Въведение в Базите от данни. SQL. MySQL

Дефиниция

База от данни (БД) представлява колекция от логически свързани данни в конкретна предметна област, които са структурирани по определен начин.

Причини за използването на БД:

1. Намалява се обема на съхраняваните данни: Много често се налага да съхраняваме една и съща информация на различни места (например личните данни за даден служител са нужни както в счетоводството, така и в личен състав). В базите от данни се постига интегриране на информацията и се намалява повторението ѝ.

2. Цялостност на данните: В базите от данни има възможност да се направят ограничения, които да позволяват да се записват само коректни данни. Осигурява се интерфейс, чрез който се контролира записът на информацията. По този начин си гарантираме, че няма противоречиви данни и записаната информация е правдоподобна.

3. Независимост на информацията: Чрез базите от данни информацията се изолира от програмния код на софтуера. Освен това е възможно повече от един софтуерен продукт да достъпва тази информация, което позволява лесно реализиране на многозадачност.

4. Сигурност на данните: Системите за управление на бази от данни имат своя собствена защита срещу злонамерени действия или инциденти. Пряко свързано с предишната точка, това осигурява надеждност за това, че дори ако даден програмен продукт се повреди, то информацията няма да се загуби.

Така разгледани можем да приемем, че базите от данни са едно унифицирано хранилище на информация със специален режим на достъп (само автентикираните софтуерни продукти ще имат достъп до информацията). Те предлагат централизирано управление на съхранената информация и независимост на данните от програмния код.

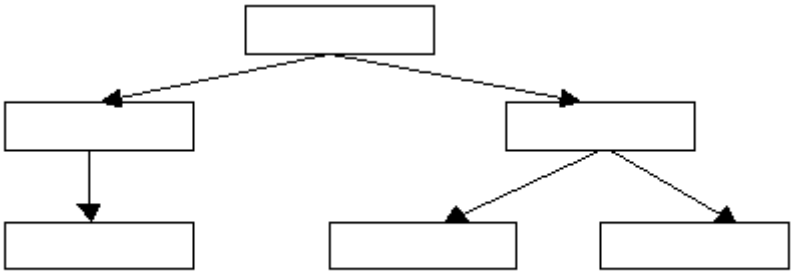
Модели на бази от данни

Описва как са структурирани данните в БД и какви са основните операции за манипулиране с тях.

- *Плосък (flat)*
  Цялата информация се съхранява в една таблица. Възможно е повторение на редица елементи.

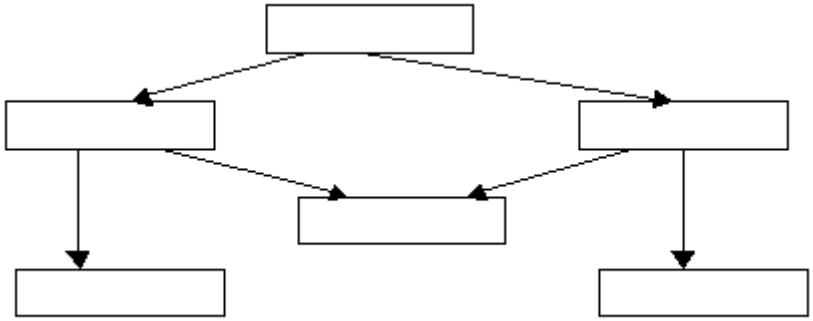
- *Йерархичен*
  Данните се съхраняват в предварително определена йерархия. Една единствена таблица в базата данни играе ролята на корен на обърнатото дърво.

Релацията в една йерархична база данни е представена от термините родител/наследник. При тази релация всяка родителска таблица може да бъде асоциирана с повече от една дъщерни таблици, но една дъщерна таблица може да бъде асоциирана само с една родителска таблица. Тези таблици са изрично свързани чрез указател или чрез физическата подредба на записите в таблиците.
Даден потребител осъществява достъп до данните в рамките на този модел, като започва от таблицата-корен и обхожда дървото надолу, докато достигне желаните данни.



- *Мрежови*

  Представянето на данни и връзките между тях се извършва чрез граф, върховете на който са типове записи, а дъгите – типове връзки. Графичното изображение на графа се нарича структурна диаграма на данните.
  Моделът се представя с помощта на възли и свързващи структури. Всеки възел представя колекция от записи, а всяка свързваща структура създава и представя релация в мрежова база данни. Чрез структурната диаграма се свързват два възела, като единият възел е собственик, а другият възел е член. Всяка свързваща структура поддържа релация от тип едно към много, което означава, че един запис във възела собственик може да бъде свързан с един или повече записи във възела-член, но един единствен запис във възела-член може да бъде свързан само с един запис във възела-собственик.
  Освен това един запис във възела-член не може да съществува без да е свързан със съществуващ запис във възела-собственик.



- *Релационен*

  Най-общо казано релационните бази от данни групират информацията по характерни признаци. За целта се използват доста близки до

човешкото разбиране структури – таблици. Всяка таблица се състои от редове и колони. Всяка колона има наименование, което определя данните въведени в нея по редове, както и тип на тези данни. Обикновено имената и типа на колоните се наричат атрибути, а редовете/записите в таблицата – кортежи.
Както подсказва името на този модел на бази от данни, освен че организира информацията в табличен вид, той въвеждат и връзки (релации) между различните таблици.

Уникалното свойство за всеки обект, което еднозначно го идентифицира спрямо другите обекти в даден клас, се нарича ключ. В релационните бази от данни това означава, че всяка колона, чийто стойности по редове са различни, може да бъде "ключова". Например ако имаме таблица с име, факултетен номер и ЕГН на студент може да се каже, че ключови са колоните "факултетен номер" и "ЕГН", тъй като те са уникални. Името не може да се приеме за ключов атрибут, тъй като е възможно да съществуват двама студенти с напълно еднакви имена. В релационните бази от данни обикновено се избира един от ключовете за дадена таблица и се прави ***първичен ключ*** (primary key). Той е основен инструмент за осъществяване на релации. Първичен ключ може да е съставен както от един, така и от множество атрибути на дадена таблица.

Данните по редове в релационните бази от данни се записват с определен тип. Често не е възможно да се запише дадена информация (например в момента не знаем презимето на даден служител). За целта се използва специална стойност на полето – ***NULL***. Едно от основните ограничения в релационните бази от данни е, че ***първичните ключове не могат да приемат стойност NULL***!

Външен ключ (foreign key) в релационна база от данни наричаме атрибут на дадена таблица, който съответства на първичен ключ в друга (с други думи външния ключ на таблица А трябва е сравним с първичния ключ на таблица В). Можете да приемете външния ключ като указател към първичен ключ.

Външните ключове могат да приемат и стойност NULL. Ограничението за външен ключ в релационна база от данни е, че той може да приема или стойност NULL, или стойност равна на стойността в съответстващия му първичен ключ.

- *Обектно-релационен*
  Допълване на релационен модел с някои възможности за работа със структури на данни, които са познати от обектно ориентираните езици за програмиране (C++, Java). Този модел се нарича хибриден и се стреми да съчетае предимствата на двата основни подхода – релационен и обектен.
  В обектно-релационния модел релацията отново е фундаментална концепция. За разлика от релационния модел, в обектно-релационния модел са добавени следните пет нови характеристики:
  1. Въвеждат се структурирани типове за атрибутите.

2. Въвеждат се методи, които се обвързват с релациите и могат да се прилагат към кортежите на релациите.
3. Въвеждат се идентификатори на кортежи - в обектно-релационните системи кортежите играят ролята на обекти. В определени ситуации е полезно кортежите да се идентифицират уникално. По принцип идентификаторите на кортежи са невидими за потребителя, но понякога той може да има достъп до тях.
4. Въвеждат се reference към кортежи, които се използват по различни начини в обектно-релационните системи.

Базите от данни се управляват от специален софтуер, който ще наричаме **СУБД** (система за управление на бази от данни). СУБД представлява мощен инструментариум за създаване и ефективно управление на големи обеми от данни. Данните трябва да се поддържат и съхраняват за толкова време за колкото е необходимо. Освен това, те трябва да се предпазват от неправилен достъп, който може да наруши целостта им (integrity), както и от неправомерен достъп (security).

Примери:

MS SQL Server, Oracle Database, IBM DB2, MySQL, PostgreSQL

Ще въведем и още две нови понятия, свързани със структурата на базата от данни:

1. ***Data Definition Language (DDL)*** се използва за създаване на схема (описание) на данните. Чрез него абстрактно се дефинира структурата на информацията в базата от данни и се дефинират релации между различните компоненти. Основна грижа на DDL също е да дефинира ограниченията за цялостност на информацията (с цел постигане на интегритет) и да се проектира сигурността на базата от данни.

2. ***Data Manipulation Language (DML)*** са езиковите средства за обработка на данните. Най-често те се разделят на вътрешни (езика използван от сървъра за базата от данни, например SQL) и външни (езиците за програмиране, например C/C++, Java, и т.н.). Както подсказва самото име, чрез тези езици се подават команди, чрез които сървъра обработва данните.

SQL е съкращение от ***Structured Query Language***. Това е език за обработка на бази от данни, който отдавна се е наложил като световен стандарт.

Езикът **SQL** най-общо казано се използва са създаване на бази от данни, създаване на таблици и връзките между тях, вмъкване на информация, извличане на информация, промяна на информация и изтриване на информация от таблици в база от данни. Въпреки съществуването на официален стандарт на езика, почти всяка система за управление на бази от данни има своя собствена имплементация. Почти винаги се спазват основните правила в езика, но често се добавя допълнителна функционалност, която е специфична за използваната СУБД.

SQL се базира на последователност от заявки (команди). Когато част от тези заявки са зависими една от друга говорим за понятието ”транзакция”. Вече споменахме за **DDL** и **DML**. Ето как най-общо са въведени тези понятия чрез езика **SQL**:

1. Data Definition Language:
- CREATE DATABASE – създава база от данни
- ALTER DATABASE – променя дефиницията на база от данни
- DROP DATABASE – изтрива база от данни
- CREATE TABLE – създава таблица в база от данни
- ALTER TABLE – променя дефиницията на таблица в база от данни
- DROP TABLE – изтрива таблица от база от данни
- CREATE INDEX – създава индекс
- DROP INDEX – изтрива индекс

2. Data Manipulation Language:
- INSERT – вмъква информация в база от данни
- SELECT – извлича информация от база от данни
- UPDATE – променя информация в база от данни
- DELETE – изтрива информация от база от данни

Въпреки, че SQL би трябвало да е унифициран език, всяка система за управление на бази от данни се различава от другите. Много рядко се спазва истинския ANSI стандарт на SQL и често има съществени различия.

Както споменахме по-горе, релационните бази от данни съхраняват информацията в таблици. Таблиците се състоят от редове, като всеки ред е един запис в базата, а клетките от таблицата са отделните полета в него. Таблиците имат по една или повече колони, като всяка колона има име и тип. Колоните определят съответните полета в записите от базата. Всяка колона определя типа на поле от запис или типа данни, с които можем да запълним конкретните полета от тази колона.

P_Id	LastName	FirstName	Address
1	Hansen	Ola	Timoteivn 10
2	Svendson	Tove	Borgvn 23
3	Pettersen	Kari	Storgt 20

За целите на този курс, ще работим с ***MySQL***. Ето защо е нужно да се запознаем с някои основни характеристики на тази СУБД, преди да продължим с практическите примери, а именно – **типовете данни в MySQL**.

1. Цели числа:

При всички целочислени типове параметърът length е незадължителен. Чрез него можете да ограничите броя на цифрите на числото. Показаните интервали са валидни само ако length не е указан.

а) **TINYINT(length)** – цяло число в интервала [-128, 127]. Ако се добави параметър unsigned става от [0, 255].

б) **SMALLINT(length)** – цяло число в интервала [-32768, 32767] или [0, 65535] като unsigned.

в) **MEDIUMINT(length)** – цяло число в интервала [-8388608, 8388607] или [0, 16777215] като unsigned.

г) **INT(length)** – цяло число в интервала [-2147483648, 2147483647] или [0, 4294967295] като unsigned.

д) **BIGINT(length)** – цяло число в интервала [-9223372036854775808, 9223372036854775807] или [0, 18446744073709551615] като unsigned.

е) **BIT(length)** – приема две стойности – 0 или 1 – ако не се зададе length или "N" на брой бита, ако се зададе length=N. Максимума е 64 бита.

ж) **BOOL, BOOLEAN** – синоними на TINYINT(1). Работи със стойности 0 (false) или която и да е друга цифра (true).

з) **INTEGER(length)** – синоним на INT.

и) **SERIAL** – синоним на BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE.

2. ***Числа с плаваща запетая***:

а) **FLOAT(length, dec)** – стандартно число с плаваща запетая. Параметърът length ограничава броя на цифрите общо, а dec ограничава броя на цифрите след десетичната запетая.

б) **DOUBLE(length, dec)** – същото като FLOAT, но с двойна точност.

в**) DECIMAL(length, dec)** – това число е със същата точност като DOUBLE, но е записано по такъв начин, че различните системи ще прочетат числото с неговата оригинална точност (нещо, което не е вярно за double, където винаги приемаме, че при четене е възможна "загуба" поради различна прецизност на закръгляване). За сметка на това предимство, работата с decimal е по-бавна отколкото с double.

г) **NUMERAL(length, dec), DEC(length,dec), FIXED(length,dec) –** синоними на DECIMAL.

3. ***Текстови типове***:

а) **CHAR(length)** – Съхранява символи с фиксирана дължина. Параметърът length може да бъде от 1 до 255 (по подразбиране 1), което представлява броят символи от въпросната кодировка. Дори да не използвате целия капацитет на полето, то винаги заема максималната си дължина памет.

б) **VARCHAR(length)** – Символни низове с променлива дължина. Параметърът length указва максималния брой символи, а реално се заема точно толкова памет, колкото символа се използват. Теоретично може да е от 1 до 65535 символа, но практически е ограничен от максималната дължина на ред в MySQL (общият сбор байтове, които заемат всички колони), която е 65535 байта! Във версии на MySQL, които са по-стари от 5.0.3, може да се съхраняват не повече от 255 символа.

в) **TINYTEXT** – може да съхранява до 255 символа. TEXT полетата работят както VARCHAR, но за разлика от него самите данни се съхраняват като обекти извън самата таблица, а в таблицата се пазят само референции към тях. По принцип се смята, че работата с полета от тип TEXT е по-бавна. За сметка на това „облекчават“ самата таблица откъм обем и работата с други данни от нея става много по-бърза. Също така те използват малко по-малко памет, но създават риск от фрагментация.

г) **TEXT** – 65535 символа.

д) **MEDIUMTEXT** – 16777215 символа.

е) **LONGTEXT** – 4294967295 символа.

ж) **BINARY(bytes)** – поле подобно на CHAR, но с тази разлика, че в него текстовите низове се записват като бинарна поредица и нямат кодировка. Това практически означава, че в последствие сравнението между такива низове ще се извършва байт по байт вместо символ по символ. Незаетите символи се запълват с 0x00 (нулевия байт). С BINARY обикновено се работи доста по-бързо, отколкото с CHAR. Недостатък в този тип данни се явява липсата на collation (правила за сравнение на низове). Поради тази причина е препоръчително да използвате този тип данни само за бинарна информация или текст, който ще използвате само за четене или пряко сравнение байт по байт.

з) **VARBINARY(bytes)** – аналогично по действие с VARCHAR, но също както BINARY работи с поредица от байтове и сравненията между записани текстови низове ще се правят байт по байт, а не символ по символ. И по същия начин работата с VARBINARY е по-бърза, отколкото с VARCHAR, но липсата на collation го прави неподходящ при сложни сравнения между низове.

и) **TINYBLOB** – Същото като TINYTEXT, но съхранява 255 байта, а не символи.

й) **BLOB** – Същото като TEXT, но съхранява байтове, а не символи.

к) **MEDIUMBLOB** – Същото като MEDIUMTEXT, но съхранява байтове, а не символи.

л) **LONGBLOB** – Същото като LONGTEXT, но съхранява байтове, а не символи.

4. ***Типове за дата и време***:

а) **DATE** – Дата във формат YYYY-MM-DD. Минималната стойност е ’1000-01-01′, а максималната ’9999-12-31′.

б) **DATETIME** – Дата и време във формат YYYY-MM-DD HH:MM:SS. Минималната стойност е ’1000-01-01 00:00:00′, а максималната ’9999-12-31 23:59:59′.

в) **TIMESTAMP** – Дата и време във формат YYYYMMDDHHMMSS. Отговаря на “Unix епохата”, т.е. започва от ’1970-01-01 00:00:01′ UTC и завършва максимално в ’2038-01-19 03:14:07′.

г) **TIME** – Време във формат HH:MM:SS.

д) **YEAR(length)** – Стойността на length може да е 2 или 4, като по подразбиране е 4. Ако е 4, то това поле представлява година от ’1901′ до ’2155′ или стойността ’0000′ (години извън диапазона ще се приемат автоматично за нулевата стойност). Ако length е 2, то възможните стойности са от “70 до 69″ – числата от 70 до 99 представляват годините от 1970 до 1999, а числата от 0 до 69 представляват годините от 2000 до 2069.

5. **Други типове данни**:

а) **ENUM(“o1″,”o2″, ...)** – Изброим тип данни. Записва се само една стойност от изброените. Може да има максимум 65535 различни стойности.

б) **SET(“o1″,”o2″, ...)** – Също като enum е изброим тип данни, но позволява да се записват повече от една стойност в клетка на таблица (това всъщност е единствения тип данни, които позволява това). Може да съдържа максимум 64 елемента.

Създаване на база от данни

Командата за създаване на база от данни към системата на управление има следния синтаксис:

```
CREATE DATABASE <име на базата от данни>;
```

От примера ни по-горе с университет, това може да бъде:

```
CREATE DATABASE university;
```

Повечето системи за управление на бази от данни предоставят възможност за допълнителни характеристики за базата от данни, като например разположение на файла, максимален размер, кодировка за низове по подразбиране и т.н.

За създаването на таблици се използва следният опростен формат:

```
CREATE TABLE <име на таблица>
   (
       <име на колона> <тип данни>,
       <име на колона> <тип данни>,
       ...
       <име на колона> <тип данни>
   );
```

Например, за да създадем таблица “студенти”, в която запазваме лично, бащино и фамилно име на студент, неговия телефон, адрес и факултетен номер, можем да изпълним следното:

```
CREATE TABLE `university`.`students` (
     `firstname` TINYTEXT NOT NULL ,
```

```
        `middlename` TINYTEXT NULL ,
        `lastname` TINYTEXT NOT NULL ,
        `phone` VARCHAR( 32 ) NULL ,
        `address` TEXT NULL ,
        `faknum` BIGINT( 12 ) NOT NULL ,
        `id` INT NOT NULL AUTO_INCREMENT ,
        PRIMARY KEY ( `id` )
    ) ENGINE = MYISAM ;
```

Както забелязвате от примера можем да добавяме и допълнителни параметри. В случая указахме, че някои от колоните могат да не приемат стойност (да бъдат NULL), а други не (NOT NULL). Освен това указахме, че първичният ключ ще бъде полето id. То от своя страна е "AUTO_INCREMENT", което значи, че ще се записва автоматично при изпълнение на заявка за попълване на данни и освен това всеки нов ред ще съдържа стойност по-голяма с 1 от предишния запис. ENGINE = MYISAM е специфично за MySQL допълнително условие, което указва, че базата от данни ще се запише в MYISAM хранилище. Всяка MyISAM таблица се разполага на три физически файла:

.FRM за дефиниция на таблицата;
.MYD (MYData) за запис на информацията;
.MYI (MYIndex) за индекси.

MyISAM не поддържа транзакции и затова тенденцията е да бъде заменен с InnoDB (от версия 5.5 нататък хранилището по подразбиране ще е именно това). Затова ние ще се фокусираме именно към втората система. При нея стандартно всички бази данни на системата се съхраняват в един общ файл. При добавяне на параметър "innodb_file_per_table" в конфигурационния файл my.cnf може да се направи така, че всяка таблица да се разполага в отделен файл, подобно на MyISAM.

Що се отнася до FOREIGN KEY (засега са налични само при InnoDB, но не и за MyISAM), нека разгледаме следният пример от официалната документация:

```
CREATE TABLE parent (
     id INT NOT NULL,
     PRIMARY KEY (id)
) ENGINE=INNODB;

CREATE TABLE child (
     id INT NOT NULL,
     parent_id INT NULL,
     FOREIGN KEY (parent_id) REFERENCES parent(id)
               ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=INNODB;
```

Това е елементарен пример, в който създаваме две таблици, като едната е зависима от другата. В случая таблицата parent е клас, в който се записва единствена стойност "id" от тип INT. Таблицата child е негов подклас, като съдържа собствено (независимо) поле id и второ поле parent_id – и двете от тип INT.

Чрез реда “FOREIGN KEY (parent_id) REFERENCES parent(id)” сме указали, че полето parent_id от таблица child ще бъде външен ключ, който сочи към полето id на таблица parent. Понеже този външен ключ не е “unique” (ключовата дума в дефиницията на колоната parent_id липсва), то в тази колона може да има повторения на стойности, следователно връзката е едно към много. Записа “ON DELETE CASCADE” означава, че при изтриване на конкретно “id” от таблица parent ще бъдат изтрити всички записи от таблица child, чийто външен ключ “сочи” към него. Алтернативата е “ON DELETE SET NULL” (при изтриване на ред от parent ще бъде изтрита стойността на съответния външния ключ в таблицата child) или “ON DELETE RESTRICT” (забранява се изтриването на редове от parent ако има външен ключ към тях в child – това се използва по подразбиране ако не е указано друго).

Към всяка колона в таблиците можете да добавяте и следните ограничения (CONSTRAINTS):

NULL или NOT NULL: указва дали е позволено на клетка от тази колона да не приема данни;

UNIQUE: указва дали е възможно да има повторение на данни в колоната. Прилага се винаги върху външен ключ при моделиране на връзки 1:1. Полета, които са дефинирани като PRIMARY KEY са винаги NOT NULL и UNIQUE;

DEFAULT: указва стойност по подразбиране (ако при вмъкване на информация не подадем стойност). Например “salary DOUBLE NOT NULL DEFAULT 600”;

CHECK: прави проверка на информацията по зададен критерии. Например “salary DOUBLE CHECK (salary > 240)” ще прави проверка дали не сме записали заплата по-ниска от минималната възможна. Засега MySQL приема клаузите в CHECK ограниченията, но в последствие не ги използва.

Промяна на съществуващи таблици

Нека вече имаме съществуващи таблици, които обаче бихме искали да променим. Да изтриваме и създаваме наново дадена таблица би било непрактично. Поради тази причина са въведени команди за промяна на таблица.

Нека разгледаме предишния пример. Искаме да добавим полета за среден успех в таблицата “zadochnici” и “students”. Ще изпълним следната команда:

```
ALTER TABLE `university`.`zadochnici`
ADD `sruspeh` FLOAT( 3 ) NULL;

ALTER TABLE `university`.`students`
ADD `sruspeh` FLOAT( 3 ) NULL;
```

В следващия момент обаче ще се досетим, че така се нарушава логическата връзка на таблиците. Знаем, че таблицата “zadochnici” е подклас на таблицата “students”, т.е. напълно нелогично е да пазим оценките на задочниците два пъти. Затова ще изтрием така създаденото поле:

```
ALTER TABLE `university`.`zadochnici`
DROP `sruspeh`;
```

Ако искаме да променим името и/или типа данни на дадена колона, то използваме аналогичен синтаксис:

```
ALTER TABLE `university`.`students`
CHANGE `sruspeh` `sr_uspeh` FLOAT( 3 ) NULL;
```

Тук променихме името на колоната “sruspeh” в “sr_uspeh” в таблица “students”.

Чрез `ALTER TABLE` можете също да слагате и премахвате ключове. Това става чрез командите “`ADD CONSTRAINT`” и “`DROP CONSTRAINT`”.

Вмъкване на данни

След като вече знаем как се създава база от данни е време да научим как се вмъква информация в нея. За целта съществува команда “`INSERT INTO`”. Нека вкараме имената на два университета в таблица “university” от база от данни “university”:

```
INSERT INTO `university`.`university` (`id` , `name` , `founded`)
VALUES ( NULL , 'Sofia University', '1889-01-29' );

INSERT INTO `university`.`university` (`id` , `name` , `founded`)
VALUES ( NULL , 'Techical University', '1941-06-12' );
```

Виждате, че на полета `id` зададохме стойност NULL, но ако прелистим базата ще видим, че то не е. Това е така, поради дефиницията на полето `id` от таблица `university` – там бяхме посочили, че то е “auto_increment“.

Досещате се, че при добавяне на още един запис той ще получи id = 3. Поради тази причина такива колони се използват изключително често като уникален идентификатор и съответно първичен ключ.

Възможно е добавянето на повече от един ред с една INSERT заявка:

```
INSERT INTO `university`.`faculties` (`id` ,`name` ,`univ_id` ,`dekan_id`)
VALUES ( NULL , 'Avtomatika', '2', NULL ),
  ( NULL , 'Elektronna Tehnika i Tehnologii', '2', NULL ),
  ( NULL , 'Elektrotehnicheski', '2', NULL ),
  ( NULL , 'Energo-mashinostroitelen', '2', NULL ),
  ( NULL , 'Kompiutarni sistemi i upravlenie', '2', NULL ),
  ( NULL , 'Mashino-tehnologichen', '2', NULL );

INSERT INTO `university`.`faculties` (`id` ,`name` ,`univ_id` ,`dekan_id` )
VALUES ( NULL , 'Biologicheski', '1', NULL ),
```

```
( NULL , 'Himicheski', '1', NULL ),
( NULL , 'Matematika i informatika', '1', NULL ),
( NULL , 'Istoricheski', '1', NULL ),
( NULL , 'Fizicheski', '1', NULL );
```

Заявки за еднотабличен оператор SELECT

Заявките SELECT имат за цел да четат информация от таблица по дадени критерии. Най-общо казано синтаксисът е:

```
SELECT <редове от колони> FROM <таблица> WHERE <условие>;
```

Ще демонстрираме как се използва с няколко примера с базата данни от миналата тема:

1. Извежда списък с имената и датата на основаване на всички университети:

```
SELECT name, founded
FROM university.university;
```

Резултатът ще бъде следният:

```
+---------------------+------------+
| name                | founded    |
+---------------------+------------+
| Sofia University    | 1889-01-29 |
| Techical University | 1941-06-12 |
+---------------------+------------+
```

В тази заявка пропуснахме клаузата WHERE, т.е. ще извикаме списък от всички редове.

2. Списък на id и име на факултетите от университет с id=2:

```
SELECT id, name
FROM university.faculties
WHERE univ_id=2;
```

```
+----+----------------------------------+
| id | name                             |
+----+----------------------------------+
|  1 | Avtomatika                       |
|  2 | Elektronna Tehnika i Tehnologii  |
|  3 | Elektrotehnicheski               |
|  4 | Energo-mashinostroitelen         |
|  5 | Kompiutarni sistemi i upravlenie |
|  6 | Mashino-tehnologichen            |
+----+----------------------------------+
```

Както се вижда в този пример се използва външния ключ univ_id.

3. Име, фамилно име и телефон на преподавателите от факултет с id=4:

```
SELECT firstname, lastname, phone
```

```
FROM university.professors
WHERE faculty_id=4;

+-----------+----------+---------+
| firstname | lastname | phone   |
+-----------+----------+---------+
| Boncho    | Bonev    | 9652295 |
| Hristina  | Antonova | 9652359 |
| Ivailo    | Banov    | 9652209 |
| Emanuil   | Agoncev  | 9652436 |
| Hristo    | Petkov   | 9653629 |
+-----------+----------+---------+
```

4. Списък на всички данни за преподаватели с първо име “Ivan”:

```
SELECT *
FROM university.professors
WHERE firstname="Ivan";
```

5. Възможно е използването на логическите опиратори AND и OR. Следният пример показва всички преподаватели с първо име “Daniela” и фамилно име “Gotceva”:

```
SELECT *
FROM university.professors
WHERE firstname="Daniela" AND lastname="Gotceva";
```

Следният пример показва списък с телефоните на преподавателите и първо име “Ivan” или “Dimitar”:

```
SELECT *
FROM university.professors
WHERE firstname="Ivan" OR firstname="Dimitar";
```

6. Можете да използвате операторите за сравнение <, <=, =, >, >=, !=. Следният пример ще покаже телефоните на преподавателите с id>=20:

```
SELECT id, phone
FROM university.professors
WHERE id>=20;
```

7. Можете да правите проверка за неопределеност чрез операторът IS. Следният пример ще изведе имената и телефоните на преподавателите, чийто телефонни номера не са NULL:

```
SELECT firstname, lastname, phone
FROM university.professors
WHERE phone IS NOT NULL;
```

8. Както виждате подредбата на резултатът е неопределена. Чрез използване на допълнителна клауза “ORDER BY” ние можем да подреждаме данните. Ето придишния пример, като сме подредили резултатите по първото име на преподавателите в азбучен ред:

```
SELECT firstname, lastname, phone
```

```
FROM university.professors
WHERE phone IS NOT NULL
ORDER BY firstname;
```

Възможно е и подреждане “наобратно”:

```
SELECT firstname, lastname, phone
FROM university.professors
WHERE phone IS NOT NULL
ORDER BY firstname DESC;
```

Ако искате да подредите по няколко полета, то ги изредете едно след друго. Първо ще се сортира по първото поле, после по второто и т.н. Например ако искаме да извлечем имената на всички преподаватели по азбучен ред, подредени първо по малкото си име, а после по фамилно ще напишем:

```
SELECT firstname, lastname, phone
FROM university.professors
ORDER BY firstname, lastname;
```

Добавяйки DESC след някой от параметрите ще обърнете реда на подреждането.

9. Чрез операторът IN можете да проверявате за принадлежност към множество. Ето пример за списък с имената и id на преподавателите от факултети 1, 3 и 5, подредени по азбучен ред на първото си име:

```
SELECT id, firstname, lastname
FROM university.professors
WHERE faculty_id IN (1, 3, 5)
ORDER BY firstname;
```

10. Чрез операторът LIKE можете да правите “маски” за търсене. Следният пример ще изведе id, имената и телефона на всички преподаватели с първа буква на фамилното име “I”:

```
SELECT id, firstname, lastname, phone
FROM university.professors
WHERE lastname LIKE 'I%';
```

Виждате, че ключова роля играе символа ‘%’. Следващата заявка ще върне всички преподаватели, които имат буквата “P” във фамилното си име:

```
SELECT id, firstname, lastname, phone
FROM university.professors
WHERE lastname LIKE '%P%';
```

11. Накрая ако не желаете да виждате всички резултати, а само част от тях, то може да се използва операторът `LIMIT`. Следния пример ще покаже само първите 10 резултата от заявка за показване на имената на всички преподаватели:

```
SELECT firstname, lastname
FROM professors
```

```
LIMIT 10;
```

Това е изключително подходящо при програми, които четат големи обеми от информация, като например статистики. Най-често подреждаме информацията по колона от тип timestamp в обратен ред и правим LIMIT, за да видим само последните записи, които ни интересуват.

Многотаблични заявки SELECT

Вече знаем, че разделянето на една база от данни се прави с цел да се спести обем информация и да имаме колкото се може по-малко дублиране на такава. Това естествено е добре, но си има и цена – много често ни се налага да комбинираме информация едновременно от две или повече таблици. В тези случаи стандартният формат на еднотаблична заявка SELECT не върши работа.

В най-простия си вариант с две таблици, които сравняваме, те ще имат две колони, по които лесно ще бъдат сравнявани. От примера с базата от данни за университет, ако искаме да изкараме имената на преподавателите заедно с името на факултета, в който преподават, то ще се наложи да направим многотаблична заявка. Това е така, защото в таблицата professors няма колона, в която пазим името на факултета, а пазим само неговото id. Ето как се изпълнява въпросната многотаблична заявка:

```
use university;
SELECT professors.firstname, professors.lastname,
faculties.name
FROM professors, faculties
WHERE professors.faculty_id = faculties.id;
+-----------+------------+----------------------------------+
| firstname | lastname   | name                             |
+-----------+------------+----------------------------------+
| Todor     | Ionkov     | Avtomatika                       |
| Emil      | Nikolov    | Avtomatika                       |
| Plamen    | Tzvetkov   | Avtomatika                       |
| Emil      | Garipov    | Avtomatika                       |
| Valeri    | Mladenov   | Avtomatika                       |
| Marin     | Hristov    | Elektronna Tehnika i Tehnologii  |
| Dimitar   | Todorov    | Elektronna Tehnika i Tehnologii  |
| Stela     | Mileva     | Elektronna Tehnika i Tehnologii  |
| Emil      | Manolov    | Elektronna Tehnika i Tehnologii  |
| Philip    | Koparanov  | Elektronna Tehnika i Tehnologii  |
| Stefcho   | Guninski   | Elektrotehnicheski               |
| Liubomir  | Balgaranov | Elektrotehnicheski               |
| Nadejda   | Peeva      | Elektrotehnicheski               |
| Petar     | Nakov      | Elektrotehnicheski               |
| Snejana   | Evtimova   | Elektrotehnicheski               |
| Boncho    | Bonev      | Energo-mashinostroitelen         |
| Hristina  | Antonova   | Energo-mashinostroitelen         |
| Ivailo    | Banov      | Energo-mashinostroitelen         |
| Emanuil   | Agoncev    | Energo-mashinostroitelen         |
| Hristo    | Petkov     | Energo-mashinostroitelen         |
| Ognian    | Nakov      | Kompiutarni sistemi i upravlenie |
| Daniela   | Gotceva    | Kompiutarni sistemi i upravlenie |
| Valentin  | Kamburov   | Mashino-tehnologichen            |
```

```
| Anelia    | Ivanova    | Mashino-tehnologichen            |
| Georgi    | Popov      | Mashino-tehnologichen            |
| Daniela   | Peneva     | Mashino-tehnologichen            |
| Nikolai   | Nikolov    | Mashino-tehnologichen            |
| Bojidar   | Galucov    | Biologicheski                    |
| Mariela   | Ojdakova   | Biologicheski                    |
| Iana      | Topalova   | Biologicheski                    |
| Natasha   | Tzanova    | Biologicheski                    |
| Toni      | Spasov     | Himicheski                       |
| Ivan      | Petkov     | Himicheski                       |
| Ivan      | Soskov     | Matematika i informatika         |
| Ivan      | Gantchev   | Matematika i informatika         |
| Liudmil   | Vasilev    | Fizicheski                       |
| Ivan      | Lalov      | Fizicheski                       |
+-----------+------------+----------------------------------+
```

Въпреки, че в практиката най-често се използват външен и съответния му първичен ключ за сравнение, не е задължително колоната за връзка да е ключ. Съвсем възможно е да са дори съставни колони (повече от една). Указването на името на таблицата преди полето не е задължително, но е препоръчително. То може да се пропуска само при условие, че колоните в таблиците са с различни имена.

Така разгледаното съединение на таблици се нарича вътрешно (INNER JOIN). Дадения пример е от SQL1 стандартът и е лесно разбираем. В SQL2 стандарта този запис остава валиден, но се налага и алтернативен начин за осъществяване на същата заявка с използването на нова ключова дума JOIN в полето “FROM” на заявката. Горния пример е еквивалентен на:

```
   SELECT professors.firstname, professors.lastname,
faculties.name
   FROM professors INNER JOIN faculties
            ON professors.faculty_id = faculties.id;
```

Съществуват и три типа външно съединение на таблици. За съжаление от нашата примерна база от данни те не могат да се демонстрират, затова ще създадем нова:

```
   CREATE DATABASE joins;
   USE joins;

   CREATE TABLE books(
   `name` VARCHAR(255) NULL DEFAULT NULL,
   `author_id` INT NULL DEFAULT NULL);

   CREATE TABLE authors(
   `name` VARCHAR(255) NULL DEFAULT NULL,
   `id` INT NULL DEFAULT NULL);

   INSERT INTO books(name, author_id)
   VALUES   ('Da obichash nepoznat', 1),
          ('Plut i kruv', 2),
          ('Tainstvoto na Iuni', NULL);

   INSERT INTO authors(name, id)
   VALUES   ('Barbara Friiti', 1),
          ('Michael Kuningham', 2),
```

```
('Tuwe Janson', 3);
```

Нека видим какво се получава при вътрешно съединение на двете таблици. В примера ще покажем име на книга съединено с името на нейния автор:

```
SELECT books.name, authors.name
FROM books INNER JOIN authors ON books.author_id = authors.id;
+----------------------+------------------+
| name                 | name             |
+----------------------+------------------+
| Da obichash nepoznat | Barbara Friiti   |
| Plut i kruv          | Michael Kuningham |
+----------------------+------------------+
```

Ключовата дума “INNER” всъщност не е необходима. Ако я пропуснем MySQL ще разбере, че съединението е вътрешно. По същия начин стои положението и с външното съединение показано по-долу, при демонстрацията на което ще пропуснем ключовата дума “OUTER” в заявките.

Виждаме, че въведохме три книги в таблицата “books”, но в списъка излязоха само две. Това, което се получи е напълно нормално, защото на третата книга (Tainstvoto na Iuni) не е въведен author_id. По този начин тя е пропусната, защото не дава истина при сравнението в клаузата ON на съединението на таблиците.

За да изведем списък на всички книги и имената техните автори, а ако автор не е въведен да бъде изведен просто като NULL, използваме т.нар. LEFT OUTER JOIN (накратко LEFT JOIN) или “ляво външно съединение”:

```
SELECT books.name, authors.name
FROM books LEFT JOIN authors ON books.author_id =
authors.id;
+----------------------+------------------+
| name                 | name             |
+----------------------+------------------+
| Da obichash nepoznat | Barbara Friiti   |
| Plut i kruv          | Michael Kuningham |
| Tainstvoto na Iuni   | NULL             |
+----------------------+------------------+
```

Виждате, че резултатът е получен точно както очаквахме. Досещате се, че съществува и дясно съединение на таблици. То действа по абсолютно същия начин, но в случая “натежава” другата таблица:

```
SELECT books.name, authors.name
FROM books RIGHT JOIN authors ON books.author_id =
authors.id;
+----------------------+------------------+
| name                 | name             |
+----------------------+------------------+
| Da obichash nepoznat | Barbara Friiti   |
| Plut i kruv          | Michael Kuningham |
| NULL                 | Tuwe Janson      |
+----------------------+------------------+
```

В ANSI стандарта съществува и FULL JOIN, който обаче не се поддържа от MySQL (както и от почти всички системи за управление на бази от данни).

Възможно е да бъде направен чрез няколко заявки, като се използва временна таблица за съхранение на резултатите. Истината е, че FULL JOIN почти никога не се използва и затова няма да разглеждаме такъв пример.

Ако условие ON в съединението липсва (винаги е true) или то не отразява никаква връзка между съединяваните таблици, то получаваме т.нар. "декартово произведение" на таблиците. Това означава, че всеки елемент от първата таблица ще бъде долепен със всеки от втората. Вижте подробно резултата, като от горния пример дадете винаги истинно условие след ON – например 1=1.

Вложен SELECT

Когато сме разделили базата от данни на множество класове обекти, много често се налага да "прескачаме" през един или повече обекти. Това се получава, когато между два или повече класа обекти няма пряка връзка с външен ключ или друга колона, по която да бъдат сравнени.

От примера с базата от данни с университет това може да се получи, ако например поискаме да изкараме имената на всички преподаватели от университет с id=1. Ще видите, че в таблицата professors съществува връзка между преподавател и факултет, но няма пряка връзка до таблицата university. Затова ще бъде нужно да използваме веднъж връзката до факултета, след което връзката на факултета до университета:

```
SELECT professors.firstname, professors.lastname
FROM professors
WHERE professors.faculty_id IN (
  SELECT faculties.id
  FROM faculties
  WHERE faculties.univ_id = 1
);
```

Важно е да отбележим, че тук използвахме операторът IN, който означава "принадлежност към множество". Вложеният SELECT може да връща повече от един ред в резултата си и затова не е уместно да се използва сравнение с равенство "=". Действително – университет с id=1 може да има повече от един факултет, т.е. от вложената заявка ще се върне множество от id-та на факултети!

Ето как бихме могли да използваме същото в комбинация с JOIN, като добавим и името на университета в резултата от заявката:

```
SELECT professors.firstname, professors.lastname,
university.name
FROM professors JOIN university
     ON professors.faculty_id IN (
  SELECT faculties.id
  FROM faculties
  WHERE faculties.univ_id = university.id
     )
```

```
   WHERE university.id = 1;
```

Условието в частта ON на FROM ще наричаме “свързващо” за двете таблици. Условието, което се намира в клаузата WHERE на основния SELECT ще наричаме “ограничаващо“. Практически ние не сме ограничени например да ги разменим или дори обединим с AND – заявката ще работи по същия начин и ще върне същия резултат. Реално между двете условия просто е приложено логическо “И”. Добра практика е все пак да се спазва точно разделението и в “ON” да се съдържат само свързващите условия, а всички останали да бъдат в “WHERE”.

Що се отнася до свързващите условия – там можем да спестим вложения SELECT по следния начин:

```
   SELECT professors.firstname, professors.lastname,
university.name
   FROM professors JOIN faculties ON
professors.faculty_id=faculties.id
                   JOIN university ON faculties.univ_id =
university.id
   WHERE university.id = 1;
```

Първо към таблицата university се присъединява faculties, а после към резултатната се присъединява и professors. Така “паразитно” добавихме във FROM таблица faculties, която не участва в нито една колона от изхода. Записът е по-кратък и лесно четим, но реално този подход прави заявките по-бавни. Причината е, че при вложен select не се прави пълно свързване на таблиците, а само на подмножество от тях. Затова когато е възможен ще го предпочитаме. Най-лесното правило за писане на правилни заявки е следното: правете JOIN на само на таблиците споменати в условието на задачата.

Ето още един пример: да се изведат имената на преподавателите от Технически университет и името на факултета, в който работят:

```
   SELECT professors.firstname, professors.lastname,
faculties.name
   FROM professors JOIN faculties ON professors.faculty_id =
faculties.id
   WHERE faculties.univ_id = (
           SELECT id FROM university
           WHERE university.name = 'Technical University'
         );
```

Ясно се вижда, че в “свързващото условие” се използват външните и първичните ключове на таблиците. В “ограничаващото условие” пък добавяме допълнителни параметри, които намаляват обема на връщания резултат.

Защо не използвахме LEFT JOIN, вместо INNER ? Така бихме извели в списъка и преподавателите, които нямат запис в кой факултет работят. За нашата примерна база от данни обаче това не е възможно, защото и двете сравнявани полета (professors.faculty_id = faculties.id) са дефинирани като NOT NULL (faculties.id е PRIMARY KEY) при създаването на таблиците. Това означава, че няма да има разлика в резултата при различните видове JOIN, защото не сме допуснали възможност да съществува преподавател, който не принадлежи на някой

факултет. На тези моменти трябва да се обръща специално внимание по време на етапа на проектиране на базата от данни.

В много случаи свързващото условие може да се измести към ограничаващото. Това се получава тогава, когато искаме да изведем информация само от едната таблица, а другата се използва само като ограничение. Например нека извадим списък от имената на преподавателите от университет с име “Technical University” (не знаем неговото id):

```
SELECT professors.firstname, professors.lastname
FROM professors
WHERE professors.faculty_id IN (
  SELECT faculties.id
  FROM faculties
  WHERE faculties.univ_id = (
       SELECT id
       FROM university
       WHERE university.name = ‘Technical University’
  )
);
```

Дали ще напишем тази заявка с JOIN между таблиците professors и university или ще я напишем по горния начин практически няма значение – резултатът ще бъде еквивалентен. Колкото повече връзки има по “пътя” на заявката, толкова повече SELECT заявки трябва да вложим. Освен това никой не ни ограничава да използваме вложен SELECT в условията ON на JOIN (вижте втория пример по-горе).

Ето и още една задача: да се изведе списък на всички предмети от ‘Technical University’ като до тях се долепи името на факултета им:

```
SELECT faculties.name, subjects.name
FROM subjects JOIN faculties
              ON subjects.lead_professor_id IN(
                 SELECT professors.id
                 FROM professors
                 WHERE professors.faculty_id=faculties.id
              )
WHERE faculties.univ_id IN(
  SELECT university.id
  FROM university
  WHERE university.name = 'Technical University'
      );
```

Агрегатни функции

Агрегатните функции ни позволяват да обединим (групираме) дадено множество и да направим някакво обобщение за него. За да ги илюстрираме по-нагледно, нека създадем база данни с примерни записи, върху която да работи.

```
CREATE DATABASE `banks`;

CREATE TABLE `banks`.`banks` (
   `code` INT NOT NULL AUTO_INCREMENT ,
   `name` VARCHAR( 255 ) NOT NULL ,
   `country` VARCHAR( 255 ) NOT NULL ,
   PRIMARY KEY ( `code` )
) ENGINE = InnoDB;

CREATE TABLE `banks`.`branches` (
   `id` TINYINT NOT NULL AUTO_INCREMENT ,
   `name` VARCHAR( 255 ) NOT NULL ,
   `address` VARCHAR( 255 ) NOT NULL ,
   `bank_code` INT NOT NULL ,
   PRIMARY KEY ( `id` ) ,
   FOREIGN KEY ( `bank_code` )
         REFERENCES `banks`.`banks`( `code` )
         ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE = InnoDB;

CREATE TABLE `banks`.`employees` (
   `id` INT NOT NULL AUTO_INCREMENT ,
   `name`  VARCHAR( 255 ) NOT NULL ,
   `branch_id` TINYINT NOT NULL,
   PRIMARY KEY ( `id` ),
   FOREIGN KEY ( `branch_id` )
         REFERENCES `banks`.`branches`( `id` )
         ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE = InnoDB;

CREATE TABLE `banks`.`customers` (
   `id` INT NOT NULL AUTO_INCREMENT ,
   `name` VARCHAR( 255 ) NOT NULL ,
   `address` VARCHAR( 255 ) NULL DEFAULT NULL ,
   `bank_mgr` INT NULL ,
   PRIMARY KEY ( `id` ) ,
   FOREIGN KEY ( `bank_mgr` )
         REFERENCES `banks`.`employees`( `id` )
) ENGINE = InnoDB;

CREATE TABLE `banks`.`accounts` (
   `id` INT NOT NULL AUTO_INCREMENT ,
   `amount` DECIMAL( 9,2 ) NOT NULL ,
   `type` TINYINT NOT NULL,
   `branch_id` TINYINT NOT NULL,
   `customer_id` INT NOT NULL,
   PRIMARY KEY ( `id` ),
   FOREIGN KEY ( `branch_id` )
         REFERENCES `banks`.`branches`( `id` )
         ON DELETE CASCADE ON UPDATE CASCADE,
   FOREIGN KEY ( `customer_id` )
         REFERENCES `banks`.`customers`( `id` )
         ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE = InnoDB;

USE banks;

INSERT INTO `banks` (`code` ,`name` ,`country`)
VALUES  ( NULL , 'Bulbank', 'Bulgaria' ),
   ( NULL , 'Wells Fargo', 'USA' ),
   ( NULL , 'Bank of America', 'USA' ),
   ( NULL , 'Societe General', 'France' );

INSERT INTO `branches` (`id` ,`name` ,`address`, `bank_code`)
VALUES  ( NULL , 'Serdica', 'Sofia centar', 1 ),
```

```
    ( NULL , 'Slatina', 'Sofia Geo Milev', 1 ),
    ( NULL , 'Manhattan', 'Manhattan, New York', 2 ),
    ( NULL , 'LA', 'Los Angeles', 3 ),
    ( NULL , 'Paris', 'Paris', 4 ),
    ( NULL , 'Marseilles', 'Marseilles', 4 );

INSERT INTO `employees` (`id` ,`name` , `branch_id`)
VALUES  ( NULL , 'Ivan Ivanov', 1 ),
    ( NULL , 'Ivan Stoianov', 1 ),
    ( NULL , 'Mihail Zahariev', 1 ),
    ( NULL , 'Milen Stoilov', 2 ),
    ( NULL , 'Svilen Petrov', 2 ),
    ( NULL , 'Ilian Stoianov', 2 ),
    ( NULL , 'Petar Petrov', 2 ),
    ( NULL , 'Jimmy Carter', 3 ),
    ( NULL , 'John Smith', 3 ),
    ( NULL , 'Mary Jane', 3 ),
    ( NULL , 'James Pitt', 4 ),
    ( NULL , 'Francoa Dupres', 5 ),
    ( NULL , 'Alfonso Levi', 6 );

INSERT INTO `customers` (`id` ,`name` , `bank_mgr`)
VALUES  ( NULL , 'Todor Ivanov', 1 ),
    ( NULL , 'Petko Stoianov', 1 ),
    ( NULL , 'Neno Nenov', 2 ),
    ( NULL , 'Mariana Zaharieva', 3 ),
    ( NULL , 'Elica Zaharieva', 3 ),
    ( NULL , 'Atanas Petrov', 4 ),
    ( NULL , 'Ivan Ivanov', 4 ),
    ( NULL , 'Zlatomir Petrov', 4 ),
    ( NULL , 'Mihail Ivchev', 5 ),
    ( NULL , 'Todor Shtilianov', 6 ),
    ( NULL , 'Ivailo Ivanov', 7 ),
    ( NULL , 'George Lucas', 8 ),
    ( NULL , 'George Harison', 8 ),
    ( NULL , 'Michael Jackson', 8 ),
    ( NULL , 'Tony Martin', 8 ),
    ( NULL , 'Tony McCarter', 10 ),
    ( NULL , 'Alexander Smith', 11 ),
    ( NULL , 'Maria Smith', 11 ),
    ( NULL , 'Alain Delrick', 12 ),
    ( NULL , 'Devry Henry', 12 ),
    ( NULL , 'Lenard Renne', 12 ),
    ( NULL , 'Fontaine Rupert', 13 );

INSERT INTO `accounts` (`id` ,`amount` , `type` , `customer_id`,
`branch_id`)
VALUES  ( NULL , 156.38, 2, 1, 1 ),
    ( NULL , 136.22, 1, 2, 1 ),
    ( NULL , 42.98, 1, 3, 1 ),
    ( NULL , 1236.33, 1, 4, 1 ),
    ( NULL , 211.98, 2, 5, 1 ),
    ( NULL , 1200.00, 2, 6, 2 ),
    ( NULL , 133.48, 1, 7, 2 ),
    ( NULL , 256.41, 2, 8, 2 ),
    ( NULL , 1331.50, 2, 9, 2 ),
    ( NULL , 116.88, 2, 10, 2 ),
    ( NULL , 200.91, 1, 10, 2 ),
    ( NULL , 99.18, 1, 11, 2 ),
    ( NULL , 6712.52, 1, 12, 3 ),
    ( NULL , 12000.56, 1, 12, 3 ),
    ( NULL , 322.99, 2, 12, 3 ),
    ( NULL , 991.63, 1, 13, 3 ),
    ( NULL , 559.32, 2, 14, 3 ),
    ( NULL , 680.13, 1, 15, 3 ),
```

```
  ( NULL , 532.57, 1, 15, 3 ),
  ( NULL , 402.26, 1, 16, 3 ),
  ( NULL , 1536.91, 2, 17, 4 ),
  ( NULL , 14921.43, 1, 18, 4 ),
  ( NULL , 3910.50, 1, 19, 5 ),
  ( NULL , 231.37, 1, 20, 5 ),
  ( NULL , 7236.60, 1, 21, 5 ),
  ( NULL , 2226.63, 2, 21, 5 ),
  ( NULL , 500.00, 2, 22, 6 );
```

Ще дадем няколко прости примера:

1. Изкарайте броят на въведените банки в базата от данни:

```
SELECT COUNT(*) FROM banks;
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

Функцията COUNT() в случая преброи броят редове на таблицата banks.

2. Изведете най-голямата сума на акаунт в базата от данни:

```
SELECT MAX(amount) FROM accounts;
+-------------+
| MAX(amount) |
+-------------+
|    14921.43 |
+-------------+
```

Функцията MAX() намери най-голямата стойност в колоната и я изведе на екрана.

3. Изведете най-малката сума на акаунт в базата от данни от банков клон с id=2:

```
SELECT MIN(amount)
FROM accounts
WHERE branch_id=2;
+-------------+
| MIN(amount) |
+-------------+
|       99.18 |
+-------------+
```

Тук очевидно MIN намери най-малката стойност в колоната “amount” на тези редове от таблица “accounts”, за които branch_id е равно на 2.

4. Намерете средното аритметично на сумите на всички акаунти на Bulbank:

```
mysql> SELECT AVG(amount)
FROM accounts
WHERE branch_id IN(
  SELECT id
  FROM branches
  WHERE bank_code = (
```

```
        SELECT code
        FROM banks
        WHERE name = 'Bulbank'
  )
);
+-------------+
| AVG(amount) |
+-------------+
|  426.854167 |
+-------------+
```

Ясно е, че AVG() сумира стойностите в колоната по редове и разделя сумата на техния брой.

5. Изведете името и общата сума пари, която притежава клиент с id=12, във всички негови акаунти:

```
SELECT customers.name, SUM(accounts.amount)
FROM customers JOIN accounts ON customers.id =
accounts.customer_id
WHERE customers.id = 12;
+--------------+----------------------+
| name         | SUM(accounts.amount) |
+--------------+----------------------+
| George Lucas |             19036.07 |
+--------------+----------------------+
```

Функцията SUM() връща сумата на върнатите редове.

Групиране на данни

Когато имаме повтарящи се данни в дадена колона, то можем да ги "групираме". За целта се използва клаузата "GROUP BY". Този оператор обикновено се използва в комбинация с агрегатни функции. Целта е агрегатната функция да се приложи върху множество по-малки резултатни таблици, резултатите от които накрая да се обединят в една.

1. Да се изведе сумата на акаунти от тип 1 и сумата на акаунти от тип 2:

```
SELECT type, SUM(amount)
FROM accounts
GROUP BY type;
+------+-------------+
| type | SUM(amount) |
+------+-------------+
|    1 |    49468.67 |
|    2 |     8419.00 |
+------+-------------+
```

2. Изкарайте името на клиентите и сумата от техните сметки в системата:

```
SELECT customers.name, SUM(accounts.amount)
```

```
   FROM customers LEFT JOIN accounts ON customers.id =
accounts.customer_id
   GROUP BY customers.name;
   +------------------+----------------------+
   | name             | SUM(accounts.amount) |
   +------------------+----------------------+
   | Alain Delrick    |              3910.50 |
   | Alexander Smith  |              1536.91 |
   | Atanas Petrov    |              1200.00 |
   | Devry Henry      |               231.37 |
   | Elica Zaharieva  |               211.98 |
   | Fontaine Rupert  |               500.00 |
   | George Harison   |               991.63 |
   | George Lucas     |             19036.07 |
   | Ivailo Ivanov    |                99.18 |
   | Ivan Ivanov      |               133.48 |
   | Lenard Renne     |              9463.23 |
   | Maria Smith      |             14921.43 |
   | Mariana Zaharieva |              1236.33 |
   | Michael Jackson  |               559.32 |
   | Mihail Ivchev    |              1331.50 |
   | Neno Nenov       |                42.98 |
   | Petko Stoianov   |               136.22 |
   | Todor Ivanov     |               156.38 |
   | Todor Shtilianov |               317.79 |
   | Tony Martin      |              1212.70 |
   | Tony McCarter    |               402.26 |
   | Zlatomir Petrov  |               256.41 |
   +------------------+----------------------+
```

В последния пример групирането е по име. Трябва да отбележим, че това по принцип е грешно, защото е възможно да има двама клиенти с едно и също име. Затова е по-добре да групираме по уникалния ключ:

```
   SELECT customers.name, SUM(accounts.amount)
   FROM customers LEFT JOIN accounts ON customers.id =
accounts.customer_id
   GROUP BY customers.id
   ORDER BY customers.name;
```

За да видите разликата, въведете още един клиент с име “Ivan Ivanov” и направете горните заявки още веднъж. Тук трябва да забележим, че ANSI стандарта изисква задължително в условието GROUP BY да присъства някоя от колоните изредени след SELECT. В MySQL това не е задължително.

3. Изведете имената на банките и средната сума на акаунтите в тях:

```
   SELECT banks.name, AVG(accounts.amount)
   FROM banks JOIN accounts
              ON accounts.branch_id IN(
                 SELECT branches.id
                 FROM branches
                 WHERE branches.bank_code = banks.code
              )
   GROUP BY banks.name;
   +----------------+-------------+
   | name           | AVG(amount) |
   +----------------+-------------+
   | Bank of America | 8229.170000 |
```

```
| Bulbank         |  426.854167 |
| Societe General | 2821.020000 |
| Wells Fargo     | 2775.247500 |
+-----------------+-------------+
```

4. За да демонстрираме по-сложни заявки, нека усложним предишния пример малко – изведете името на банката и средната сума на акаунтите в нея, но само за банките, които имат буквата "M" някъде в името на банковия клон:

```
SELECT banks.name, AVG(accounts.amount)
FROM banks JOIN branches ON branches.bank_code = banks.code
           JOIN accounts ON accounts.branch_id = branches.id
WHERE branches.name LIKE '%M%'
GROUP BY banks.name;
+-----------------+-------------+
| name            | AVG(amount) |
+-----------------+-------------+
| Societe General | 2821.020000 |
| Wells Fargo     | 2775.247500 |
+-----------------+-------------+
```

Тук добавихме и таблицата branches в JOIN между таблиците, защото в WHERE използвахме информация от нея.

5. Изведете името на банката, клона, името на служителя и общата сума в акаунтите, на чийто клиенти той е мениджър. Полученият резултат да се сортира в реда име на банка, име на клон и име на служител:

```
SELECT banks.name, branches.name, employees.name,
SUM(accounts.amount)
FROM banks JOIN branches ON banks.code = branches.bank_code
JOIN employees ON branches.id = employees.branch_id
JOIN accounts ON accounts.customer_id IN(
                 SELECT customers.id
                 FROM customers
                 WHERE customers.bank_mgr = employees.id
               )
GROUP BY employees.id
ORDER BY banks.name, branches.name, employees.name;
+-----------------+------------+-----------------+-------------+
| name            | name       | name            | SUM(amount) |
+-----------------+------------+-----------------+-------------+
| Bank of America | LA         | James Pitt      |    16458.34 |
| Bulbank         | Serdica    | Ivan Ivanov     |      292.60 |
| Bulbank         | Serdica    | Ivan Stoianov   |       42.98 |
| Bulbank         | Serdica    | Mihail Zahariev |     1448.31 |
| Bulbank         | Slatina    | Ilian Stoianov  |      317.79 |
| Bulbank         | Slatina    | Milen Stoilov   |     1589.89 |
| Bulbank         | Slatina    | Petar Petrov    |       99.18 |
| Bulbank         | Slatina    | Svilen Petrov   |     1331.50 |
| Societe General | Marseilles | Alfonso Levi    |      500.00 |
| Societe General | Paris      | Francoa Dupres  |    13605.10 |
| Wells Fargo     | Manhattan  | Jimmy Carter    |    21799.72 |
| Wells Fargo     | Manhattan  | Mary Jane       |      402.26 |
+-----------------+------------+-----------------+-------------+
```

Както виждате GROUP BY, в комбинация с агрегатни функции, ни дава добри възможности за обобщаване на данни и извършване на прости статистически изчисления.

6. Изведете името на банката, клона, името на служителят и общата сума в акаунтите, на чийто клиенти той е мениджър. Получения резултат да се сортира в реда име на банка, име на клон и име на служител, но само за служителите, чието име започва с “M”:

```
   SELECT banks.name, branches.name, employees.name,
SUM(accounts.amount)
   FROM banks JOIN branches ON banks.code = branches.bank_code
     JOIN employees ON branches.id = employees.branch_id
           JOIN accounts ON accounts.customer_id IN(
                            SELECT customers.id
                            FROM customers
                            WHERE customers.bank_mgr =
employees.id
                           )
   WHERE employees.name LIKE 'M%'
   GROUP BY employees.id
   ORDER BY banks.name, branches.name, employees.name;
   +-------------+-----------+-----------------+-------------+
   | name        | name      | name            | SUM(amount) |
   +-------------+-----------+-----------------+-------------+
   | Bulbank     | Serdica   | Mihail Zahariev |     1448.31 |
   | Bulbank     | Slatina   | Milen Stoilov   |     1589.89 |
   | Wells Fargo | Manhattan | Mary Jane       |      402.26 |
   +-------------+-----------+-----------------+-------------+
```

Клауза HAVING

Понякога се налага да филтрираме данните след като вече сме направили дадена калкулация. Фразата HAVING се използва за прилагане на условия върху групи (обикновено оформени чрез GROUP BY).

Ще демонстрираме това с пример. Вече знаем как можем да изкараме списък на клиентите и средните суми на техните сметки:

```
   SELECT customers.name, AVG(accounts.amount)
   FROM customers JOIN accounts ON customers.id =
accounts.customer_id
   GROUP BY customers.name;
```

Нека обаче да направим следното – искаме да изкараме списъка само на клиентите, чийто сметки имат средна сума по-голяма от 500. Това не би могло да стане в клауза WHERE, защото при нея все още не е пресметнато AVG(accounts.amount). Затова можем да използваме допълнително филтриране чрез клауза HAVING:

```
   SELECT customers.name, AVG(accounts.amount)
```

```
   FROM customers JOIN accounts ON customers.id =
accounts.customer_id
   GROUP BY customers.name
   HAVING AVG(accounts.amount) > 500;
```

Ето как бихме могли да използваме WHERE и HAVING в комбинация – следната заявка изкарва списък на клиентите, чийто сметки имат средна сума на акаунта по-голяма от 500 и името им започва с "M":

```
   SELECT customers.name, AVG(accounts.amount)
   FROM customers JOIN accounts ON customers.id =
accounts.customer_id
   WHERE customers.name LIKE 'M%'
   GROUP BY customers.name
   HAVING AVG(accounts.amount) > 500;
```

Избягвайте да използвате HAVING, когато можете. В повечето прехвърляне на клаузата WHERE в HAVING ще даде идентичен резултат, но това ще е с цената на по-бавна заявка, защото много повече информация ще бъде обработвана (няма пропускане на редове още преди изпълнение на агрегатната функция). Също така индексите не работят с клаузи HAVING.

В общи линии винаги прилагайте теоремата: "ако в условието HAVING не участват агрегатни функции, то може да бъде пренесено в WHERE".

Псевдоними на колони и таблици

Псевдонимите на колони ни улесняват да пишем по-ясни записи на заявките.

Задача: Изведете името на банката и средната сума на акаунтите в нея, но само за банките, които имат буквата "M" някъде в името на банковия клон:

```
   SELECT banks.name, AVG(accounts.amount)
   FROM banks JOIN branches ON branches.bank_code = banks.code
              JOIN accounts ON accounts.branch_id = branches.id
   WHERE branches.name LIKE '%M%'
   GROUP BY banks.name;
   +-----------------+-------------+
   | name            | AVG(amount) |
   +-----------------+-------------+
   | Societe General | 2821.020000 |
   | Wells Fargo     | 2775.247500 |
   +-----------------+-------------+
```

Виждате, че резултатът е таблица с две колони. Имената им са съответно "name" и "AVG(amount)". Въпреки, че от гледна точка на програмиста това е напълно достатъчно, то не е лошо да ги именуваме с по-описателни имена. Например:

```
   SELECT banks.name AS bank_name, AVG(amount) AS
average_amount
```

```
FROM banks JOIN branches ON branches.bank_code = banks.code
           JOIN accounts ON accounts.branch_id = branches.id
WHERE branches.name LIKE '%M%'
GROUP BY banks.name;
```

```
+----------------+---------------+
| bank_name      | average_amount |
+----------------+---------------+
| Societe General | 2821.020000    |
| Wells Fargo     | 2775.247500    |
+----------------+---------------+
```

Още повече – ако след where сме използвали в заявката пълното име, то можем да го заменим с неговия псевдоним. В случая това е само на едно място при сортирането:

```
SELECT banks.name AS bank_name, AVG(amount) AS
average_amount
FROM banks JOIN branches ON branches.bank_code = banks.code
           JOIN accounts ON accounts.branch_id = branches.id
WHERE branches.name LIKE '%M%'
GROUP BY bank_name;
+----------------+---------------+
| bank_name      | average_amount |
+----------------+---------------+
| Societe General |    2821.020000 |
| Wells Fargo     |    2775.247500 |
+----------------+---------------+
```

Можем да правим и псевдоними вътре в заявката. Например нека разгледаме заявка, която изкарва името на клиента и сумата пари, които притежава, но само за клиентите с id> 3 и id < 10:

```
SELECT customers.name, SUM(accounts.amount)
FROM customers JOIN accounts ON customers.id =
accounts.customer_id
WHERE customers.id > 3 AND customers.id < 10
GROUP BY customers.id
ORDER BY customers.name;
+------------------+----------------------+
| name             | SUM(accounts.amount) |
+------------------+----------------------+
| Atanas Petrov    |              1200.00 |
| Elica Zaharieva  |               211.98 |
| Ivan Ivanov      |               133.48 |
| Mariana Zaharieva |              1236.33 |
| Mihail Ivchev    |              1331.50 |
| Zlatomir Petrov  |               256.41 |
+------------------+----------------------+
```

Можем да “прекръстим” таблицата customers с по-кратко име, като и зададем псевдоним в FROM:

```
SELECT cst.name, SUM(accounts.amount)
FROM customers AS cst JOIN accounts ON cst.id =
accounts.customer_id
```

```
WHERE cst.id > 3 AND cst.id < 10
GROUP BY cst.id
ORDER BY cst.name;
+------------------+----------------------+
| name             | SUM(accounts.amount) |
+------------------+----------------------+
| Atanas Petrov    |              1200.00 |
| Elica Zaharieva  |               211.98 |
| Ivan Ivanov      |               133.48 |
| Mariana Zaharieva|              1236.33 |
| Mihail Ivchev    |              1331.50 |
| Zlatomir Petrov  |               256.41 |
+------------------+----------------------+
```

Така навсякъде, където сме се обръщали към таблицата customers вече задължително трябва да използваме псевдонима. По принцип преименуването на таблици, които си имат вече статични имена не се счита за добър начин на работа. Наистина често ни спестява писане (ако използваме по-кратко име), но пък внася допълнително объркване, защото трябва да следим допълнителни данни.

Когато работим с таблици, които са извадки от други, то вече задължително трябва да им избираме псевдоним. Ще демонстрираме това с елементарна заявка, която изкарва броя на банките, които започват с буквата "B":

```
SELECT COUNT(*)
FROM ( SELECT code, name FROM banks ) AS orgs
WHERE orgs.name LIKE 'B%';
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

Заявки Update

Заявките от тип UPDATE се използват за обновяване на данни. Базовият синтаксис е:

```
UPDATE <име на таблица>
SET <правило за обновяване>
WHERE <условие>;
```

От примера с базата данни "banks" можем да направим следните задачи:

1. Добавя по 2% лихва на всички акаунти от тип 1:

```
UPDATE accounts
SET amount = amount + amount*2/100
WHERE type = 1;
```

2. Дава по 20 лева бонус на клиентите с име 'Ivan Ivanov':

```
UPDATE accounts
SET amount = amount + 20
WHERE customer_id IN(
   SELECT id
   FROM customers
   WHERE name = 'Ivan Ivanov'
);
```

3. Дава по 10 лева бонус на всички клиенти, чийто мениджър на акаунт е ‘John Smith’:

```
UPDATE accounts
SET amount = amount + 10
WHERE customer_id IN(
   SELECT id
   FROM customers
   WHERE bank_mgr = (
      SELECT id
      FROM employees
      WHERE name = 'John Smith'
   )
);
```

4. Променя адреса на клона на банка Societe General в град Paris:

```
UPDATE branches
SET address = '17 cours Valmy'
WHERE name = 'Paris' AND bank_code = (
   SELECT code
   FROM banks
   WHERE name = 'Societe General'
);
```

5. Можем да обновяваме информация и от две таблици. Следната заявка ще даде два лева на клиент с id 2, като същевременно ще въведе и неговия адрес:

```
UPDATE customers JOIN accounts
   ON customers.id = accounts.customer_id
SET accounts.amount = accounts.amount + 2, address =
'Opalchenska 12'
WHERE customers.id = 2;
```

Ясно е, че при по-свободни таблици имаме и гъвкави възможности чрез използването на LEFT и RIGHT JOIN.

Заявки Delete

Подобно на INSERT, заявките от тип DELETE са с изключително прост синтаксис:

```
DELETE FROM <таблица>
WHERE <условие>;
```

Нека демонстрираме един пример с базата от данни “banks”. Нека видим първо списък на акаунтите:

```
SELECT * FROM accounts;
```

id	amount	type	branch_id	customer_id
1	156.38	2	1	1
2	136.22	1	1	2
3	42.98	1	1	3
4	1236.33	1	1	4
5	211.98	2	1	5
6	1200.00	2	2	6
7	133.48	1	2	7
8	256.41	2	2	8
9	1331.50	2	2	9
10	116.88	2	2	10
11	200.91	1	2	10
12	99.18	1	2	11
13	6712.52	1	3	12
14	12000.56	1	3	12
15	322.99	2	3	12
16	991.63	1	3	13
17	559.32	2	3	14
18	680.13	1	3	15
19	532.57	1	3	15
20	402.26	1	3	16
21	1536.91	2	4	17
22	14921.43	1	4	18
23	3910.50	1	5	19
24	231.37	1	5	20
25	7236.60	1	5	21
26	2226.63	2	5	21
27	500.00	2	6	22

Нека изтрием акаунт с id=26:

```
DELETE FROM accounts
WHERE id = 26;
```

Ако изпълните заявката SELECT отново ще видите, че въпросният ред е изчезнал.

Важно е да се знае, че при използването на FOREIGN KEYS и ON DELETE CASCADE ще бъдат изтрити и записите на редове в таблици, които “сочат” към записа, който ще бъде изтрит. Например нека видим таблицата “customers”:

```
SELECT * FROM customers;
```

id	name	address	bank_mgr
1	Todor Ivanov	NULL	1
2	Petko Stoianov	NULL	1
3	Neno Nenov	NULL	2
4	Mariana Zaharieva	NULL	3
5	Elica Zaharieva	NULL	3

```
|  6 | Atanas Petrov     | NULL    |        4 |
|  7 | Ivan Ivanov       | NULL    |        4 |
|  8 | Zlatomir Petrov   | NULL    |        4 |
|  9 | Mihail Ivchev     | NULL    |        5 |
| 10 | Todor Shtilianov  | NULL    |        6 |
| 11 | Ivailo Ivanov     | NULL    |        7 |
| 12 | George Lucas      | NULL    |        8 |
| 13 | George Harison    | NULL    |        8 |
| 14 | Michael Jackson   | NULL    |        8 |
| 15 | Tony Martin       | NULL    |        8 |
| 16 | Tony McCarter     | NULL    |       10 |
| 17 | Alexander Smith   | NULL    |       11 |
| 18 | Maria Smith       | NULL    |       11 |
| 19 | Alain Delrick     | NULL    |       12 |
| 20 | Devry Henry       | NULL    |       12 |
| 21 | Lenard Renne      | NULL    |       12 |
| 22 | Fontaine Rupert   | NULL    |       13 |
+----+-------------------+---------+----------+
```

Нека изтрием клиент с id 10:

```
DELETE FROM customers
WHERE id = 10;
```

Ще видите, че в редовете в таблица “accounts”, чието поле “customer_id” е било равно на 10, също са изтрити:

```
SELECT * FROM accounts;
+----+----------+------+-----------+-------------+
| id | amount   | type | branch_id | customer_id |
+----+----------+------+-----------+-------------+
|  1 |   156.38 |    2 |         1 |           1 |
|  2 |   136.22 |    1 |         1 |           2 |
|  3 |    42.98 |    1 |         1 |           3 |
|  4 |  1236.33 |    1 |         1 |           4 |
|  5 |   211.98 |    2 |         1 |           5 |
|  6 |  1200.00 |    2 |         2 |           6 |
|  7 |   133.48 |    1 |         2 |           7 |
|  8 |   256.41 |    2 |         2 |           8 |
|  9 |  1331.50 |    2 |         2 |           9 |
| 12 |    99.18 |    1 |         2 |          11 |
| 13 |  6712.52 |    1 |         3 |          12 |
| 14 | 12000.56 |    1 |         3 |          12 |
| 15 |   322.99 |    2 |         3 |          12 |
| 16 |   991.63 |    1 |         3 |          13 |
| 17 |   559.32 |    2 |         3 |          14 |
| 18 |   680.13 |    1 |         3 |          15 |
| 19 |   532.57 |    1 |         3 |          15 |
| 20 |   402.26 |    1 |         3 |          16 |
| 21 |  1536.91 |    2 |         4 |          17 |
| 22 | 14921.43 |    1 |         4 |          18 |
| 23 |  3910.50 |    1 |         5 |          19 |
| 24 |   231.37 |    1 |         5 |          20 |
| 25 |  7236.60 |    1 |         5 |          21 |
| 27 |   500.00 |    2 |         6 |          22 |
+----+----------+------+-----------+-------------+
```

Виждате, че акаунти с id 10 и 11 са изтрити.

Това може да доведе до някои “неудобства”. Нека например се опитаме да изтрием служител с id = 2:

```
DELETE FROM employees
WHERE id = 2;
ERROR 1451 (23000): Cannot delete or update a parent row: a
foreign key constraint fails (`banks`.`customers`, CONSTRAINT
`customers_ibfk_1` FOREIGN KEY (`bank_mgr`) REFERENCES
`employees` (`id`))
```

Проблемът тук е, че в таблицата "customers" има FOREIGN KEY "bank_mgr", който сочи към таблицата "employees", но НЯМА `ON DELETE CASCADE`. Това всъщност е съвсем нормално, защото ако уволним даден служител не би следвало да прекратяваме договорите с клиентите на банката, които той обслужва. Как тогава все пак да изтрием служител с id = 2?

Отговорът е, че трябва да направим NULL или да променим към друг bank_mgr всички клиенти, които сочат към bank_mgr=2:

```
UPDATE customers
SET bank_mgr = NULL
WHERE bank_mgr = 2;
```

Вече можем да изтрием служител с id = 2:

```
DELETE FROM employees
WHERE id = 2;
```

Въпреки, че първоначалното впечатление е, че FOREIGN KEYS ни "пречат" с тази си особеност, това всъщност ни помага значително, защото така се грижим да правилен интегритет на базата от данни. Не би трябвало в нашата база от данни да има неверни данни, нали?

Тук отново трябва да се обърне изключително внимание на дизайна на базата от данни. Хубаво е в ER диаграмата да си отбелязвате изрично коя връзка има `ON DELETE CASCADE` и коя не. Винаги преди изпълнение на DELETE трябва да прецените засегнатите връзки и ако има такава без `ON DELETE CASCADE`, то трябва първо да изпълните заявка от тип UPDATE.

В заключение ще кажем, че е възможно да си спестим този труд, като направим външния ключ с опция "`ON DELETE SET NULL`". Това всъщност ще свърши абсолютно същата работа, която демонстрирахме по-горе със заявката UPDATE. Препоръчително е да се възползвате от тези удобства.

Виртуални таблици (view)

Виртуалните таблици са още познати с директния си превод от английски език като "изгледи". На практика виртуалната таблица е съхранен SQL SELECT оператор, който си има собствено име в базата данни. Използва се когато често използваме едни и същи SELECT заявки.

Виртуалните таблици имат и редица други предимства:

- Различните потребители в системата могат да виждат едни и същи данни по различен начин;
- Удобни са за ограничаване се достъпа на потребителите до базовата таблица и така те могат да достъпват само данните, които извежда виртуалната таблица.

Виртуална таблица се създава чрез операторът `CREATE VIEW`:

```
CREATE VIEW <име>(<имена на колони>)
AS SELECT <имена на колони> ... ;
```

Във вложения оператор SELECT не може да се използва ORDER BY и UNION. Важно е да има съответствие между върнатите колони от оператора SELECT и изброените след името на VIEW.

Пример: Създаване на виртуална таблица с данните на служителите на банка Bulbank:

```
CREATE VIEW bulbank_employees
 AS SELECT * FROM employees
    WHERE branch_id IN(
       SELECT id FROM branches
       WHERE bank_code IN(
          SELECT code FROM banks
          WHERE name = "Bulbank"
       )
    );
```

Сега можем да го използваме също както обикновена таблица:

```
SELECT * FROM bulbank_employees;
+----+----------------+-----------+
| id | name           | branch_id |
+----+----------------+-----------+
|  1 | Ivan Ivanov    |         1 |
|  3 | Mihail Zahariev |         1 |
|  4 | Milen Stoilov  |         2 |
|  5 | Svilen Petrov  |         2 |
|  6 | Ilian Stoianov |         2 |
|  7 | Petar Petrov   |         2 |
+----+----------------+-----------+
```

Можем да изпълняваме и заявки от тип INSERT, UPDATE и DELETE – те ще бъдат трансформирани автоматично от системата за управление на бази данни върху базовата таблица.

Ако в операторът SELECT участва агрегатна функция или GROUP BY, то става напълно необновяемо. Изобщо използването на INSERT, UPDATE и DELETE при VIEW не се препоръчва. Най-често VIEW се използва “само за четене” и съответно потребителите се рестриктират да имат само права SELECT върху тези таблици.

За изтриването на VIEW става чрез оператора `DROP VIEW`:

```
DROP VIEW bulbank_employees CASCADE;
```

Ключовата дума CASCADE означава, че ако има производно на това VIEW (т.е. VIEW създадено чрез изтриваното VIEW), то също ще бъде изтрито. Алтернативата е с ключова дума RESTRICT, където ако има производно VIEW, то ще бъде върната грешка. По принцип не е добра идея да създавате производни една на друга виртуални таблици. Стойността по подразбиране на DROP VIEW е CASCADE.

Индекси

Индексите са обекти в базата данни, които ни осигуряват бърз достъп до редовете на базова таблица, чрез физическото представяне (адреси в паметта) на данните. Индексите се създават върху колони на таблиците.

Присъствието или отсъствието на индекс няма ефект върху крайния резултат на заявките. Единствената разлика е в евентуалното повишено бързодействие (при по-големи таблици може разликата да е огромна). Важно е обаче да създаваме индексите правилно, защото от това зависи дали системата ще ги използва или не.

За да демонстрираме повишеното бързодействие сме изпълнили една проста заявка на доста бавен компютър.:

```
SELECT * FROM accounts
WHERE customer_id = 11;
+----+--------+------+-----------+-------------+
| id | amount | type | branch_id | customer_id |
+----+--------+------+-----------+-------------+
| 12 |  99.18 |    1 |         2 |          11 |
+----+--------+------+-----------+-------------+
```

Нека сега създадем индекс по колоната, която участва в условието WHERE на оператора SELECT и изпълним заявката отново:

```
CREATE INDEX accounts_customer_id
ON accounts(customer_id);

SELECT * FROM accounts
WHERE customer_id = 11;
+----+--------+------+-----------+-------------+
| id | amount | type | branch_id | customer_id |
+----+--------+------+-----------+-------------+
| 12 |  99.18 |    1 |         2 |          11 |
+----+--------+------+-----------+-------------+
```

Виждате, че разликата в бързодействието е 10 пъти! За тази тестова постановка изтрих базата данни, рестартирах сървъра и я създадох наново, защото обикновено системата за управление на бази данни си пази cache на вече използвани заявки. Също така трябва да отбележа, че в този тестов пример колоната customer_id НЕ Е foreign key. Системата за управление на бази данни обикновено създава автоматично индекси по първични и външни ключове.

Обикновено за структура от данни на индексите се използва BTREE (бинарно дърво). При различните системи има възможности за използване на алтернативни структури, но това не е задължително.

Истината е, че индексите не винаги са подходящи. В редица случаи те може да навредят на производителността на системата. Това се получава ако имаме таблици, върху които често правим заявки от тип UPDATE, INSERT и DELETE, то всеки път индексът трябва да се обновява. Затова таблици, в които често се добавят данни и рядко се четат такива е по-добре да не използваме индекс.

Същото важи и за индексите върху колони с много повтарящи се редове. Ако имаме такава колона, в която данните се повтарят изключително много, то по-добре да не създаваме индекс, защото ефектът ще бъде на забавяне на системата. Създавайте предимно индекси върху колони с ключ UNIQUE.

Ако вашите таблици се обновяват периодично (например в края на месеца се обновява таблица със статистики), то следвайте следната последователност:
- Изтриване на индекса;
- Обновяване на данните;
- Създаване на индекса отново.

Ето още един пример за създаване и изтриване на индекс:

```
CREATE INDEX customers_name
ON customers(name);

DROP INDEX customers_name
ON customers;
```

Можете лесно да се досетите, че е възможно да направите повече от един индекс върху една таблица. В последствие когато правите заявки SELECT MySQL обикновено се досеща кой индекс е най-подходящ при заявката. Въпреки това е възможно вие сами да укажете кой индекс да се използва:

```
SELECT * FROM mytable
USE INDEX (col1index, col2index)
WHERE col1 = 1 AND col2 = 2;
```

Ако пък желаете е възможно да укажете кой индекс да НЕ се използва чрез командата “IGNORE INDEX”:

```
SELECT * FROM mytable
IGNORE INDEX (col1index)
WHERE col1 = 1 AND col2 = 2;
```

Транзакции

Транзакция наричаме последователност от SQL заявки, които трябва да изпълняват условието или всичките да бъдат изпълнени или нито една от тях да

не бъде изпълнена. Може да дадем класически пример с банковите транзакции. Например ако искаме да прехвърлим 50 лева от акаунт 1 в акаунт 2, то трябва да изпълним следните две заявки:

```
UPDATE accounts
SET amount = amount - 50
WHERE id = 1;

UPDATE accounts
SET amount = amount + 50
WHERE id = 2;
```

Какво обаче ще се случи ако първата заявка се изпълни, но поради някаква причина втората не (например възникне грешка)? Отговорът е, че парите ще бъдат изгубени. Тук на помощ ни идват именно транзакциите – те гарантират че ако някоя заявка не се изпълни, то данните ще бъдат възстановени в първоначалния им вид.

Групирането на заявки в транзакция се изпълнява изключително лесно. Единствено трябва да оградим данните с BEGIN (започване на транзакция) и COMMIT (край на транзакция):

```
BEGIN;

UPDATE accounts
SET amount = amount - 50
WHERE id = 1;

UPDATE accounts
SET amount = amount + 50
WHERE id = 2;

COMMIT;
```

Ако някоя от транзакциите пропадне, то се прави т.нар. ROLLBACK. За целта се използва innodb log файл, в който се записват старите данни преди изпълнението на всяка заявка. Естествено ние можем да правим ROLLBACK и сами. Например:

```
SELECT amount
FROM accounts
WHERE id = 1;
+--------+
| amount |
+--------+
| 106.38 |
+--------+

BEGIN;

UPDATE accounts
SET amount = amount - 50
WHERE id = 1;

ROLLBACK;
```

```
SELECT amount
FROM accounts
WHERE id = 1;
+--------+
| amount |
+--------+
| 106.38 |
+--------+
```

Виждате, че резултатите от заявката SELECT са едни и същи, тоест ROLLBACK е "върнал" данните в първоначалния им вид преди заявката UPDATE.

Транзакциите трябва да отговарят на условие за консистентно четене. Това означава, че всеки SELECT чете данните записани точно след последния COMMIT.

Важно е да споменем, че при InnoDB всяка заявка, която НЕ участва в блокове "BEGIN – COMMIT" е автоматично записана, т.е. можем да приемем, че единичните заявки са завършени транзакции сами по себе си.

Забележка: За стартиране на транзакция можете да използвате и по-популярната сред останалите СУБД команда "START TRANSACTION".

Заключване на данните при транзакция

Синхронизацията на данните е изключително важна. За да демонстрираме това нека покажем първо един пример. Нека проверим колко пари има в акаунт с id = 1:

```
USE banks;
Database changed

SELECT amount FROM accounts
WHERE id = 1;
+--------+
| amount |
+--------+
| 306.38 |
+--------+
```

Сега нека напишем заявка UPDATE, с която искаме да изтеглим 500 лева, но така, че ако искаме да няма такава наличност, то заявката да не се изпълни:

```
UPDATE accounts
SET amount = amount - 500
WHERE id = 1 AND amount >= 500;

SELECT amount FROM accounts
WHERE id = 1;
```

```
+--------+
| amount |
+--------+
| 306.38 |
+--------+
```

Нека сега стартираме две връзки към базата от данни и да започнем транзакции. С тези транзакции ние ще се опитаме да изтеглим два пъти по 250 лева от акаунт с id = 1.

Виждаме, че втората връзка не върна резултат – тя стои в "спящ" (idle) режим. Това е така, защото сървъра знае, че в този момент друга транзакция все още не е приключила, тоест резултатът от нея все още не е записан. Така втората връзка все още не "вижда", че вече сумата в акаунта е намалена и ако вземе 250 лева, то банката ще изгуби пари. Затова при стартиране на транзакция се прави т.нар. "заключване" (lock).

Нека сега приключим първата транзакция. Виждаме, че това се отрази моментално във втората връзка. Както очаквахме там UPDATE не се направи, защото в акаунта вече няма достатъчно пари (те бяха взети от първата връзка).

Същото заключване на транзакции важи и за INSERT заявки. Нека създадем една примерна елементарна база от данни:

```
CREATE DATABASE locks;

USE locks;

CREATE TABLE test(
                    val INT
       ) ENGINE=InnoDB;

INSERT INTO test
       VALUES (1), (2);
```

Нека сега стартираме две транзакции, с които искаме да прочетем най-голямата стойност в колоната val и съответно добавят нова стойност с едно по-голяма от втората.

Виждаме, че отново втората транзакция изчаква завършването на първата. В момента, в който извършим COMMIT в първата транзакция втората също ще изпълни заявката си. Сега ще видим, че се е получило точно това, което искахме – първата транзакция е добавила val = 3, а втората val = 4.

Ако заключването на връзката не съществуваше, то и двете транзакции щяха да вмъкнат стойност "3" и щяхме да имаме повтарящи се редове.

От тези примери трябва да си извадим важна бележка – важно е да приключваме транзакциите си възможно най-бързо. Ако ние се "бавим", то други транзакции ще трябва да ни изчакват. В MySQL има стандартен timeout от няколко секунди (естествено може да бъде настройван чрез my.cnf/my.ini) за изчакване – ако дадена заявка е в режим "изчакване" и този timeout бъде достигнат, то заявката ще пропадне.

Заключването на връзките важи за заявки UPDATE, INSERT и DELETE. По подразбиране то не е валидно за заявки от тип SELECT. MySQL обаче ни предоставя възможност да го правим ръчно:

```
SELECT <rows> FROM <table> FOR UPDATE;
```

В последния пример указваме, че данните, които са върнати от SELECT заявката, ще бъдат заключени. По този начин ако някоя друга транзакция се опита да чете или променя данните, то тя ще трябва да изчака изпълнението на първата започната.

По-слабо заключване е "`LOCK IN SHARE MODE`", с което позволяваме на други транзакции да четат, но не и да променят данните. Чрез такова заключване сме сигурни, че прочетените данни ще бъдат най-новите налични в системата и никоя друга транзакция няма да ги промени междувременно:

```
SELECT <rows> FROM <table> LOCK IN SHARE MODE;
```

С последния пример при отваряне на втора връзка всички видове заявки ще бъдат блокирани в idle режим докато първата транзакция не завърши.

Процедури и входни параметри

Процедурите или иначе казано Stored Procedures ни дават възможност да създаваме скриптове за извършване на типизирани заявки с различни входни данни. Нека демонстрираме една елементарна процедура, която извиква обикновена заявка SELECT:

```
DELIMITER |

CREATE PROCEDURE show_customers()
BEGIN
SELECT * FROM customers;
END
|
DELIMITER ;

CALL show_customers();
+----+------------------+---------+----------+
| id | name             | address | bank_mgr |
+----+------------------+---------+----------+
|  1 | Todor Ivanov     | NULL    |        1 |
|  2 | Petko Stoianov   | NULL    |        1 |
|  3 | Neno Nenov       | NULL    |     NULL |
|  4 | Mariana Zaharieva | NULL    |        3 |
|  5 | Elica Zaharieva  | NULL    |        3 |
|  6 | Atanas Petrov    | NULL    |        4 |
|  7 | Ivan Ivanov      | NULL    |        4 |
|  8 | Zlatomir Petrov  | NULL    |        4 |
|  9 | Mihail Ivchev    | NULL    |        5 |
| 11 | Ivailo Ivanov    | NULL    |        7 |
| 12 | George Lucas     | NULL    |     NULL |
```

```
| 13 | George Harison    | NULL    |     NULL |
| 14 | Michael Jackson   | NULL    |     NULL |
| 15 | Tony Martin       | NULL    |     NULL |
| 16 | Tony McCarter     | NULL    |     NULL |
| 17 | Alexander Smith   | NULL    |       11 |
| 18 | Maria Smith       | NULL    |       11 |
| 19 | Alain Delrick     | NULL    |       12 |
| 20 | Devry Henry       | NULL    |       12 |
| 21 | Lenard Renne      | NULL    |       12 |
| 22 | Fontaine Rupert   | NULL    |       13 |
+----+-------------------+---------+----------+
```

Обърнете внимание на командата “DELIMITER”. Чрез първото й изпълнение указваме, че края на заявка вече няма да става с “;”, а с “|”. Това е нужно, защото при създаването на процедурата трябва да използваме символа “;” при края на заявката SELECT. DELIMITER може да бъде всяка комбинация от символи.

Можем да създаваме процедури за изпълнение на повече от една заявка наведнъж. Освен това една процедура може да изпълнява различни заявки, в зависимост от подадени към нея входни данни. За целта на помощ идват т.нар. параметри (parameters). Нека демонстрираме как се дефинира променлива (параметър) в MySQL с елементарен пример от базата от данни banks:

```
USE banks;

SET @cust_name = 'Ivan Ivanov';

SELECT * FROM CUSTOMERS
       WHERE name = @cust_name;
+----+------------+---------+----------+
| id | name       | address | bank_mgr |
+----+------------+---------+----------+
|  7 | Ivan Ivanov | NULL    |        4 |
+----+------------+---------+----------+
```

Казахме, че има аналогия с променливи в потребителски сесии – това е така, защото тази променлива е валидна само за текущата връзка. При приключване на връзката тя ще бъде унищожена.

Процедурите могат да достъпват параметри по три различни начина:

1. IN:

```
DELIMITER |

CREATE PROCEDURE proc_in(IN var VARCHAR(255))
BEGIN
   SET @cust_name = var;
END
|

DELIMITER ;

CALL proc_in('Atanas Petrov');

SELECT * FROM customers WHERE name = @cust_name;
```

```
+----+---------------+---------+----------+
| id | name          | address | bank_mgr |
+----+---------------+---------+----------+
|  6 | Atanas Petrov | NULL    |        4 |
+----+---------------+---------+----------+
```

Виждаме как към процедурата подадохме параметър и чрез него променихме стойността на променливата, която беше дефинирана глобално.

Трябва да знаете, че ако към IN параметър на процедура подадете параметър вместо конкретна стойност и го промените вътре в процедурата, то този параметър ще се върне в първоначалното си състояние след приключване на процедурата. Можете да си направите аналогия с предаването на параметър към функция по стойност от програмирането на C++.

2. OUT:

```
DELIMITER |

CREATE PROCEDURE proc_out(OUT var VARCHAR(255))
BEGIN
  SET var = 'Todor Ivanov';
END
|

DELIMITER ;

CALL proc_out(@cust_name);

SELECT * FROM customers WHERE name = @cust_name;
+----+--------------+---------+----------+
| id | name         | address | bank_mgr |
+----+--------------+---------+----------+
|  1 | Todor Ivanov | NULL    |        1 |
+----+--------------+---------+----------+
```

Тук подадохме променливата като входен параметър на процедурата и я променихме вътре във функцията. Виждате, че след приключването на процедурата глобалния параметър остана променен – това нямаше да стане, ако го бяхме направили с IN. Можете да си направите аналогия с подаването на параметър към функция като псевдоним в програмирането на C++.

3. INOUT:

```
SET @newvar = 'Petko Stoianov';

DELIMITER |
CREATE PROCEDURE proc_inout(INOUT var VARCHAR(255))
BEGIN
  SET var = @cust_name;
END
|

DELIMITER ;
```

```
CALL proc_inout(@newvar);

SELECT * FROM customers WHERE name = @newvar;
+----+-------------+---------+----------+
| id | name        | address | bank_mgr |
+----+-------------+---------+----------+
|  1 | Todor Ivanov | NULL    |        1 |
+----+-------------+---------+----------+
```

Виждаме, че променливата @newvar беше със стойност ‘Petko Stoianov’, но след изпълнението на процедурата я променихме със стойността на @cust_name (‘Todor Ivanov’). Разликата между OUT и INOUT параметрите е, че при INOUT параметъра трябва да бъде дефиниран глобално преди извикването на функцията – при OUT не е така (ако не съществува ще бъде създаден).

Сега вече можем да създаваме и по-смислени процедури. Нека дадем един пример – процедура, която премества пари от един акаунт в друг. В случая ще използваме IN входен параметър (в процедурата не променяме стойностите на параметри):

```
USE banks;

DELIMITER |

CREATE PROCEDURE transfer_money(acc_in INT, acc_out INT,
money DOUBLE)
        BEGIN
                START TRANSACTION;

                UPDATE accounts
                SET amount = amount - money
                WHERE id = acc_out;

                UPDATE accounts
                SET amount = amount + money
                WHERE id = acc_in;

                COMMIT;

        END
        |

DELIMITER ;
```

Ето как можем да преместим 20 лева от акаунт номер 5 в акаунт номер 6:

```
SELECT id, amount
        FROM accounts
        WHERE id = 5 OR id = 6;
+----+---------+
| id | amount  |
+----+---------+
|  5 |  211.98 |
|  6 | 1200.00 |
+----+---------+
```

```
CALL transfer_money(6, 5, 20);

SELECT id, amount
       FROM accounts
       WHERE id = 5 OR id = 6;
+----+---------+
| id | amount  |
+----+---------+
|  5 |  191.98 |
|  6 | 1220.00 |
+----+---------+
```

Логически оператори и цикли

Чрез процедурите MySQL много наподобява завършен език за програмиране. За това силно спомагат възможностите за логически оператори и цикли. Ще ги разгледаме поотделно:

1. **IF-ELSE**:

Операторите IF-ELSE имат следната структура:

```
IF <условие>
   THEN <заявки>;
   ELSE <заявки>;
END IF;
```

Нека демонстрираме с един пример – процедура, на която подаваме параметри сума и номер на акаунт. Процедурата връща резултат "1" ако в акаунта има повече пари от посочените или "0" в противен случай:

```
DELIMITER |

CREATE PROCEDURE check_availability(IN acc INT, IN money
DECIMAL)
        BEGIN

                DECLARE acc_avail DECIMAL;

                SELECT amount
                INTO acc_avail
                FROM accounts
                WHERE id = acc;

                IF (acc_avail >= money)
                       THEN SELECT 1;
                       ELSE SELECT 0;
                END IF;

        END
```

```
        |

DELIMITER ;

SELECT id, amount FROM accounts WHERE id = 5;
+----+--------+
| id | amount |
+----+--------+
|  5 | 191.98 |
+----+--------+

CALL check_availability(5, 200);
+---+
| 0 |
+---+
| 0 |
+---+

CALL check_availability(5, 150);
+---+
| 1 |
+---+
| 1 |
+---+
```

Виждате, че се получи точно желания резултат – в акаунт №5 има точно 191.98. Когато попитаме процедурата дали има 200 тя връща резултат 0, а когато я попитаме дали има 150 връща резултат 1.

Искаме да обърнем внимание на заявката “DECLARE x INT”. Чрез нея ние дефинираме т.нар. локална променлива за процедурата. Тя е валидна само вътре в процедурата и се изтрива след нейното приключване. Виждате, че в случая я инициализирахме като резултат от временна таблица (SELECT -> INTO).

Трябва много да внимавате при евентуално подаване на NULL стойности. Както и при обикновените заявки, всяко сравнение с NULL стойност връща резултат FALSE.

2. **CASE:**

CASE е аналог на оператора за многовариантен избор switch в езика за програмиране C. Синтаксисът е следния:

```
CASE <променлива>
   WHEN <условие>
      THEN <заявки>;
   WHEN <условие>
      THEN <заявки>;
   ...
   ELSE <заявки>
END CASE;
```

Частта ELSE се достига тогава, когато нито едно от условията по-горе не е изпълнено.

Нека демонстрираме с пример – процедура, която по зададен номер на акаунт връща името на типа му:

```
DELIMITER |
CREATE PROCEDURE acc_type(IN acc INT)
       BEGIN

              DECLARE acc_t TINYINT;

              SELECT type
              INTO acc_t
              FROM accounts
              WHERE id = acc;

              CASE acc_t
              WHEN 1 THEN
                     SELECT "3 months deposit" AS acctype;
              WHEN 2 THEN
                     SELECT "Annual deposit" AS acctype;
              ELSE
                     SELECT "Unknown account" AS acctype;
              END CASE;

       END
       |

DELIMITER ;

SELECT id, type FROM accounts WHERE id = 6 OR id = 7;
+----+------+
| id | type |
+----+------+
|  6 |    2 |
|  7 |    1 |
+----+------+

CALL acc_type(6);
+----------------+
| acctype        |
+----------------+
| Annual deposit |
+----------------+

CALL acc_type(7);
+------------------+
| acctype          |
+------------------+
| 3 months deposit |
+------------------+
```

3. **WHILE**:

Процедурите в MySQL добиват още по-голяма сила с наличието на цикли. Първият и може би най-популярен е WHILE:

```
WHILE <условие>
```

```
DO
   <заявки>;
END WHILE;
```

Например следната процедура ще изведе сумите по акаунти на всички клиенти в зададен диапазон (x,y):

```
DELIMITER |

CREATE PROCEDURE check_clients(IN x INT, IN y INT)
       BEGIN

              DECLARE iterator INT;

              SET iterator = x;

              WHILE (iterator >= x AND iterator <= y)
              DO
                     SELECT id, amount
                     FROM accounts
                     WHERE id = iterator;

                     SET iterator = iterator + 1;
              END WHILE;

       END
       |

DELIMITER ;

CALL check_clients(5,9);
```

```
+----+--------+
| id | amount |
+----+--------+
|  5 | 191.98 |
+----+--------+
```

```
+----+---------+
| id | amount  |
+----+---------+
|  6 | 1220.00 |
+----+---------+
```

```
+----+--------+
| id | amount |
+----+--------+
|  7 | 133.48 |
+----+--------+
```

```
+----+--------+
| id | amount |
+----+--------+
|  8 | 256.41 |
+----+--------+
```

```
+----+---------+
| id | amount  |
+----+---------+
|  9 | 1331.50 |
+----+---------+
```

4. **REPEAT-UNTIL**:

Цикълът е с абсолютно същото действие както WHILE, с изключение, че ако условието е погрешно в самото начало, то въпреки това тялото на цикъла ще се изпълни поне веднъж:

```
REPEAT
   <заявки>;
UNTIL <условие>;
END REPEAT;
```

Тригери

Тригерите са начин за автоматизиране на действия свързани с обработката на информация в базите от данни. Те са процедури, които се изпълняват при извикване на заявки insert, update или delete. Ако приемем изпълнението на такива заявки за "събитие", то тригерите са процедури, които се изпълняват преди или след дадено събитие.

Нека покажем един пример – ще създадем таблица с няколко футболни отбора:

```
CREATE DATABASE football;
USE football;

CREATE TABLE clubs(
  id TINYINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL UNIQUE,
  points TINYINT UNSIGNED NULL DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO clubs(name, points)
VALUES ('Славия', 0),
       ('ЦСКА', 0),
       ('Левски', 0);
```

И таблица с възможните мачове помежду им:

```
CREATE TABLE matches(
  hostID TINYINT UNSIGNED NOT NULL,
  FOREIGN KEY (hostID) REFERENCES clubs(id)
     ON DELETE CASCADE ON UPDATE CASCADE,
  guestID TINYINT UNSIGNED NOT NULL,
  FOREIGN KEY (guestID) REFERENCES clubs(id)
     ON DELETE CASCADE ON UPDATE CASCADE,
```

```
  hostGoals TINYINT UNSIGNED NULL DEFAULT NULL,
  guestGoals TINYINT UNSIGNED NULL DEFAULT NULL,
  PRIMARY KEY(hostID, guestID)
)ENGINE=InnoDB;

INSERT INTO matches(hostID, guestID)
SELECT host.id AS dname, guest.id AS gname
FROM clubs AS host JOIN clubs AS guest
     ON host.id <> guest.id;
```

Преди започването на турнира точките на отборите са 0, а мачовете все още не са изиграни, т.е. стойностите за "голове на домакина" (hostGoals) и "голове на госта" (guestGoals) не са попълнени:

SELECT * FROM clubs;

```
+----+--------+--------+
| id | name   | points |
+----+--------+--------+
|  1 | Славия |      0 |
|  2 | ЦСКА   |      0 |
|  3 | Левски |      0 |
+----+--------+--------+
```

SELECT * FROM matches;

```
+--------+---------+-----------+------------+
| hostID | guestID | hostGoals | guestGoals |
+--------+---------+-----------+------------+
|      1 |       2 |      NULL |       NULL |
|      1 |       3 |      NULL |       NULL |
|      2 |       1 |      NULL |       NULL |
|      2 |       3 |      NULL |       NULL |
|      3 |       1 |      NULL |       NULL |
|      3 |       2 |      NULL |       NULL |
+--------+---------+-----------+------------+
```

Сега искаме да реализираме следната функционалност – когато два отбора изиграят даден мач, то ние ще записваме резултата между тях, а след това ще обновяваме точките (3 за победа, 1 за равенство и нищо за загуба). Бихме могли да направим това с две заявки – една UPDATE заявка за таблица matches и една UPDATE заявка за таблица clubs. Двете обаче са свързани и зависими помежду си, следователно можем да осъществим обновяването на точките чрез тригер:

```
DELIMITER //
  CREATE TRIGGER pointsUpdate
  AFTER UPDATE ON matches FOR EACH ROW
    BEGIN
      IF NEW.hostGoals > NEW.guestGoals
      THEN UPDATE clubs
           SET clubs.points = clubs.points + 3
           WHERE clubs.id = OLD.hostID;
      ELSEIF NEW.hostGoals < NEW.guestGoals
      THEN UPDATE clubs
           SET clubs.points = clubs.points + 3
           WHERE clubs.id = OLD.guestID;
      ELSE UPDATE clubs
           SET clubs.points = clubs.points + 1
```

```
            WHERE clubs.id = OLD.hostID
                  OR clubs.id = OLD.guestID;
      END IF;
    END;//
DELIMITER ;
```

Виждате, че може да се възползвате от всичко, което беше налично при съхранените процедури. Ключовите думи NEW и OLD означават съответно "нова" и "стара" стойност. В случая когато четем стойностите hostGoals и guestGoals задължително трябва да взимаме новите им стойности (старите са били NULL). За hostID и guestID няма значение дали ще използваме NEW или OLD, защото тези колони няма да ги променяме с update заявките. Важно е да се отбележи, че OLD стойностите са само за четене, докато при нужда NEW могат да се променят от тригера.

Нека изпробваме – нека се е изиграл първият мач между Славия и ЦСКА с резултат 1:0, втория между Славия и Левски 2:2, а третия между Левски и ЦСКА 0:2:

```
UPDATE matches
SET hostGoals=1, guestGoals=0
WHERE hostID=1 AND guestID=2;

UPDATE matches
SET hostGoals=2, guestGoals=2
WHERE hostID=1 AND guestID=3;

UPDATE matches
SET hostGoals=0, guestGoals=2
WHERE hostID=3 AND guestID=2;
```

Да видим мачовете и съответното класиране:

SELECT * FROM matches;

```
+--------+---------+-----------+------------+
| hostID | guestID | hostGoals | guestGoals |
+--------+---------+-----------+------------+
|      1 |       2 |         1 |          0 |
|      1 |       3 |         2 |          2 |
|      2 |       1 |      NULL |       NULL |
|      2 |       3 |      NULL |       NULL |
|      3 |       1 |      NULL |       NULL |
|      3 |       2 |         0 |          2 |
+--------+---------+-----------+------------+
```

SELECT * FROM clubs;

```
+----+--------+--------+
| id | name   | points |
+----+--------+--------+
|  1 | Славия |      4 |
|  2 | ЦСКА   |      3 |
|  3 | Левски |      1 |
+----+--------+--------+
```

Така създаденият тригер от примера ще свърши отредената му работа, но е далеч от "перфектен". Ако например сте направили грешка и сте въвели даден

резултат невярно, то може би ще искате в последствие да го поправите. Да, но всяка UPDATE заявка ще обновява класирането, т.е. ако например сме решили, че Славия е победила ЦСКА с 1:0, а после сме се поправили и сме променили резултата като 2:0, то с горния тригер ще се получи така, че Славия ще е получила 6 точки от този мач, а не 3. Именно за такива ситуации трябва или много да внимавате, или да предвиждате подобни действия:

```
DROP TRIGGER pointsUpdate;

DELIMITER //
CREATE TRIGGER pointsUpdate
AFTER UPDATE ON matches FOR EACH ROW
  BEGIN
    IF OLD.hostGoals IS NOT NULL
       AND OLD.guestGoals IS NOT NULL
    THEN IF OLD.hostGoals > OLD.guestGoals
         THEN UPDATE clubs
              SET clubs.points = clubs.points - 3
              WHERE clubs.id = OLD.hostID;
         ELSEIF OLD.hostGoals < OLD.guestGoals
         THEN UPDATE clubs
              SET clubs.points = clubs.points - 3
              WHERE clubs.id = OLD.guestID;
         ELSE UPDATE clubs
              SET clubs.points = clubs.points - 1
              WHERE clubs.id = OLD.hostID
                    OR clubs.id = OLD.guestID;
         END IF;
    END IF;

    IF NEW.hostGoals > NEW.guestGoals
    THEN UPDATE clubs
         SET clubs.points = clubs.points + 3
         WHERE clubs.id = OLD.hostID;
    ELSEIF NEW.hostGoals < NEW.guestGoals
    THEN UPDATE clubs
         SET clubs.points = clubs.points + 3
         WHERE clubs.id = OLD.guestID;
    ELSE UPDATE clubs
         SET clubs.points = clubs.points + 1
         WHERE clubs.id = OLD.hostID
               OR clubs.id = OLD.guestID;
    END IF;
  END;//
DELIMITER ;
```

Така вече намаляваме точките ако вече има записан резултат в системата преди да правим обновяване. Проверете горния тригер с различни примери (*).

(*) Все още не сме се предпазили напълно от невалидни данни. Ако например се въведе резултат (X, NULL) или (NULL, X) в системата (т.е. една от стойностите hostGoals и guestGoals е NULL), то със сигурност хем мачът ще е невалиден, хем точките ще се обновят погрешно. Един начин да поправим това е като създадем

тригери, които ще променят стойностите ПРЕДИ обновяването на данните, като ако една от двете (но НЕ И ДВЕТЕ) е NULL, то ще я променим на 0:

```
DELIMITER //
CREATE TRIGGER matchValidatorInsert
BEFORE INSERT ON matches FOR EACH ROW
  BEGIN
    IF NEW.hostGoals IS NULL
    AND NEW.guestGoals IS NOT NULL
    THEN SET NEW.hostGoals = 0;
    END IF;

    IF NEW.guestGoals IS NULL
    AND NEW.hostGoals IS NOT NULL
    THEN SET NEW.guestGoals = 0;
    END IF;
  END; //
DELIMITER ;

DELIMITER //
CREATE TRIGGER matchValidatorUpdate
BEFORE UPDATE ON matches FOR EACH ROW
  BEGIN
    IF NEW.hostGoals IS NULL
    AND NEW.guestGoals IS NOT NULL
    THEN SET NEW.hostGoals = 0;
    END IF;

    IF NEW.guestGoals IS NULL
    AND NEW.hostGoals IS NOT NULL
    THEN SET NEW.guestGoals = 0;
    END IF;
  END; //
DELIMITER ;
```

Задача 1. Направете тригер clubsAdd, който при въвеждане на нов футболен клуб в таблица clubs добавя редове с възможните му мачове с всички други отбори в таблица matches, като головете и за домакин и за гост са NULL.

Решение: Ще се възползваме от фактът, че hostGoals и guestGoals са ‘null default null’ и ще използваме стойностите им по подразбиране (null):

```
DELIMITER //
CREATE TRIGGER clubsAdd
AFTER INSERT ON clubs FOR EACH ROW
  BEGIN
    INSERT INTO matches(hostID, guestID)
    SELECT host.id AS dname, guest.id AS gname
    FROM clubs AS host JOIN clubs AS guest
         ON host.id <> guest.id
       AND( host.id = NEW.id
            OR
            guest.id = NEW.id
      );
```

```
  END;//
DELIMITER ;
```

Задача 2. Направете тригер matchDelete, който при изтриване на даден футболен мач от програмата (таблица matches) обновява точките на отборите в таблица clubs, но само ако резултатът от този мач е бил различен от (NULL, NULL).

Решение:

```
DELIMITER //
CREATE TRIGGER matchDelete
AFTER DELETE ON matches FOR EACH ROW
  BEGIN
    IF OLD.hostGoals IS NOT NULL
       OR OLD.guestGoals IS NOT NULL
    THEN
        IF OLD.hostGoals > OLD.guestGoals
      THEN UPDATE clubs
           SET clubs.points = clubs.points - 3
           WHERE clubs.id = OLD.hostID;
      ELSEIF OLD.hostGoals < OLD.guestGoals
      THEN UPDATE clubs
           SET clubs.points = clubs.points - 3
           WHERE clubs.id = OLD.guestID;
      ELSE UPDATE clubs
           SET clubs.points = clubs.points - 1
           WHERE clubs.id = OLD.hostID
                 OR clubs.id = OLD.guestID;
      END IF;
    END IF;
  END;//
DELIMITER ;
```

Задача 3. Направете тригер matchesAdd, който при добавяне на нови мачове в програмата в таблица matches (например разширяваме турнира с нови мачове или ако сме изтрили даден мач и после сме го вмъкнали отново) прави съответните обновявания на точките в таблица clubs (само ако при INSERT заявката са указани голове – ако резултатите от добавените мачове са с null стойности, то няма нужда да се прави!). Забележете, че за да можете да добавяте нови мачове, то или преди това трябва да изтриете съществуващ мач или да премахнете PRIMARY KEY от таблица matches – в създадената по-горе таблица не е възможно да въведете един и същи мач повече от два пъти.

Решение:

```
DELIMITER //
CREATE TRIGGER extraMatchAdd
AFTER INSERT ON matches FOR EACH ROW
  BEGIN
    IF NEW.hostGoals IS NOT NULL
       AND NEW.guestGoals IS NOT NULL
    THEN IF NEW.hostGoals > NEW.guestGoals
         THEN UPDATE clubs
```

```
                SET clubs.points = clubs.points + 3
                WHERE clubs.id = NEW.hostID;
          ELSEIF NEW.hostGoals < NEW.guestGoals
          THEN UPDATE clubs
                SET clubs.points = clubs.points + 3
                WHERE clubs.id = NEW.guestID;
          ELSE UPDATE clubs
                SET clubs.points = clubs.points + 1
                WHERE clubs.id = NEW.hostID
                      OR clubs.id = NEW.guestID;
          END IF;
    END IF;
  END;//
DELIMITER ;
```

Внимание (!!!). Очевидно тук тригерът от задача 1 (който добавя редове в таблица matches при добавяне на ред в таблица clubs) би задействал и тригерът от задача 3 (който обновява таблица clubs при добавяне на редове в таблица matches). За щастие в този конкретен при случай тригерът в задача 3 няма да направи нищо и няма да има проблем. При всички случаи обаче трябва изключително много да внимавате в такива ситуации (когато един тригер активира друг) . Ето ви един пример, в който се получава “патова ситуация” и изпълнението не може да продължи:

```
CREATE TABLE a(
      ID INT
);

CREATE TABLE b(
    ID INT
);

DELIMITER //
CREATE TRIGGER proba
AFTER INSERT ON a FOR EACH ROW
  BEGIN
    INSERT INTO b(ID)
      VALUES (NEW.ID);
  END; //
DELIMITER ;

DELIMITER //
CREATE TRIGGER proba2
AFTER INSERT ON b FOR EACH ROW
  BEGIN
    INSERT INTO a(ID)
      VALUES (NEW.ID);
  END; //
DELIMITER ;

INSERT INTO a(ID) VALUES (1);
ERROR 1442 (HY000): Can't update table 'a' in stored
function/trigger because it is already used by statement
which invoked this stored function/trigger.
```

```
INSERT INTO b(ID) VALUES (1);
ERROR 1442 (HY000): Can't update table 'b' in stored
function/trigger because it is already used by statement
which invoked this stored function/trigger.
```

Освен това ако използвате тригери, то вече няма да може да може да използвате вложени заявки, които извличат информация от таблицата, която ще бъде обновявана:

```
UPDATE matches
SET hostGoals=1, guestGoals=1
   WHERE hostID=(
   SELECT id FROM clubs WHERE name="Левски"
   )
   AND
   guestID=(
   SELECT id FROM clubs WHERE name="Славия"
   );
ERROR 1442 (HY000): Can't update table 'clubs' in stored
function/trigger because it is already used by statement
which invoked this stored function/trigger
```

Затова винаги планирайте тригерите си много внимателно.

Важно: Поне до версия 5.5 на MySQL тригерите НЕ се активират при каскадно изтриване/обновяване на данни.

UNION

Не споменахме една пренебрегната досега възможност на SQL, а именно – обединението. То се използва, за да може две или повече SELECT заявки да бъдат комбинирани в една резултатна таблица. Нека преди да демонстрираме да създадем една примерна база от данни:

```
CREATE DATABASE unions;

CREATE TABLE vegetables(
    `id` INT NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255) NULL DEFAULT NULL,
    `price` DOUBLE NULL DEFAULT NULL,
    PRIMARY KEY (`id`)
);

CREATE TABLE fruits(
    `id` INT NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255) NULL DEFAULT NULL,
    `price` DOUBLE NULL DEFAULT NULL,
    PRIMARY KEY (`id`)
);
```

```
INSERT INTO vegetables(`id`, `name`, `price`)
VALUES    (NULL, 'krastavici', 4.00),
    (NULL, 'domati', 5.00),
    (NULL, 'kartofi', 3.00);

INSERT INTO fruits(`id`, `name`, `price`)
VALUES    (NULL, 'banani', 3.00),
    (NULL, 'iagodi', 2.50),
    (NULL, 'chereshi', 7.00);
```

Сега искаме да създадем таблица, която да изведе имената както на плодовете, така и на зеленчуците. Познатия досега вариант е с използване на JOIN, но така таблицата ще бъде с две колони. Ако искаме резултата да е подреден в една колона, то използваме UNION:

```
(SELECT name as Fruits_and_veggies
FROM vegetables)
UNION
(SELECT name
FROM fruits);
+--------------------+
| Fruits_and_veggies |
+--------------------+
| krastavici         |
| domati             |
| kartofi            |
| banani             |
| iagodi             |
| chereshi           |
+--------------------+
```

Както виждате името на колоната се приема за името от първата таблица. Естествено можем да направим UNION и по две колони:

```
(SELECT name as Fruits_and_veggies, price
FROM vegetables)
UNION
(SELECT name, price
FROM fruits);
+--------------------+-------+
| Fruits_and_veggies | price |
+--------------------+-------+
| krastavici         |     4 |
| domati             |     5 |
| kartofi            |     3 |
| banani             |     3 |
| iagodi             |   2.5 |
| chereshi           |     7 |
+--------------------+-------+
```

Не е задължително данните да са от един и същ тип. Важно е обаче броя на колоните от първия SELECT да съвпада с броя на колоните на втория! В противен случай няма как да се извърши обединението.

След като натрупате опит в използването на UNION рано или късно ще достигнете до един първоначално изглеждащ странен ефект – дублиращите се редове автоматично се премахват! Ето един пример:

```
(SELECT price
FROM vegetables)
UNION
(SELECT price
FROM fruits);
+-------+
| price |
+-------+
|     4 |
|     5 |
|     3 |
|   2.5 |
|     7 |
+-------+
```

Знаем, че имаме общо 6 плода и зеленчука в таблиците, а в резултата от цените излязоха само 5. Това е така, защото цената на banani и kartofi е една и съща и UNION е премахнал дублиращите се редове. Ако това предизвиква потенциален проблем във вашата заявка, то трябва да използвате допълнителната дума ALL:

```
(SELECT price
FROM vegetables)
UNION ALL
(SELECT price
FROM fruits);
+-------+
| price |
+-------+
|     4 |
|     5 |
|     3 |
|     3 |
|   2.5 |
|     7 |
+-------+
```

Естествено възможно е UNION да е част от под-заявка (например вложен SELECT). Затова ограждането на заявките в скоби не е задължително, но е силно препоръчително.

FULL JOIN в MySQL

След като научихме заявките, използващи UNION, вече сме готови да посочим как се прави и липсващия в MySQL FULL JOIN. Ще използваме таблиците с плодове и зеленчуци.

Нека припомним как работеха LEFT и RIGHT JOIN. За целта ще направим многотаблична заявка по колоната “цена”:

```
SELECT * FROM vegetables
LEFT JOIN fruits ON vegetables.price = fruits.price;
+----+-----------+-------+------+--------+-------+
| id | name      | price | id   | name   | price |
+----+-----------+-------+------+--------+-------+
|  1 | krastavici |     4 | NULL | NULL   |  NULL |
|  2 | domati     |     5 | NULL | NULL   |  NULL |
|  3 | kartofi    |     3 |    1 | banani |     3 |
+----+-----------+-------+------+--------+-------+
```

Резултата е точно това, което очаквахме – само banani имат една и съща цена с kartofi и затова от втората таблица само един ред не е с резултат NULL. В RIGHT JOIN се случваше същото, но в обратна посока:

```
SELECT * FROM vegetables
RIGHT JOIN fruits ON vegetables.price = fruits.price;
+------+---------+-------+----+----------+-------+
| id   | name    | price | id | name     | price |
+------+---------+-------+----+----------+-------+
|    3 | kartofi |     3 |  1 | banani   |     3 |
| NULL | NULL    |  NULL |  2 | iagodi   |   2.5 |
| NULL | NULL    |  NULL |  3 | chereshi |     7 |
+------+---------+-------+----+----------+-------+
```

Е, вече може би се досетихте как можем да постигнем FULL JOIN – просто трябва да обединим резултата от двете заявки и да премахнем дублиращите се редове (т.е. да използваме UNION без ALL):

```
(SELECT * FROM vegetables
LEFT JOIN fruits ON vegetables.price = fruits.price)
UNION
(SELECT * FROM vegetables
RIGHT JOIN fruits ON vegetables.price = fruits.price);
+------+-----------+-------+------+----------+-------+
| id   | name      | price | id   | name     | price |
+------+-----------+-------+------+----------+-------+
|    1 | krastavici |     4 | NULL | NULL     |  NULL |
|    2 | domati     |     5 | NULL | NULL     |  NULL |
|    3 | kartofi    |     3 |    1 | banani   |     3 |
| NULL | NULL       |  NULL |    2 | iagodi   |   2.5 |
| NULL | NULL       |  NULL |    3 | chereshi |     7 |
+------+-----------+-------+------+----------+-------+
```

Редовете са точно 5, а това е именно каквото очаквахме от FULL JOIN. Този метод върши работа в почти всички случай, но за съжаление НЕ покрива 100% стандарта на определението за FULL JOIN. Проблемът е, че UNION премахва всички дублиращи се редове, а FULL JOIN не го прави ако има такива. За да се демонстрира това трябва в една от таблиците да има дублиращи се редове – в този случай по стандарта за FULL JOIN ще трябва този ред да излезе два пъти, а с горната реализация няма да се получи. Ето един пример:

```
INSERT INTO vegetables (`id`, `name`, `price`)
VALUES (NULL, 'krastavici', 4);

SELECT * FROM vegetables;
```

```
+----+------------+-------+
| id | name       | price |
+----+------------+-------+
|  1 | krastavici |     4 |
|  2 | domati     |     5 |
|  3 | kartofi    |     3 |
|  4 | krastavici |     4 |
+----+------------+-------+
```

```
(SELECT vegetables.name, fruits.name FROM vegetables
LEFT JOIN fruits ON vegetables.price = fruits.price)
UNION
(SELECT vegetables.name, fruits.name FROM vegetables
RIGHT JOIN fruits ON vegetables.price = fruits.price);
+------------+----------+
| name       | name     |
+------------+----------+
| krastavici | NULL     |
| domati     | NULL     |
| kartofi    | banani   |
| NULL       | iagodi   |
| NULL       | chereshi |
+------------+----------+
```

Виждаме, че в резултата има само едни ‘krastavici’, а при истински FULL JOIN трябваше да са две. Този проблем няма да се реши и с UNION ALL, защото в този случай ще се появят други дублиращи се редове, които НЕ трябва да присъстват:

```
(SELECT vegetables.name, fruits.name FROM vegetables
LEFT JOIN fruits ON vegetables.price = fruits.price)
UNION ALL
(SELECT vegetables.name, fruits.name FROM vegetables
RIGHT JOIN fruits ON vegetables.price = fruits.price);
+------------+----------+
| name       | name     |
+------------+----------+
| krastavici | NULL     |
| domati     | NULL     |
| kartofi    | banani   |
| krastavici | NULL     |
| kartofi    | banani   |
| NULL       | iagodi   |
| NULL       | chereshi |
+------------+----------+
```

Тук “krastavici” се появяват два пъти, както трябва да бъде, но реда “kartofi | banani” се появява два пъти, а не трябва да бъде така! Затова за реализация на FULL JOIN се използва друг подход, който е модификация на последния:

```
(SELECT vegetables.name, fruits.name FROM vegetables
LEFT JOIN fruits ON vegetables.price = fruits.price)
UNION ALL
(SELECT vegetables.name, fruits.name FROM vegetables
RIGHT JOIN fruits ON vegetables.price = fruits.price
WHERE vegetables.price IS NULL);
+------------+----------+
| name       | name     |
+------------+----------+
| krastavici | NULL     |
```

```
| domati     | NULL     |
| kartofi    | banani   |
| krastavici | NULL     |
| NULL       | iagodi   |
| NULL       | chereshi |
+------------+----------+
```

С добавеното условие WHERE ние премахнахме редовете, които вече са общи за двете таблици. Така вече имаме напълно функционална FULL JOIN заявка!

В по-старите версии на MySQL, където UNION не съществува, реализацията се прави чрез създаването на трета временна таблица.

MySQL OFFSET

Още в началото, когато се разглеждаха заявки за еднотабличен оператор SELECT, набързо се разгледа оператор LIMIT. Да припомним – той приемаше за параметър целочислено число X, чрез което от резултатната таблица се връщат само първите X реда, а останалите "се отрязват". Това естествено има редица приложения – разглеждане на най-новите записи от статистики, разглеждане на "най-добрите" резултати от състезание, извеждане на последните записи в таблица и т.н. Почти винаги, за такива случаи, оператор LIMIT е предхождан от ORDER BY.

Съвсем логичен въпрос обаче идва по-късно: "а какво да правим ако искаме не първите, а вторите X реда?". Ето ви стандартен пример – искате да направите сайт, в който да направите "страниране" на публикациите. На първо страница ще показвате първите 10 резултата, на втора резултатите от 11 до 20, на трета от 21 до 30, и т.н. Тук определено ще срещнете затруднение в писането на заявки за извеждането на по-задните страници.

Първият подход е да се използва т.нар. OFFSET. Нека например имаме таблица с много продукти. Ако желаем да изведем на екрана всички продукти от 101 до 110, то ще напишем следната заявка:

```
SELECT id, name
  FROM products
  ORDER BY id
  LIMIT 10 OFFSET 100;
```

OFFSET има смисъл на "отместване". Тук все пак трябва да споменем, че колкото по-голямо отместване правим, толкова по-бавно се изпълнява заявката. Затова при доста големи бази данни тези заявки биха породили сериозен проблем откъм производителност.

Алтернативата е да се правили лимитиране в WHERE клаузата. Например ако във въпросната таблица products сме убедени, че в id на продуктите "няма дупки", т.е. липсващи id, то можем да напишем горната заявка като:

```
SELECT id, name
```

```
FROM products
WHERE id BETWEEN 101 AND 110
ORDER BY id;
```

Тази заявка ще се изпълни доста по-бързо от предишната, но за съжаление със съответната цена. Ясно е, че при изтриване на един продукт с id в интервала [101, 110] ще се получи въпросната “дупка” и върнатите резултати няма да са 10, а 9. Затова използването на този подход следва да се прави само ако стриктно контролираме id-тата на продуктите (в случая от примера). Това означава, че при изтриване на продукт с id X ние трябва да се погрижим да понижим с 1 id-тата на всички продукти с id по-голямо от X, или още по-добре – да променим id-то на последния въведен продукт в системата на X, като по този начин предотвратим тази “дупка”. Както се досещате това евентуално може да доведе до редица други проблеми, както и доста по-голяма загуба на производителност, особено ако често изпълняваме заявки DELETE върху таблицата.

Източници

1. C, PHP, VB, .NET. http://www.cphpvb.net/category/db/
2. Hector Garsia-Molina, Jeffrey D. Ullman, Jennifer Widom (2002) Database Systems: The Complete Book
3. www.w3schools.com. The World's Largest Web Development Site. SQL Tutorial. http://www.w3schools.com/sql/default.asp
4. Бази данни. Христо Тужаров, 2007. http://tuj.asenevtsi.com/DB2007/index.htm