовара IN
(SELECT НомерТовара FROM Сделки)
```

Вложенный запрос вернет столбец значений, поэтому правомерно использование операция IN «содержится в списке». Из таблицы «Товары» будут выбраны все те товары, которые имеются в таблице «Сделки».

Следующий пример, выбрать всех клиентов, которые ни разу не покупали товар:

```
SELECT Клиенты.ФИО FROM Клиенты
WHERE Клиенты.НомерКлиента NOT IN
(SELECT НомерКлиента FROM Сделки)
```

Из таблицы Клиенты будут выбраны только те клиенты, которых нет в таблице Сделка.

Использование операций SOME, ANY и ALL

Все эти ключевые слова используются с подзапросами, которые возвращает один столбец. Операторы SOME и ANY являются синонимами. Если записи подзапроса предшествует ключевое слово ANY или SOME, то условие считается истинным, если оно выполняется, хотя бы для одного значения из списка значений. В случае ALL условие считается истинным, если оно выполняется для всех записей подзапроса.

Например, выбрать всех клиентов, которые хотя бы раз покупали товар:

```
SELECT Клиенты.ФИО FROM Клиенты
WHERE Клиенты.НомерКлиента SOME
(SELECT НомерКлиента FROM Сделки)
```

Следующий пример, выбрать даты сделок с самым большим количеством товара:

```
SELECT ДатаСделки FROM Сделки
WHERE Количество >=ALL
(SELECT Количество FROM Сделки)
```

Использование EXISTS и NOT EXISTS

Операции EXISTS и NOT EXISTS используются, если подзапрос возвращает несколько строк, т.е. целую таблицу. Для ключевого слова EXISTS условие в WHERE будет истинным, если в возвращаемой подзапросом результирующей таблице присутствует, хотя бы одна строка. По ключевым словам EXISTS

и NOT EXISTS проверяется только наличие или отсутствие строк в подзапросе.

Например, выбрать список товаров, которые ни разу не покупались:

```
SELECT Товары.Название  
FROM Товары  
WHERE NOT EXISTS  
  (SELECT * FROM Сделки  
   WHERE Сделки.НомерТовара= Товары. НомерТовара)
```

Контрольные вопросы

1. В каких случаях в итоговом запросе не используется GROUP BY?
2. В чем отличие использования WHERE от HAVING?
3. Какие элементы можно использовать в списке SELECT итогового запроса?
4. В каких случаях следует использовать операторы IN или NOT IN в подзапросах? Приведите примеры.
5. В каких случаях следует использовать операторы EXISTS или NOT EXISTS в подзапросах? Приведите примеры.
6. Чем SOME и ANY отличается от ALL при использовании во вложенных запросах? Приведите примеры.

Задания для самостоятельной работы

Пусть даны следующие таблицы:

1. Товары(*НомерТовара, Название, Цена)
2. Клиенты(*НомерКлиента, ФИО, Адрес)
3. Сделки(*НомерСделки, НомерКлиента, НомерТовара, Количество, ДатаСделки)

Назначения полей очевидны. В таблице «Товары» содержится описание вообще всех товаров. В таблице «Клиенты» содержится информация вообще обо всех клиентах. А в таблице «Сделки» содержится информация только о тех товарах, которые покупались клиентами и только о тех клиентах, которые покупали товар.

Требуется написать SQL-операторы для выполнения следующих запросов:

1. Посчитать общее количество сделок.
2. Посчитать по каждому клиенту количество его сделок и выдать результат в виде:

Таблица 10

Номер Клиента	ФИО	Количество сделок

3. Посчитать по каждому клиенту количество его сделок, на какую сумму и выдать результат в виде:

Таблица 11

Номер Клиента	ФИО	Количество сделок	На общую Сумму

4. Посчитать количество сделок и общую сумму сделок по месяцам года. Выдать результат в виде:

Таблица 12

Номер месяца	Количество сделок	На общую Сумму

5. Выбрать Фамилия только тех клиентов, которые совершили сделок на сумму больше 5000 рублей.

Глава 7

Задачи на составление запросов

База данных «Абоненты» содержит информацию о разговорах абонентов только за один год. Абонент может иметь несколько телефонов, но каждый телефон имеет только одного владельца. С одного телефона можно сделать любое количество звонков, информация о которых хранится в таблице «Разговоры». Информация о тарифах на услуги междугородней связи хранится в таблице «Тарифы».

Таблица «Abonent» хранит информацию об абонентах телефонной сети и имеет следующую структуру:

Имя поля	Тип	Описание
Nab	Integer	Номер абонента, первичный ключ
FIO	Text	Фамилия, имя, отчество абонента
Adres	Text	Адрес абонента

Таблица «Tarif» хранит информацию о тарифах за услуги за междугородние разговоры и имеет следующую структуру:

Имя поля	Тип	Описание
Kod_gor	Integer	Код города, первичный ключ
Nazvanie	Integer	Название города
Tarif	Text	Тариф, цена разговора за минуту

Таблица «Telefon» хранит информацию о наличии телефонов у абонентов и имеет следующую структуру:

Имя поля	Тип	Описание
Nab	Integer	Номер абонента,
Ntel	Integer	Номер телефона, первичный ключ

Таблица «Razgovor» хранит информацию о разговорах абонентов за текущий год и имеет следующую структуру:

Имя поля	Тип	Описание
Nr	Integer	Номер разговора, первичный ключ
Ntel	Integer	Номер телефона
Kod_gor	Integer	Код города, с которым был разговор
Data	Date	Дата разговора
Vrema	Smallint	Время в минутах
Priznak	Smallint	Признак оплаты: 0 – не оплачено, 1 оплачено.

Внимательно изучите структуру базы данных «Абоненты» и, используя язык запросов SQL, составьте запросы к базе данных согласно заданиям.

Задания

1. Определить с какими городами было больше всего разговоров в марте. Результат должен содержать названия городов, как это изображено на рис.7.1.

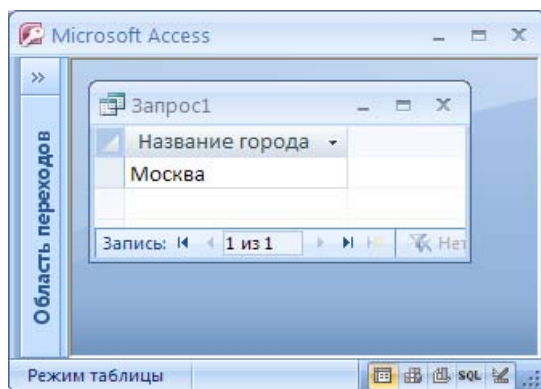


Рис. 7.1.

2. Посчитать количество разговоров по месяцам года. Результат должен содержать следующие поля: название месяца, количество разговоров, на общую сумму, как это изображено на рис.7.2.

Используйте дополнительную таблицу Month(Месяцы), следующей структуры:

Имя поля	Тип	Описание
Nm	Integer	Номер месяца, первичный ключ
Nazv	TEXT	Название месяца

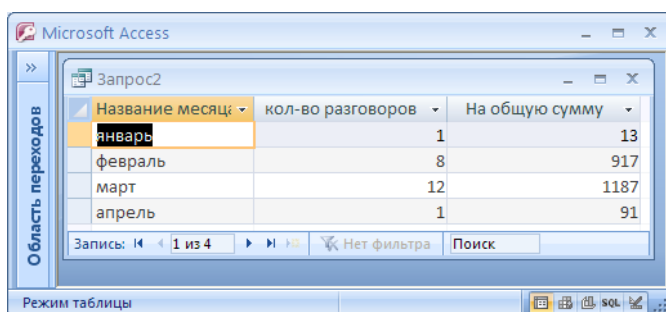


Рис.7.2.

3. Посчитать количество разговоров в марте по городам. Результат должен содержать следующие поля: Название города, Кол-во звонков, На общую сумму, как это изображено на рис.7.3.

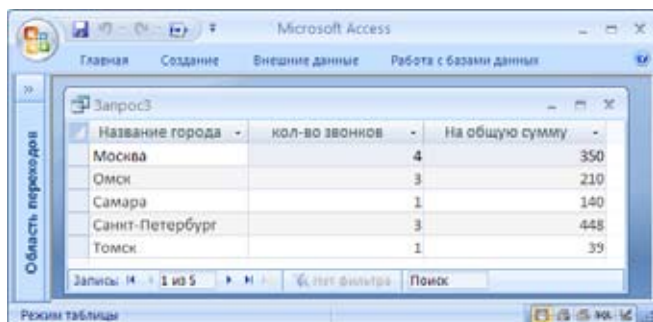
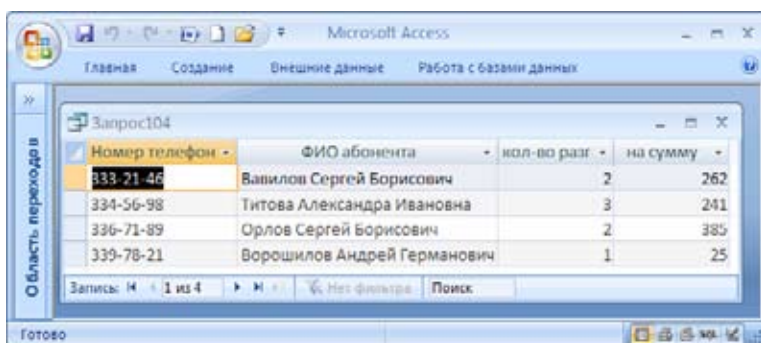


Рис.7.3.

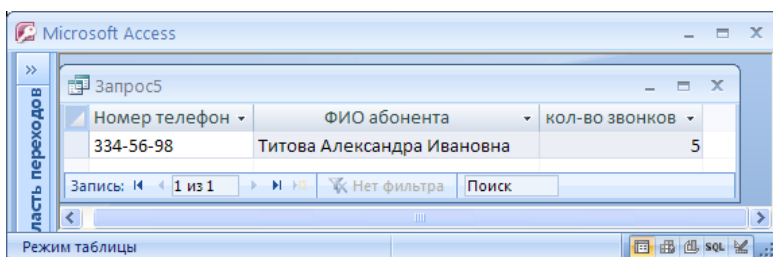
4. Выбрать фамилии и телефоны абонентов, не оплативших разговоры за март. Результат должен содержать ФИО, телефоны абонентов, количество неоплаченных разговоров, а также сумму, которую они должны заплатить (см. рис. 7.4.)



Номер телефон	ФИО абонента	кол-во разг	на сумму
333-21-46	Вавилов Сергей Борисович	2	262
334-56-98	Титова Александра Ивановна	3	241
336-71-89	Орлов Сергей Борисович	2	385
339-78-21	Ворошилов Андрей Германович	1	25

Рис.7.4.

5. Выбрать фамилии и телефоны абонентов, которые в марте произвели максимальное количество звонков. Результат должен содержать номер телефона, фамилии, имена, отчества абонентов, количество звонков, как это показано на рис.7.5.



Номер телефон	ФИО абонента	кол-во звонков
334-56-98	Титова Александра Ивановна	5

Рис 7.5.

6. Вывести фамилии и телефоны абонентов, наговоривших на сумму более 50 рублей в марте и выдать результат в том виде, как это показано на рис.7.6.

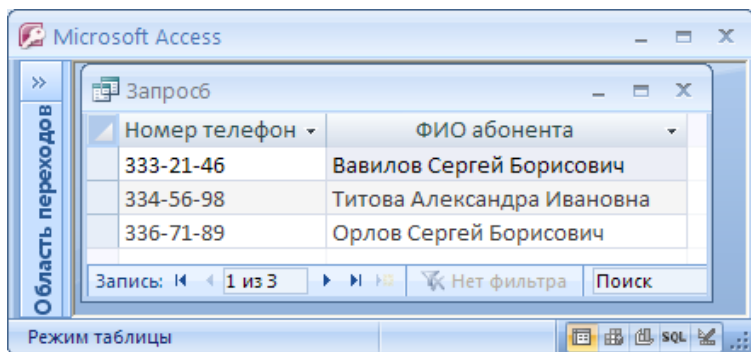


Рис. 7.6.

7. Выбрать телефоны абонентов, с которых звонили в марте только в Москву и больше нигде (рис. 7.7.)

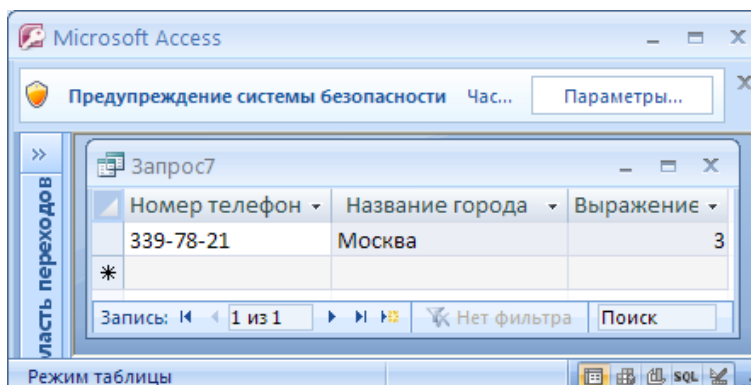


Рис.7.7.

8. Выбрать телефоны абонентов, которые звонили и в Москву и в Санкт-Петербург в марте. Выдать результат в том виде, как это показано на рис.7.8.

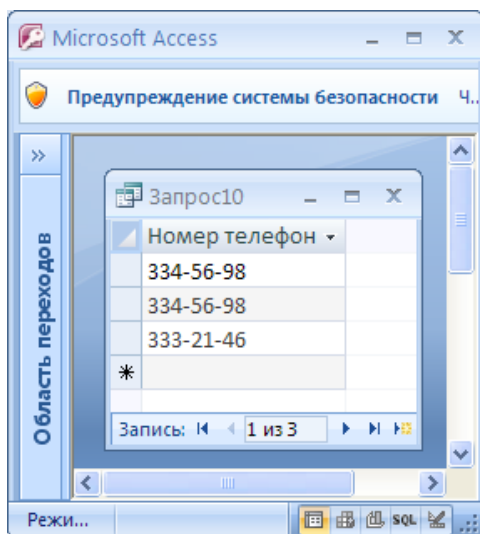


Рис 7.8.

9. Определить названия городов, в которые ни разу не звонили в течение марта.

Выдать результат в том виде, как это показано на рис.7.9.

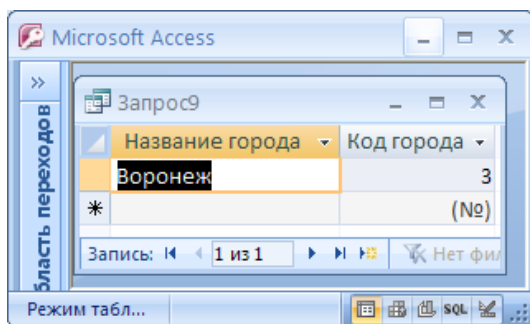


Рис. 7.9.

10. Определить даты в марте, когда количество разговоров было максимальным.

Выдать результат в том виде, как это показано на рис.7.10.

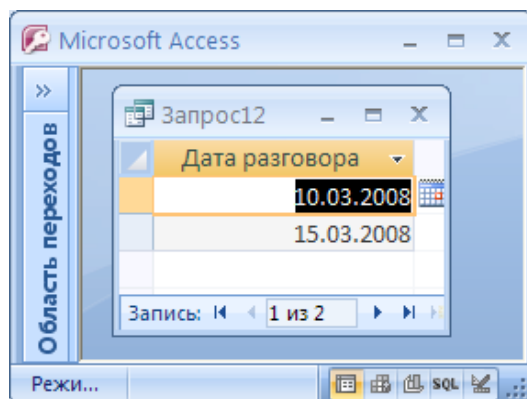


Рис 7.10.

11. Определить названия городов с самым высоким и самым низким тарифом. Результат должен содержать названия городов и тариф, как это показано на рис. 7.11.

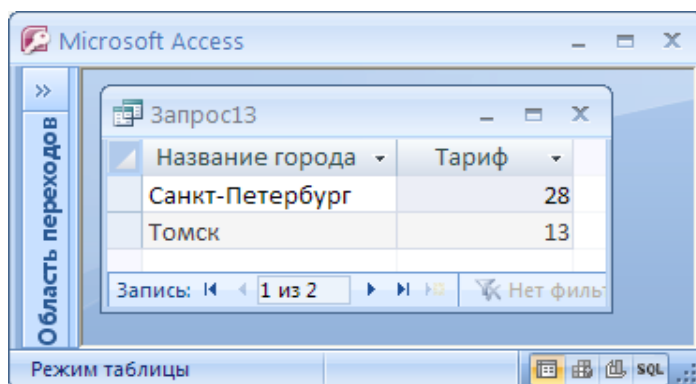


Рис. 7.11.

12. Выбрать те даты в течение года, когда звонки шли во все города, т.е. во все города, которые есть в таблице Тариф (см. рис.7.12).

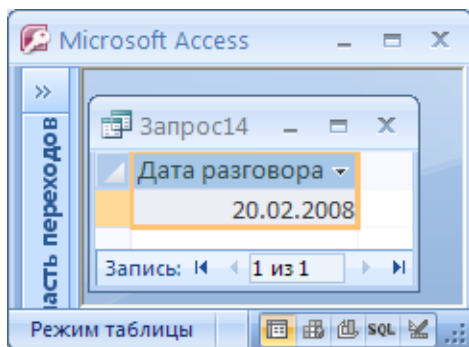


Рис. 7.12.

13. Выбрать фамилии абонентов владельцев более чем одного телефона. Результат должен содержать фамилии абонентов, как это показано на рис. 7.13.

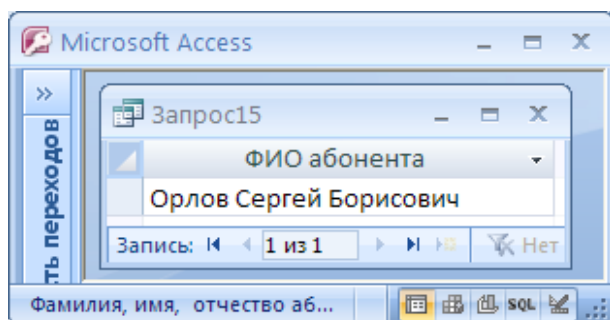


Рис. 7.13.

14. Выбрать телефоны и ФИО абонентов, говоривших ровно один раз с Москвой в марте. Выдать результат в том виде, как это показано на рис. 7.14.

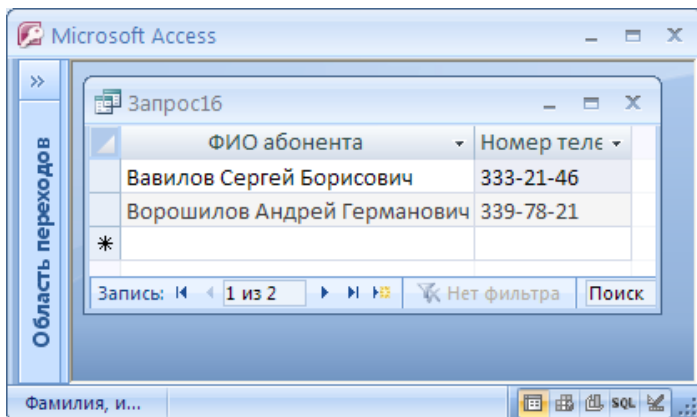


Рис 7.14.

15. Определить распределение количества разговоров по месяцам в процентах. Результат должен содержать: номер месяца, отношение в процентах количества разговоров в данный месяц к общему количеству разговоров за год. Отформатировать столбец «Процент разговоров» как процентный с точностью две цифры после точки. Выдать результат в том виде, как это показано на рис.7.15.

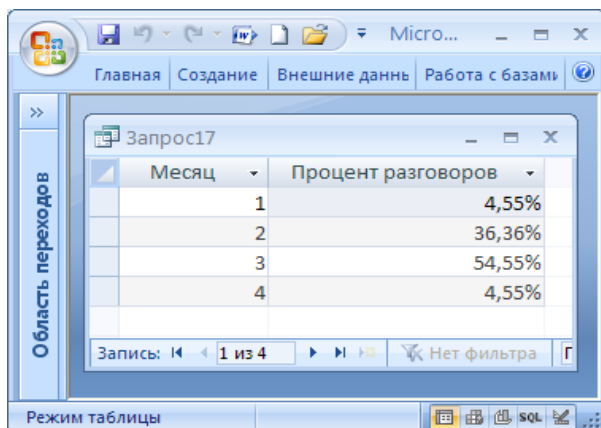


Рис. 7.15.

16. Получить ФИО и телефоны абонентов, которые ни разу не разговаривали с Самарой в течение года. Выдать результат в том виде, как это показано на рис.7.16.

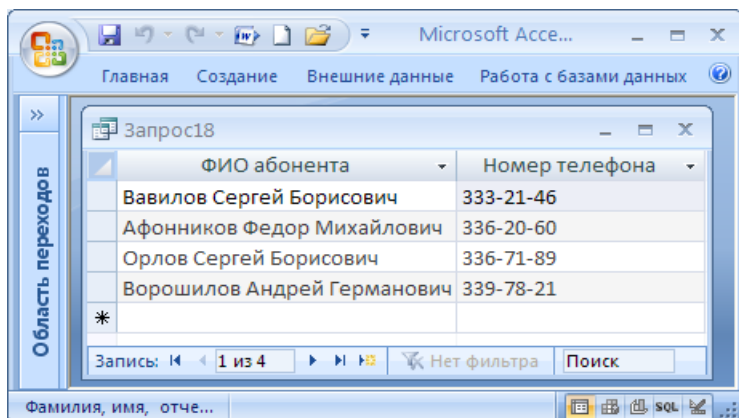


Рис. 7.16.

17. Выбрать ФИО абонентов, телефоны которых начинаются на 336 или на 333. Результат должен содержать ФИО и телефоны таких абонентов(см. рис.7.17.).

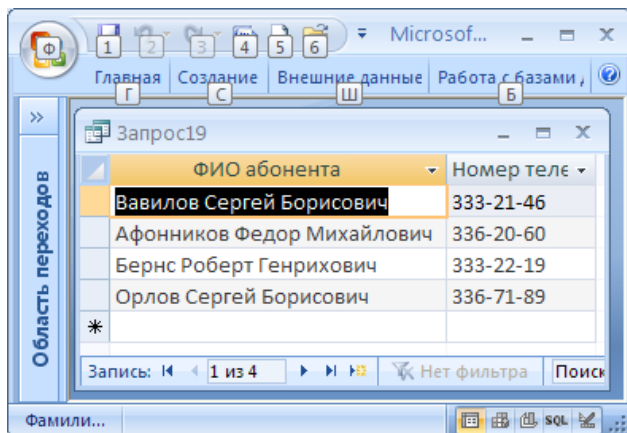


Рис. 7.17.

18. Выбрать абонентов-чемпионов, которые произвели разговоров на максимальную сумму в течение всего года. Результат должен содержать ФИО абонентов, номера телефонов и общую сумму разговоров за весь год, как это показано на рис 7.18.

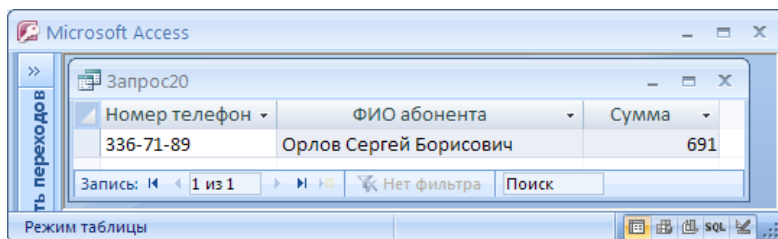


Рис. 7.18.

Глава 8

Хранимые подпрограммы

Хранимые подпрограммы это программный код, который хранится в базе данных вместе с данными. Хранимые подпрограммы могут быть либо процедурами, либо функциями. Преимущества использования хранимых подпрограмм:

Производительность. Хранимые подпрограммы хранятся в откомпилированном, т.е. готовом к выполнению виде, поэтому выполняются быстрее по сравнению с посылкой отдельных SQL-операторов на сервер.

Разгрузка сети по сравнению с посылкой отдельных SQL-операторов на сервер, т.к. для вызова хранимой подпрограммы по сети отправляется только оператор вызова возможно с параметрами, а выполнение всех операторов, входящих в подпрограмму выполняется на сервере.

Централизованное администрирование. Хранимые подпрограммы могут разделяться несколькими приложениями, они хранятся на сервере и поэтому изменение кода хранимой подпрограммы отразится сразу на всех приложениях, которые ее используют.

Создание подпрограммы

Хранимые подпрограммы создаются операторами **CREATE PROCEDURE** и **CREATE FUNCTION**. Процедура вызывается оператором **CALL** и может возвращать несколько значений, используя выходные переменные. Функция может участвовать в выражениях и возвращает скалярное значение. Хранимые подпрограммы могут вызывать другие хранимые подпрограммы.

С помощью оператора **CREATE PROCEDURE** создается процедура, с помощью оператора **CREATE FUNCTION** создается функция. Синтаксис хранимых подпрограмм приведен ниже:

```
CREATE
  PROCEDURE Имя ([ПараметрПр[,...]])
  ТелоПр

CREATE FUNCTION Имя ([ПараметрФ[,...]])
  RETURNS тип
  ТелоФ
ПараметрПр:
  [ IN | OUT | INOUT ] Имя Тип
ПараметрФ
  Имя Тип
```

Тип – любой правильный тип MySQL.

<i>ТелоФ</i> :	<i>ТелоПр</i> :
Begin	Begin
...	...
End;	End;

С помощью операторов `CREATE PROCEDURE` и `CREATE FUNCTION` выполняется синтаксический контроль, и определение подпрограммы записывается в базу данных.

Для создания хранимых подпрограмм необходимо наличие привилегий `CREATE ROUTINE`. По умолчанию MySQL автоматически предоставляет привилегии `ALTER ROUTINE` и `EXECUTE` для создателя хранимых подпрограмм.

Параметры подпрограмм заключаются в круглые скобки. Если параметров нет, то круглые скобки не требуются. Каждый параметр описывается со своим типом. По умолчанию каждый параметр есть `IN` параметр. Для описания других типов параметров нужно явно указать их вид: `OUT` или `INOUT` перед именем параметра. Виды параметров `OUT`, `INOUT` используются только для процедур. Функции используют только параметры `IN`.

`IN`-параметр передает значение в процедуру(функцию). В теле подпрограммы он может модифицироваться, но извне эти изменения не видны.

`OUT`-параметр – значение этого параметра возвращается процедурой. Его начальное значение равно `NULL`.

`INOUT`-параметр - имеет входное значение, может модифицироваться процедурой, возвращается процедурой по ее окончании.

Параметры `OUT` и `INOUT` возвращаются процедурой при ее завершении.

Оператор `RETURNS` определен только для функций.

Тело подпрограмм должно содержать правильные SQL –операторы. Это могут быть простые операторы, такие как **`SELECT`** или **`INSERT`**, или составной оператор в операторных скобках **`BEGIN`** и **`END`**. Составные операторы могут включать описания, циклы и другие управляющие конструкции. Некоторые операторы не разрешены в хранимых подпрограммах.

Запись подпрограммы в базу данных выполняется с помощью консольного приложения `mysql.exe`. Для примера, запишем простую процедуру `simpleproc`, использующую выходной параметр `param1` в базу данных. Для замены разделителя

операторов используется оператор DELIMITER, который изменяет разделитель точку с запятой на символы - //. Оператор

DELIMITER //

говорит о том, что признаком завершения оператора является не точка с запятой(;), а символы //. Поэтому код описания процедуры завершается символами //.

Для восстановления символа окончания операторов используется оператор:

DELIMITER ;

Код процедуры, так как он записывается в базу данных, приведен ниже:

```
mysql> DELIMITER //
```

```
mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
```

```
-> BEGIN
```

```
-> SELECT COUNT(*) INTO param1 FROM t;
```

```
-> END;
```

```
-> //
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> DELIMITER ;
```

Выполним вызов процедуры с помощью оператора CALL:

```
mysql> CALL simpleproc(@a);
```

```
Query OK, 0 rows affected (0.00 sec)
```

С помощью оператора SELECT убеждаемся в том, что выходной параметр param1 был изменен в теле процедуры:

```
mysql> SELECT @a;
```

```
+-----+
```

```
| @a |
```

```
+-----+
```

```
| 3 |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

Пример использования функции. Создадим функцию hello, которой в качестве параметра передается имя s. Функция возвращает строку:

```
mysql> DELIMITER //
mysql> CREATE FUNCTION hello (s CHAR(20))
mysql> RETURNS CHAR(50)
    -> RETURN CONCAT('Hello, ' ,s, '!');
    -> //
Query OK, 0 rows affected (0.00 sec)
mysql> DELIMITER ;
```

Вызов функции выполняется с помощью оператора SELECT:

```
mysql> SELECT hello('world');
+-----+
| hello('world') |
+-----+
| Hello, world! |
+-----+
```

Вызов процедуры

Вызов процедуры выполняется с помощью оператора CALL:

```
CALL ИмяПроцедуры([Параметр[,...]])
CALL ИмяПроцедуры([])
```

Оператор CALL возвращает параметры, переданные как **OUT** или **INOUT**. Если процедура не имеет параметров, она может вызываться так:

CALL p() или CALL p

Например, следующая процедура имеет **OUT** параметр, который возвращает текущую версию сервера и **INOUT** параметр, который увеличивается на единицу в теле процедуры:

```

CREATE PROCEDURE p (OUT ver_param VARCHAR(25),
INOUT incr_param INT)
BEGIN
    # Установить значение OUT параметра
    SELECT VERSION() INTO ver_param;
    # Увеличить значение INOUT параметра
    SET incr_param = incr_param + 1;
END;

```

Все, что располагается правее символа #, рассматривается как комментарий.

Перед вызовом процедуры нужно инициализировать **INOUT** параметр. После вызова процедуры значения обоих параметров будут изменены.

```

mysql> SET @increment = 10;
mysql> CALL p(@version, @increment);
mysql> SELECT @version, @increment;
+-----+-----+
| @version | @increment |
+-----+-----+
| 5.0.25-log | 11      |
+-----+-----+

```

Следующий пример иллюстрирует технику использования вызова хранимой процедуры **p1**, которая имеет два **OUT** параметра, из клиентской программы, написанной на PHP:

```

mysql_query(mysql, «CALL p1(@param1, @param2)»);
mysql_query(mysql, «SELECT @param1, @param2»);
result = mysql_store_result(mysql);
row = mysql_fetch_row(result);
mysql_free_result(result);

```

После вызова процедуры **row[0]** и **row[1]** содержат значения параметров **@param1** и **@param2** соответственно. Для установки

INOUT параметра необходимо установить начальное значение пользовательской переменной, передаваемой в процедуру, перед вызовом **CALL**.

Вызов функции

Так как функция возвращает значение, ее можно использовать в выражениях. Например, создадим функцию с именем `my_version`, которая возвращает символьную строку – текущую версию сервера MySQL. Текст функции приведен ниже:

```
delimiter //  
CREATE FUNCTION my_version()  
RETURNS CHAR(20)  
RETURN Version();  
//  
delimiter ;
```

Вызов функции выполняется двумя способами:

- 1) `SELECT my_version();` (см. рис. 8.1.)

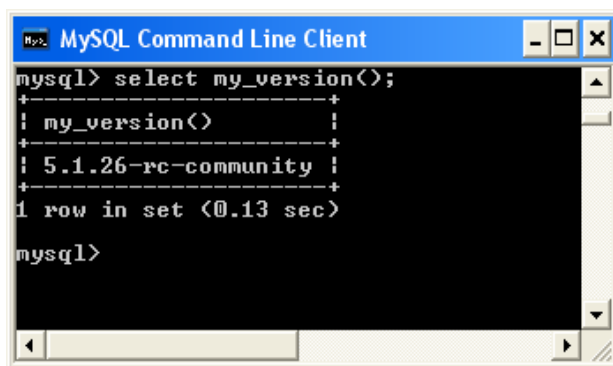


Рис. 8.1.

- 2) `mysql> SET @r=my_version();`
`mysql> SELECT @r;`

Ограничения использования некоторых операторов в хранимых подпрограммах и триггерах

В хранимых подпрограммах не разрешены следующие операторы:

- CHECK TABLE и OPTIMIZE TABLE
- LOCK TABLES и UNLOCK TABLES.
- ALTER VIEW.
- LOAD DATA и LOAD TABLE.
- PREPARE, EXECUTE, DEALLOCATE PREPARE.

В функциях не поддерживаются следующие операторы:

- COMMIT или ROLLBACK
- SELECT должен использоваться только с INTO
- Нельзя использовать рекурсивные функции

Локальные переменные

В теле подпрограмм можно использовать локальные переменные. Они существуют только в теле подпрограммы и извне не видны. Каждая переменная обязательно должна быть описана со своим типом после оператора BEGIN с помощью оператора DECLARE. Синтаксис:

DECLARE *имяПерем*[, ...] Тип [DEFAULT *значение*];

Одним оператором DECLARE можно описать несколько одноименных переменных. Но переменные разных типов описываются каждая с помощью отдельного оператора. Например:

```
DECLARE a,b,c INT DEFAULT 0;  
DECLARE kol INT;  
DECLARE summ, ost FLOAT(10,2);
```

Значение локальной переменной можно присвоить двумя способами:

- С помощью оператора SET.
- С помощью оператора SELECT .. INTO .. FROM .. , причем SELECT используется с фразой INTO.

Например:

```
SET ex=1;
```

```
SELECT count(*), sum(ost) INTO kol, summ FROM Counts;
```

Во втором примере значения count(*) и sum(ost) заносятся в переменные kol и sum, соответственно. Для этого используется фраза INTO.

Операторы, используемые в теле хранимых процедур

В теле хранимых подпрограмм можно использовать операторы ветвлений, а также операторы циклов.

Оператор ветвления

Оператор ветвления, также как это делается в других языках программирования, позволяет изменить порядок выполнения операторов в зависимости от условия. Синтаксис оператора:

```
IF условие1 THEN операторы1  
  [ELSEIF условие2 THEN операторы2] ...  
  [ELSE операторы3]  
END IF
```

Оператор выбора

Оператор выбора позволяет осуществить множественный выбор и имеет две формы. Синтаксис первой формы:

CASE *переменная*

WHEN *значение1* THEN *операторы1*

[WHEN *значение2* THEN *операторы2*] ...

[ELSE *операторы3*]

END CASE

В первой форме оператор сравнивает переменную со значением. Как только соответствие найдено, выполняется необходимая группа операторов, следующая за THEN. Если ни одно соответствие не найдено, выполняется оператор, размещенный после ключевого слова ELSE (если оно присутствует).

Синтаксис второй формы:

CASE

WHEN *условие1* THEN *операторы1*

[WHEN *условие2* THEN *операторы2*] ...

[ELSE *операторы3*]

END CASE

Вторая форма позволяет осуществить сравнение непосредственно в конструкции WHEN. Как только будет найдено первое истинное условие, выполняются операторы, следующие за ключевым словом THEN, и затем осуществляет выход из оператора CASE.

Оператор WHILE

Оператор WHILE реализует цикл с предусловием, т.е. перед входом в цикл проверяется условие. Цикл выполняется до тех пор, пока условие – истина.

Синтаксис оператора:

[*метка:*] WHILE *условие* DO

операторы

END WHILE [*метка*]

Если в теле цикла требуется выполнить более одного оператора, не обязательно заключать их в операторные скобки BEGIN...END, т.к. эту функцию выполняет сам оператор WHILE.

Метка в теле цикла предназначена не только для того, чтобы облегчить чтение кода при очень длинных циклах. Она позволяет осуществить досрочный выход из цикла.

Использование в циклах операторов LEAVE и ITERATE

Для досрочного выхода из цикла предназначен оператор LEAVE, который имеет следующий синтаксис:

LEAVE *метка*

Оператор прекращает выполнение блока, помеченного *меткой*. Под блоком понимаются BEGIN ... END или конструкции циклов: LOOP, REPEAT, WHILE.

Замечание. Оператор LEAVE эквивалентен оператору BREAK в С-подобных языках программирования.

Еще одним оператором, выполняющим досрочное прекращение цикла, является оператор ITERATE, который имеет следующий синтаксис:

ITERATE *метка*

В отличие от оператора LEAVE, оператор ITERATE не прекращает выполнение цикла, он лишь выполняет досрочное прекращение текущей итерации. ITERATE можно использовать только в циклах LOOP, REPEAT и WHILE .

Замечание. Оператор ITERATE эквивалентен оператору CONTINUE в С-подобных языках программирования.

Рассмотрим программу, которая в цикле формирует бинарную последовательность, добавляя к строке две единицы на четных итерациях и две единицы и два нуля на нечетных. Код подпрограммы приведен ниже:

```

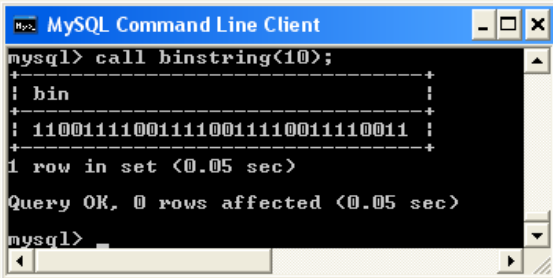
DELIMITER //
CREATE PROCEDURE binstring (IN num INT)
BEGIN
    DECLARE i INT DEFAULT 0;
    DECLARE bin TINYTEXT DEFAULT "";
    IF (num > 0) THEN
        wet: WHILE ( i < num) DO
            SET i=i+1;
            SET bin=CONCAT(bin,'11');
        IF !(i/2 – CEILING(i/2) ) THEN ITERATE wet;
        END IF;
        SET bin = CONCAT(bin,'00');
        END WHILE wet;
        SELECT bin;
    ELSE
        SELECT 'ошибочное значение параметра';
    END IF;
END;
//
DELIMITER ;

```

Вызов процедуры:

```
CALL binstring(10);
```

Результат работы процедуры binstring приведен на рис. 8.2.



```

mysql> call binstring(10);
+-----+
| bin                                     |
+-----+
| 110011110011110011110011110011      |
+-----+
1 row in set (0.05 sec)

Query OK, 0 rows affected (0.05 sec)

mysql>

```

Рис. 8.2.

Оператор REPEAT

Оператор REPEAT реализуется цикл с постусловием. Условие проверяется при выходе из цикла. Таким образом, цикл выполнится хотя бы один раз.

```
[метка:] REPEAT
    операторы
UNTIL условие
END REPEAT [метка]
```

Следует отметить, что цикл выполняется пока условие ложно. Цикл REPEAT может быть снабжен необязательной меткой, по которой легко осуществлять досрочный выход из цикла при помощи операторов LEAVE и ITERATE.

Оператор LOOP

Оператор LOOP предназначен для реализации циклов и имеет следующий синтаксис:

```
[метка:] LOOP
    операторы
END LOOP [метка]
```

Оператор LOOP в отличие от циклов WHILE и REPEAT не имеет условий выхода из цикла. Поэтому данный вид цикла должен обязательно иметь в своем составе оператор LEAVE.

Рассмотрим пример, процедуру doiterate, которая получает входной параметр p1. В теле процедуры выполняется наращивание параметра до 10, затем осуществляется выход. Если значение параметра больше 10, тогда значение параметра наращивается на 1, затем осуществляется выход. Новое значение параметра заносится в переменную @x, значение которой можно просмотреть по выходе из процедуры с помощью оператора SELECT. Код процедуры приведен ниже:

Например:

```
DELIMITER //
```

```
CREATE PROCEDURE doiterate(p1 INT)
```

```
BEGIN
```

```
  label1: LOOP
```

```
    SET p1 = p1 + 1;
```

```
    IF p1 < 10 THEN ITERATE label1;
```

```
    END IF;
```

```
    LEAVE label1;
```

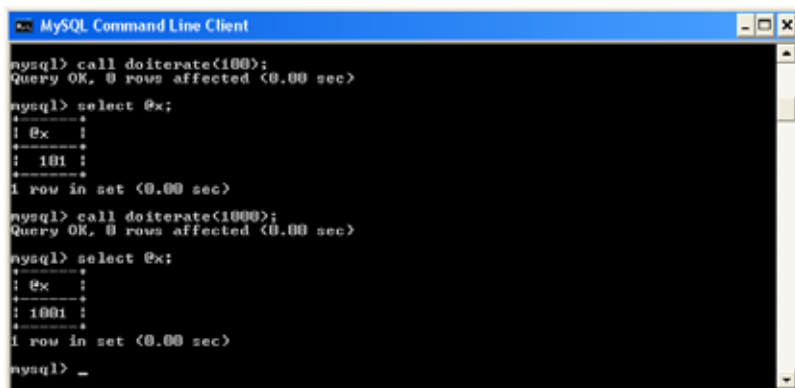
```
  END LOOP label1;
```

```
  SET @x = p1;
```

```
END
```

```
DELIMITER ;
```

На рис. 8.3. показано использование процедуры.



```
mysql> call doiterate(100);
Query OK, 0 rows affected (0.00 sec)

mysql> select @x;
+-----+
| @x    |
+-----+
| 101   |
+-----+
1 row in set (0.00 sec)

mysql> call doiterate(1000);
Query OK, 0 rows affected (0.00 sec)

mysql> select @x;
+-----+
| @x    |
+-----+
| 1001  |
+-----+
1 row in set (0.00 sec)

mysql> _
```

Рис. 8.3.

Просмотр списка хранимых процедур и функций

Информация о хранимых подпрограммах в базе данных может быть получена одним из следующих способов:

- Из таблицы **ROUTINES** базы данных **INFORMATION_SCHEMA**.
- С помощью операторов **SHOW CREATE PROCEDURE** и **SHOW**

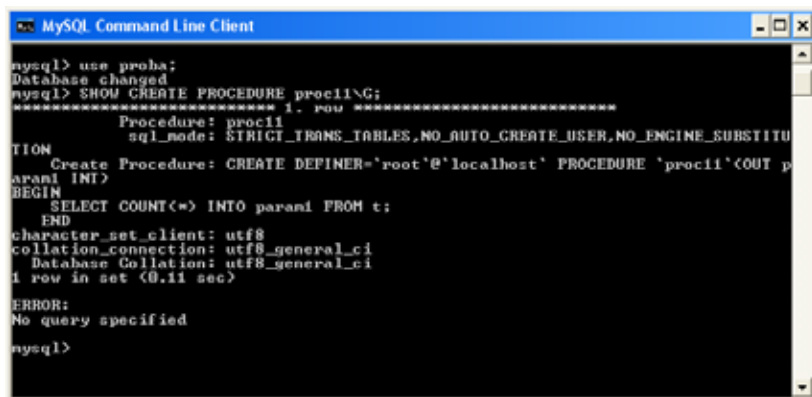
CREATE FUNCTION, которые выдают код заданной хранимой подпрограммы.

- **SHOW PROCEDURE STATUS** и **SHOW FUNCTION STATUS** используются для просмотра списка уже созданных процедур и функций и для просмотра характеристик всех хранимых подпрограмм.

Пример использования **SHOW CREATE PROCEDURE**:

```
SHOW CREATE PROCEDURE proc11\G;
```

Этот оператор позволяет посмотреть код созданной процедуры `proc11`. Пример использования оператора приведен на рис. 8.4.



```
mysql> use proba;
Database changed
mysql> SHOW CREATE PROCEDURE proc11\G;
***** 1. row *****
Procedure: proc11
sql_mode: STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITU
TION
Create Procedure: CREATE DEFINER='root'@'localhost' PROCEDURE 'proc11'(OUT p
param1 INT)
BEGIN
    SELECT COUNT(*) INTO param1 FROM t;
END
character_set_client: utf8
collation_connection: utf8_general_ci
Database Collation: utf8_general_ci
1 row in set (0.11 sec)

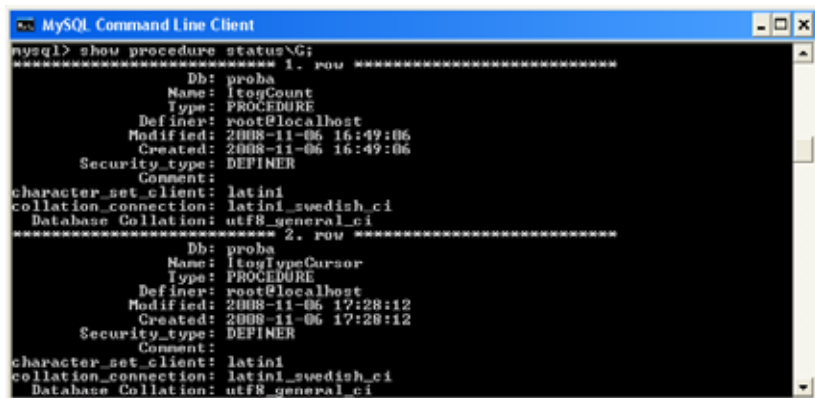
ERROR:
No query specified
mysql>
```

Рис. 8.4.

Пример использования **SHOW PROCEDURE STATUS**:

```
SHOW PROCEDURE STATUS\G;
```

Этот оператор позволяет вывести список всех существующих в базе данных процедур. Пример использования приведен на рис. 8.5.



```
mysql> show procedure status\G;
===== 1. row =====
      Db: proba
      Name: ItogCount
      Type: PROCEDURE
      Definer: root@localhost
      Modified: 2008-11-06 16:49:06
      Created: 2008-11-06 16:49:06
      Security_type: DEFINER
      Comment:
character_set_client: latin1
collation_connection: latin1_swedish_ci
Database Collation: utf8_general_ci
===== 2. row =====
      Db: proba
      Name: ItogTypeCursor
      Type: PROCEDURE
      Definer: root@localhost
      Modified: 2008-11-06 17:28:12
      Created: 2008-11-06 17:28:12
      Security_type: DEFINER
      Comment:
character_set_client: latin1
collation_connection: latin1_swedish_ci
Database Collation: utf8_general_ci
```

Рис. 8.5.

Удаление хранимых процедур и функций

Для удаления хранимых процедур и функций предназначены операторы DROP PROCEDURE и DROP FUNCTION, имеющие следующий синтаксис, соответственно:

DROP PROCEDURE [IF EXISTS] *ИмяПроцедуры*

DROP FUNCTION [IF EXISTS] *ИмяФункции*

Оператор DROP PROCEDURE удаляет существующую процедуру. Если процедура с таким именем не существует, оператор возвращает ошибку, которая подавляется с помощью ключевых слов IF EXISTS. Аналогичным образом работает оператор DROP FUNCTION, который удаляет существующую функцию.

Курсоры

Курсор позволяет в цикле просмотреть каждую строку результирующего набора данных. Работа с курсора происходит по следующему алгоритму.

1. Объявление курсора с помощью оператора DECLARE CURSOR.

2. Открытие курсора с помощью оператора OPEN. Оператор OPEN выполняет запрос, связанный с курсором и устанавливает курсор перед первой записью результирующего набора данных.
3. С помощью оператора FETCH курсор помещается на первую запись набора данных и извлекает данные из записи в локальные переменные. Повторный вызов FETCH приводит к перемещению курсора на следующую запись, и так до тех пор, пока записи набора данных не будут исчерпаны. Эту операцию удобно осуществлять в цикле.
4. Оператор CLOSE прекращает доступ к результирующему набору данных и ликвидирует связь между курсором и этим набором данных.

Ниже приведен пример использования курсора в хранимой процедуре.

Примеры использования хранимых процедур и функций

Пример 1. Отсутствующие идентификаторы

Таблица tb1 имеет первичный ключ id, снабженный атрибутом AUTO_INCREMENT. По мере работы с таблицей некоторые записи удаляются таким образом, что образуются пропуски значений ключевого поля. Таблица tb1 имеет следующую структуру:

```
CREATE TABLE tb1 (  
id INT NOT NULL AUTO_INCREMENT,  
PRIMARY KEY (id));
```

Следующий оператор заполняет таблицу данными:

```
INSERT INTO tb1 VALUES (3), (5), (10), (12), (15), (19), (20), (21),  
(23), (24), (26), (29), (30), (35), (37), (41), (44);
```

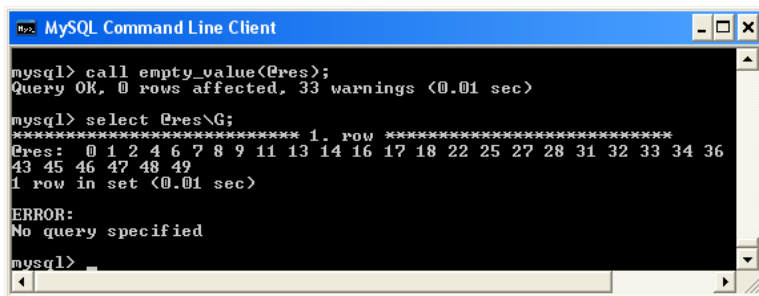
Пусть задача состоит в том, чтобы создать список отсутствующих идентификаторов. Решить эту задачу можно с помощью хранимой процедуры:


```

DELIMITER //
CREATE PROCEDURE empty_value (OUT result TEXT)
BEGIN
    DECLARE i,x INT DEFAULT 0;
    SET result="";
    WHILE i<50 DO
        SET x=NULL;
        SELECT i INTO x FROM DUAL WHERE i IN
        (SELECT id FROM tb1);
        IF (x IS NULL) THEN
            SET result=CONCAT(result, ',i);
        END IF;
        SET i=i+1;
    END WHILE;
END;
//
DELIMITER ;

```

Цикл WHILE будет выполняться 50 раз. DUAL это пустая таблица, для тех случаев, когда не требуется никакая таблица. Каждый раз в теле цикла в переменную x будут записываться значения id, которые присутствуют в таблице tb1. Если значение i не соответствует значению id из таблицы tb1, тогда переменная x равна NULL. В выходной переменной result накапливается строка значений id, которых нет в таблице tb1. Результат выполнения процедуры приведен на рис. 8.6.



```

MySQL Command Line Client
mysql> call empty_value(@res);
Query OK, 0 rows affected, 33 warnings (0.01 sec)

mysql> select @res\G;
***** 1. row *****
@res:  0 1 2 4 6 7 8 9 11 13 14 16 17 18 22 25 27 28 31 32 33 34 36
43 45 46 47 48 49
1 row in set (0.01 sec)

ERROR:
No query specified
mysql>

```

Рис. 8.6.

Пример 2. Создание хранимой процедуры «Открытие счета».

Даны следующие таблицы:

Таблица «Справочник типов счетов» с полями:

Ntype – номер типа счета

Nazv – название типа, например «До востребования», «Срочный на 1 мес», «Срочный на 3 мес» и т.д.

Таблица создается с помощью оператора:

```
CREATE TABLE TypeCounts (  
Ntype INT AUTO_INCREMENT PRIMARY KEY,  
Nazv CHAR(30));
```

Таблица «Клиенты» с полями:

Ncl – номе клиента

Fio – ФИО клиента

Adres - адрес клиента

Tel – телефон клиента.

Таблица создается с помощью оператора:

```
CREATE TABLE Client (  
Ncl INT AUTO_INCREMENT PRIMARY KEY,  
Fio CHAR(50),  
Adres CHAR(50),  
Tel CHAR(20));
```

Таблица «Счета» с полями:

Nc – номер счета клиента, первичный ключ;

Ncl – номер клиента;

Ntype – номер типа счета, например 1 или 2;

Data – дата открытия счета;

Ost – текущий остаток по счету.

Таблица создается с помощью оператора:

```
CREATE TABLE Counts (Nc INT AUTO_INCREMENT PRIMARY KEY,  
Ncl INT, Ntype INT, data DATE, ost FLOAT(10,2));
```

Требуется написать хранимую процедуру «Открытие счета», которая бы записывала информацию о клиенте в таблицу «Клиенты» и создавала новый счет для этого клиента, т.е. добавляла строку в таблицу «Счета».

Входными параметрами для этой процедуры служат ФИО клиента, его адрес, телефон, дата открытия счета, номер типа счета. Выходной параметр только один – это номер сгенерированного счета.

Код этой процедуры имеет следующий вид:

```
DELIMITER //
CREATE PROCEDURE OpenCounts(fio1 CHAR(50),
    adres1 CHAR(50),
    tel1 CHAR(20),
    data1 DATE,
    ntype1 INT,
    ost1 FLOAT(10,2),
    OUT nomer INT)
BEGIN
    INSERT INTO Client (Fio, Adres, Tel) VALUES (fio1, adres1, tel1);
    SELECT LAST_INSERT_ID() INTO nomer;
    INSERT INTO Counts (Ncl, Ntype, Data, ost) VALUES
(nomer,ntype1,data1,ost1);
    END;
//
DELIMITER ;
```

Запись процедуры выполняется в консольном приложении mysql. Для замены разделителя операторов используется оператор DELIMITER, который изменяет разделитель на //. Для восстановления разделителя используется этот же оператор.

В теле процедуры происходит запись строки в таблицу Client. Функция LAST_INSERT_ID() позволяет получить последнее автоматически сгенерированное значение столбца

AUTO_INCREMENT, которое заносится с помощью оператора SELECT в выходную переменную nomer. Последний оператор INSERT добавляет строку в таблицу «Счетов».

Запуск процедуры выполняется с помощью оператора CALL:

```
CALL OpenCounts(«Иванов», «Мира 5», «22-33-44»,  
«2008.11.15»,1, 1000,@nom);
```

Обратите внимание, как передается дата: год, месяц, день(YYYY.MM.DD).

Здесь в выходной переменной @nom мы получим номер сгенерированного счета. Значение номера можно получить с помощью команды SELECT:

```
SELECT @nom;
```

Для проверки корректности работы процедуры можно посмотреть содержимое таблиц:

```
SELECT * FROM Client;  
SELECT * FROM Counts;
```

Пример 3. Создание функции «Открытие счета».

Создать функцию, которая выполняет тоже действие, что и процедура OpenCounts. На вход функции передаются параметры: ФИО клиента, его адрес, телефон, дата открытия счета, номер типа счета. Функция возвращает номер сгенерированного счета.

Решение:

```
DELIMITER //  
CREATE FUNCTION FOpenCounts(fio1 CHAR(50),  
    adres1 CHAR(50),  
    tel1 CHAR(20),  
    data1 DATE,  
    ntype1 INT,  
    ost1 FLOAT(10,2))  
RETURNS INT
```

```

BEGIN
  DECLARE nomer INT;

  INSERT INTO Client (Fio, Adres, Tel) VALUES (fio1,adres1,tel1);
  SELECT LAST_INSERT_ID() INTO nomer;
  INSERT INTO Counts (Ncl, Ntype, Data, Ost) VALUES
(nomer, ntype1, data1, ost1);
  RETURN nomer;
END;
//
DELIMITER ;

```

Выполнить функцию можно двумя способами.

Способ 1:

```

SELECT FOpenCounts( “Петров Т.Н.”, “Ленина 10, 7”,
“789-32-14”, “2008.11.09”, 1, 5500);

```

Способ 2:

```

SET @nomer=FOpenCounts( “Васичкин К.Р.”,
“Жемчужная 4,15”, “889-12-75”, “2008.11.15”,1, 15500);
SELECT @nomer;

```

Пример 4. Использование курсоров в хранимых процедурах.

Требуется посчитать количество счетов каждого типа вклада, суммы на каждом из этих типов счетов, процентное соотношение суммы этого типа вклада ко всем вкладам банка. Вывести этот результат в табличной форме. Например, такой, какая приведена в таблице 13:

Таблица 13

Название типа вклада	Количество вкладчиков	Общая сумма	Процентное соотношение
До востребования	100		
Срочный на 1 мес	1978		
Срочный на 3 мес	1560		
Срочный на 6 мес	2350		

Для выполнения этого задания необходимо создать дополнительную таблицу, в которой будет храниться эта итоговая информация. Назовем таблицу ItogType, она состоит из следующих полей:

Name – название типа вклада;

Kol – количество вкладчиков;

Summ – сумма всех вкладов данного типа;

Proc – процентное соотношение.

Назовем процедуру ItogCount. Процедура не имеет входных параметров. В результате работы процедуры формируется содержимое таблицы ItogType.

Решение:

```
DELIMITER //
CREATE PROCEDURE ItogCount()
BEGIN
DECLARE summ FLOAT(10,2);
DECLARE kol INT;
# удалить все строки из таблицы itogtype
DELETE FROM itogtype;
SELECT sum(ost),count(*) INTO summ,kol FROM Counts;
INSERT INTO itogtype
SELECT nazv, count(*),sum(ost), sum(ost)*100/summ
FROM Counts C INNER JOIN TypeCounts T
ON C.Ntype=T.Ntype
GROUP BY nazv;
END;
//
DELIMITER ;
```

Выполнение процедуры ItogCount с помощью оператора:

CALL ItogCount;

Для проверки корректности работы процедуры можно

посмотреть содержимое таблицы:

```
SELECT * FROM itogtype;
```

Решить ту же задачу с использованием курсоров можно следующим способом:

```
DELIMITER //
CREATE PROCEDURE ItogTypeCursor()
BEGIN
    DECLARE ex INT DEFAULT 0;
    DECLARE kol INT;
    DECLARE summ FLOAT(10,2);
    DECLARE id INT;
    DECLARE name CHAR(50);
    DECLARE sum1 FLOAT(10,2);
    DECLARE kol1 INT;
    DECLARE proc1 FLOAT(5,2);

    # объявление курсора
    DECLARE CurType CURSOR FOR SELECT Ntype,Nazv
    FROM TypeCounts;
    # объявление обработчика
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET ex=1;

    SELECT “название вклада”, “кол-во”, “сумма”, “процент”;
    OPEN CurType;
    SELECT count(*), sum(ost) INTO kol,summ FROM Counts;

    W: LOOP
        FETCH CurType INTO id, name;
        SELECT sum(ost),count(*), sum(ost)*100/summ
        INTO sum1, kol1, proc1 FROM Counts
        WHERE Ntype=id;
        IF ex THEN LEAVE W;
```

```

        END IF;
        SELECT name, koll, sum1, proc1;
    END LOOP W;
CLOSE CurType;
END;
//
DELIMITER ;

```

Объявляется курсор CurType как набор следующего вида:

```
SELECT Ntype, Nazv FROM TypeCounts.
```

Оператор OPEN открывает курсор. Далее в цикле с помощью оператора FETCH выполняется перемещение по строкам курсора до тех пор, пока не будет достигнут конец набора данных. При перемещении по строкам курсора выполняется подсчет суммарного остатка, количество клиентов и процентное соотношение для каждого вида вклада. Насчитанные значения заносятся в переменные sum1, koll, proc1. Далее с помощью оператора SELECT значения переменных sum1, koll, proc1 выводятся на экран. Выход из цикла обеспечивается с помощью обработчика ошибок CONTINUE, объявленного следующим образом:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET ex=1;
```

Это означает, что в случае возникновения ошибки, например, достижения конца набора данных, выполнение текущей операции продолжается, но значение переменной ex становится равным 1, что и обеспечивает выход из цикла.

Запуск процедуры на выполнение с помощью оператора:

```
CALL ItogTypeCursor();
```

Результат выполнения процедуры ItogTypeCursor приведен на рисунке 8.7.


```
MySQL Command Line Client
mysql>
mysql> call Itogtypecursor();
+-----+-----+-----+-----+
| название вклада | кол-во | сумма | процент |
+-----+-----+-----+-----+
| название вклада | кол-во | сумма | процент |
+-----+-----+-----+-----+
1 row in set (0.11 sec)

+-----+-----+-----+-----+
| name | kol1 | sum1 | proc1 |
+-----+-----+-----+-----+
| До востребования | 4 | 27500.00 | 90.16 |
+-----+-----+-----+-----+
1 row in set (0.48 sec)

+-----+-----+-----+-----+
| name | kol1 | sum1 | proc1 |
+-----+-----+-----+-----+
| Срочный на 1 мес | 1 | 3000.00 | 9.84 |
+-----+-----+-----+-----+
1 row in set (0.48 sec)

+-----+-----+-----+-----+
| name | kol1 | sum1 | proc1 |
+-----+-----+-----+-----+
| Срочный на 3 мес | 0 | NULL | NULL |
+-----+-----+-----+-----+
1 row in set (0.48 sec)

+-----+-----+-----+-----+
| name | kol1 | sum1 | proc1 |
+-----+-----+-----+-----+
| Срочный на 6 мес | 0 | NULL | NULL |
+-----+-----+-----+-----+
1 row in set (0.50 sec)
```

Рис. 8.7.

Контрольные вопросы и упражнения

1. В чем преимущество использования хранимых подпрограмм?
2. Чем хранимая процедура отличается от хранимой функции?
3. В чем отличие IN, OUT и INOUT параметров подпрограмм? Какие параметры IN, OUT или INOUT можно использовать в функциях?

4. Какие операторы языка SQL можно использовать в теле подпрограмм?
5. Можно ли в теле подпрограмм использовать ветвления, циклические конструкции?
6. Как подпрограмму записать в базу данных? Для чего используется оператор DELIMITER?
7. Как вызывается процедура? Как вызывается функция?
8. Можно ли в теле подпрограмм использовать переменные? Требуют ли они предварительного описания? Если да, то как это сделать?
9. Как переменной назначить значение?
10. Для чего используется оператор CASE в теле подпрограммы?
11. Для чего используются операторы LEAVE и ITERATE в теле подпрограммы?
12. Как просмотреть созданные процедуры и функции?
13. Что такое курсор, для чего он используется? Порядок работы с курсором.
14. Приведите пример хранимой процедуры, функции.

Задания для самостоятельной работы

Задание 1

Валюта покупается «Обменным пунктом валюты» в обмен на рубли. Будем считать, что код рубля в таблице «Валюта» равен 1. Даны следующие таблицы:

1. Таблица Валюта(Valuta), которая хранит информацию о валютных остатках в обменном пункте, и имеет следующую структуру:

Nval - Номер валюты, первичный ключ,
Nazv - Название валюты,
Ost - текущий остаток валюты в обменном пункте.

2. Таблица Покупка(Pokupka) хранит информацию о покупке валюты обменным пунктом и имеет следующую структуру:

N_pok - Номер покупки, первичный ключ,
Data_pok - Дата покупки,
N_val - Номер валюты, внешний ключ, ссылающийся на первичный ключ таблицы Валюта.
Kurs - Курс,
Kol – Количество купленной валюты.

Требуется разработать процедуру «Покупка валюты» обменным пунктом. Входные параметры: Номер валюты, дата покупки, курс покупки, количество покупаемой валюты. Выходной параметр: результат выполнения процедуры, т.е. выполнена или не выполнена процедура.

В теле процедуры выполняются следующие действия:

1. Если остаток в рублях недостаточен для выполнения процедуры, то выход из процедуры, при этом выходной параметр сигнализирует о том, что процедура не выполнена.
2. Изменяются остатки: остаток в рублях уменьшается, остаток покупаемой валюты увеличивается.
3. Операция записывается в таблицу «Покупка валюты».
4. Выходной параметр сигнализирует о том, что процедура успешно выполнена.

Задание 2

Даны таблицы

1. Абоненты (*номер телефона, ФИО, Адрес) - хранит всю информацию об абонентах телефонной сети. Поле «Номер телефона» - первичный ключ.
2. Тариф (код города, Название города, цена за минуту) – хранит информацию о тарифах, где поле «код города» - первичный ключ.
3. Разговоры(*номер разговора, номер телефона, дата, код города, время) – хранит информацию о разговорах абонентов, где поле «номер разговора» - первичный ключ, а поле «номер телефона» - внешний ключ, который ссылается на первичный ключ таблицы «Абоненты», поле «код города» - внешний ключ, который ссылается на первичный ключ таблицы Тариф.

Задание. Разработать процедуру или функцию, которая по каждому абоненту(номеру телефона) выдает информацию о разговорах этого абонента за период времени с даты₁ по дату₂, например, в следующем виде:

Код города	Название города	Дата	Время	Цена за мин	Сумма

Глава 9

Триггеры

Триггер это специальная хранимая процедура, которая выполняется автоматически при наступлении одного из событий: INSERT, UPDATE или DELETE, т.е. при добавлении, изменении или удалении строк в таблице. Триггер всегда привязывается к конкретной таблице.

Создание триггера

Триггер создается с помощью оператора CREATE TRIGGER. Синтаксис оператора:

```
CREATE TRIGGER ИмяТриггера  
    КогдаСрабатывает Событие  
    ON ИмяТаблицы FOR EACH ROW ОператорыТелаТриггера
```

Событие:

INSERT – триггер привязан к событию вставки новых записей в таблицу.

UPDATE – триггер привязан к событию обновления записей в таблице.

DELETE – триггер привязан к событию удаления записей в таблице.

КогдаСрабатывает – может иметь два значения:

BEFORE – до события

AFTER – после события.

ОператорыТелаТриггера:

BEGIN

...

END;

В теле триггера допускаются все специфичные для хранимых процедур и функций конструкции:

- Другие составные операторы BEGIN... END.
- Операторы управления потоком выполнения (IF, CASE, WHILE, REPEAT, LEAVE, ITERATE).
- Объявления локальных переменных с помощью оператора DECLARE и задание им значений с помощью оператора SET.
- Именованные условия и обработчики ошибок.

Помимо указанных операторов в теле триггера можно использовать так называемые контекстные переменные, т.е. переменные с префиксом NEW и OLD, которые позволяют получать доступ, соответственно, к новому и старому значению переменной. Синтаксис:

NEW.*ИмяПеременной* и OLD.*ИмяПеременной*.

Например:

NEW.Total – позволяет получить доступ к новому значению переменной Total.

OLD.Total – позволяет получить доступ к старому значению переменной Total.

NEW не используется для события DELETE.

OLD не используется для события INSERT.

Для создания триггера требуется специальная привилегия TRIGGER.

Замечание

В СУБД MySQL триггер нельзя привязать к каскадному обновлению или удалению записей из таблицы типа InnoDB по связи первичный ключ/внешний ключ.

Примеры использования триггеров

Пример 1. Создадим триггер, который реализует каскадное удаление для таблиц типа MyISAM. После каждого удаления строки в таблице Otdel выполняется автоматическое удаление соответствующих строк из таблицы Sotr. Код триггера имеет следующий вид:

```
DELIMITER //
CREATE TRIGGER TRDelOtdel AFTER DELETE On Otdel
FOR EACH ROW
BEGIN
    DELETE FROM Sotr WHERE NOtd=OLD.NOtd;
END;
//
DELIMITER ;
```

Пример 2. Пусть заданы таблицы «ЗаполнениеСалонов» и «Билеты».

Таблица «Билеты» имеет имя `bilet` и содержит информацию о проданных билетах на различные авиарейсы. Таблица имеет следующую структуру:

Nb – номер билета, первичный ключ.

Nr – номер рейса.

Data – дата вылета.

FIO – ФИО пассажира.

Comfort – уровень комфортности, может принимать одно из двух значений либо В – бизнес-класс, либо Е – класс-эконом.

Таблица создается с помощью следующего оператора.

```
CREATE TABLE `bilet` (
    `Nb` smallint(6) NOT NULL AUTO_INCREMENT,
    `Nr` smallint(6) NOT NULL,
    `Data` DATE NOT NULL,
```

```

`comfort` enum('B','E') NOT NULL,
`FIO` varchar(30) NOT NULL,
PRIMARY KEY (`Nb`)
) ENGINE=InnoDB AUTO_INCREMENT=1
DEFAULT CHARSET=utf8 ;

```

Таблица «ЗаполнениеСалонов» содержит информацию о том, сколько билетов каждого вида комфортности уже продано на данный рейс на данную дату. Таблица называется Salon и имеет следующую структуру:

Nr – номер рейса;

Data – дата вылета;

KolB – содержит количество проданных билетов бизнес-класса на этот рейс.

KolE – содержит количество проданных билетов класса эконом на этот рейс.

Таблица имеет составной первичный ключ: Nr, Data. Таблица Reis создается с помощью следующего оператора:

```

CREATE TABLE `Salon` (
  `Nr` smallint(6) NOT NULL,
  `Data` date NOT NULL,
  `kolB` smallint(6) DEFAULT NULL,
  `KolE` smallint(6) DEFAULT NULL,
  PRIMARY KEY (`Nr`,`Data`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

Требуется написать триггер на добавление записей к таблице «Билет». Каждый раз при покупке билета пассажиром происходит добавление новой записи к таблице «Билет», которое сопровождается автоматическим выполнением следующего:

- Проверяется, есть ли уже информация об этом рейсе в таблице «ЗаполнениеСалонов». Если нет, то добавляется новая запись в таблицу «ЗаполнениеСалонов».

- К значениям полей KOLB и KOLE таблицы «ЗаполнениеСалонов» добавляется количество купленных билетов, соответственно, бизнес класса и класса эконом.

Код триггера имеет следующий вид:

```

DELIMITER //
CREATE TRIGGER INSBilet BEFORE INSERT
  ON Bilet FOR EACH ROW
BEGIN
  DECLARE nomR SMALLINT;
  DECLARE dataR DATE;
  DECLARE kolE1 SMALLINT;
  DECLARE kolB1 SMALLINT;
  SELECT Nr, Data INTO nomR, dataR FROM Salon
    WHERE Nr=NEW.Nr AND data=NEW.data;
  SET kolB1=0;
  SET kolE1=0;
  IF (NEW.comfort='B') THEN
    SET kolB1=1;
  ELSE
    SET kolE1=1;
  END IF;
  IF (dataR IS NULL) THEN
    INSERT INTO Salon (Nr, data, kolB, kolE)
      VALUES (NEW.Nr, NEW.data, kolB1, kolE1);
  ELSE
    UPDATE Salon SET kolB=kolB+kolB1, kolE=kolE+kolE1
      WHERE Nr=nomR AND data=dataR;
  END IF;
  END;
//
DELIMITER ;

```

Триггер срабатывает автоматически при добавлении записи к таблице Билет:

```
> INSERT INTO Bilet (Nr, data, FIO, comfort)
VALUES (3337, "2008.11.20", "Иванов И И", "В");
```

Для проверки работы триггера следует просмотреть содержимое таблицы Salon.

Удаление триггера

Для удаления триггера предназначен операторы DROP TRIGGER, имеющий следующий синтаксис:

```
DROP TRIGGER ИмяТаблицы.ИмяТриггера
```

Оператор DROP TRIGGER удаляет существующий триггер для данной таблицы.

Просмотр существующих триггеров

Получить список существующих триггеров можно при помощи оператора SHOW TRIGGERS, который имеет следующий синтаксис:

```
SHOW TRIGGERS [FROM ИмяБазыДанных ] [LIKE Шаблон ]
```

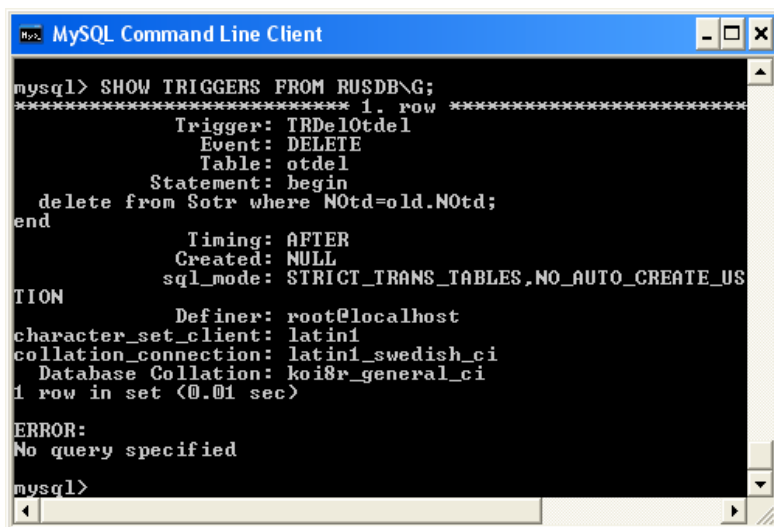
Оператор позволяет извлечь список и характеристики триггера из заданной базы данных. Если база данных имеет большое количество триггеров, вывод можно ограничить при помощи ключевого слова LIKE, имеющего тот же синтаксис, что и ключевое слово LIKE в операторе SELECT. На рис. 8.1 показано использование команды SHOW TRIGGERS для базы данных RUSDB. Наряду с именем триггера, выводится событие и таблица, к которой привязан триггер, а также тело триггера:

BEGIN

DELETE FROM Sotr WHERE NOtd=OLD.NOtd;

END

Указан момент срабатывания триггера: AFTER.



```
mysql> SHOW TRIGGERS FROM RUSDB\G;
***** 1. row *****
      Trigger: TRDel0tdel
        Event: DELETE
         Table: otdel
    Statement: begin
delete from Sotr where NOtd=old.NOtd;
end
      Timing: AFTER
      Created: NULL
    sql_mode: STRICT_TRANS_TABLES,NO_AUTO_CREATE_US
TION
      Definer: root@localhost
character_set_client: latin1
collation_connection: latin1_swedish_ci
      Database Collation: koi8r_general_ci
1 row in set (0.01 sec)

ERROR:
No query specified
mysql>
```

Рис. 8.1.

Контрольные вопросы и упражнения

1. Что такое триггер?
2. Для чего используются контекстные переменные в теле триггера?
3. Можно ли в теле триггера использовать локальные переменные?
4. Можно ли в теле триггера использовать операторы ветвлений и циклические операторы?
5. С помощью каких команд можно создать триггер? Как удалить триггер?
6. Каким образом посмотреть какие триггеры уже созданы в базе данных?
7. Приведите примеры триггеров.

Задания для самостоятельной работы

Задание 1

Создать таблицы следующей структуры:

1. Квартиры(*НомерКвартиры, Адрес, Телефон, КолвоКомнат, Этаж, Район, Владелец, Стоимость, ПризнакПродажи), если квартира не продана, то ПризнакПродажи=0, если продана, то 1.
2. Сделка(*НомерСделки, Дата, СуммаСделки, ФИО_Реелтера)

Написать триггер, который каждый раз при добавлении записи в таблицу «Сделка», помечает квартиру в таблице «Квартиры» как проданную, т.е. поле «ПризнакПродажи» устанавливается равным 1.

Задание 2

Отпуск товара со склада выполняется по документу «Накладная», который имеет шапку и строки. В шапке накладной содержится информация о том, кому, кем и когда отпускается товар со склада. В строках накладной содержится информация о тех товарах, которые отпускаются по этой накладной. По одной накладной можно отпустить несколько разных товаров со склада.

Информация о накладных хранится в двух таблицах: «ШапкаНакладной» и «СтрокиНакладной». Таблицы имеют следующую структуру:

1. ШапкаНакладной(*НомерНакл, Поставщик, Покупатель, ДатаОтпуска, СуммаПоНакл)
2. СтрокиНакладной(*НомерЗаписи, НомерНакл, НазваниеТовара, Цена, Колво, ЕдИзм).

Где

* - отмечены первичные ключи

ЕдИзм – единица измерения товара: штуки, пачки, ящики и т.д.

СуммаПоНакл – сумма товара по всей накладной, считается как сумма произведения Цена* Колво по всем строкам данной накладной.

Поле «НомерНакл» в таблице «СтрокиНакладной» является внешним ключом и ссылается на первичный ключ таблицы «ШапкаНакладной».

Задание. Создать таблицы. Написать триггеры, которые бы при изменении, удалении, добавлении строк в таблице «СтрокиНакладной» автоматически обновляли поле СуммаПоНакл в таблице «ШапкаНакладной».

Литература

1. М. Кузнецов, И. Симдянов «MySQL на примерах», С-Пб, 2007
2. www.mysql.com – электронная документация
(на английском языке) по языку SQL СУБД MySQL
3. www.mysql.ru – подборка статей по языку SQL СУБД MySQL
4. Дейт К. «Введение в системы баз данных». М: Наука, 1980.
5. Дейт К. «Руководство по реляционной СУБД DB-2». М: Финансы и статистика, 1988.
6. Грабер М. «Введение в SQL», 2000.
7. Дюбуа П. «MySQL», М, СПб, Киев, 2007

Татьяна Александровна Иваньчева

Методическое пособие по языку SQL

(Диалект СУБД MySQL)

Часть 2