MySQL Relational Database

Езикът за релационни база данни е един с леки диалекти – просто мениджмънт системата за управление на данни е различна – Oracle, MariaDB, MySQL, etc.
https://www.w3schools.com/sql/sql_where.asp

За Judge на СофтУни – ако не е зададено другояче, базата данни /схемата не я цитираме като подаваме решенията си в Judge.

0. Някои basic неща

MySQL е case insensitive – команди можем да пишем както с големи, така и с малки букви
Слагаме коментари с: # или /*…. */
Ctrl + / - слага в коментар по друг начин
Полета/обекти винаги ограждаме с тилда кавички `….` като по този начин escape-ваме запазени думи в SQL
Ctlr + D – добавяме един ред като горния ред
Числата въвеждаме без скоби
Текстовете ограждаме с единични обикновени скоби ‘….’
За пари ползваме DECIMAL вместо DOUBLE
NULL
TRUE
FALSE
= присвоява стойност – ДА и знак за сравнение - ДА
SET e_count **:=** присвояване
!= или <> значат и двете различно, работи само за числови стойности. Иначе използваме IS NULL / IS NOT NULL
>= по-голямо
<= по-малко
Като кликнем колоните на таблица, то името на колоната се нанася в SQL заявката – да не си играем да пишем ръчно името на колоната
Като цяло, при базите данни, гледаме 90% предварително, и след това пишем заявките.
Ctrl + R – reverse Engineering
Ctrl + Space – Auto suggest
Реално не можем да дебъгваме в MySQL дадена функция/процедура/или какво и да е
+
-
*
/ - обикновено **дробно** делене
% или mod() модулно делене не работи по нормален начин

1. Introduction to MySQL

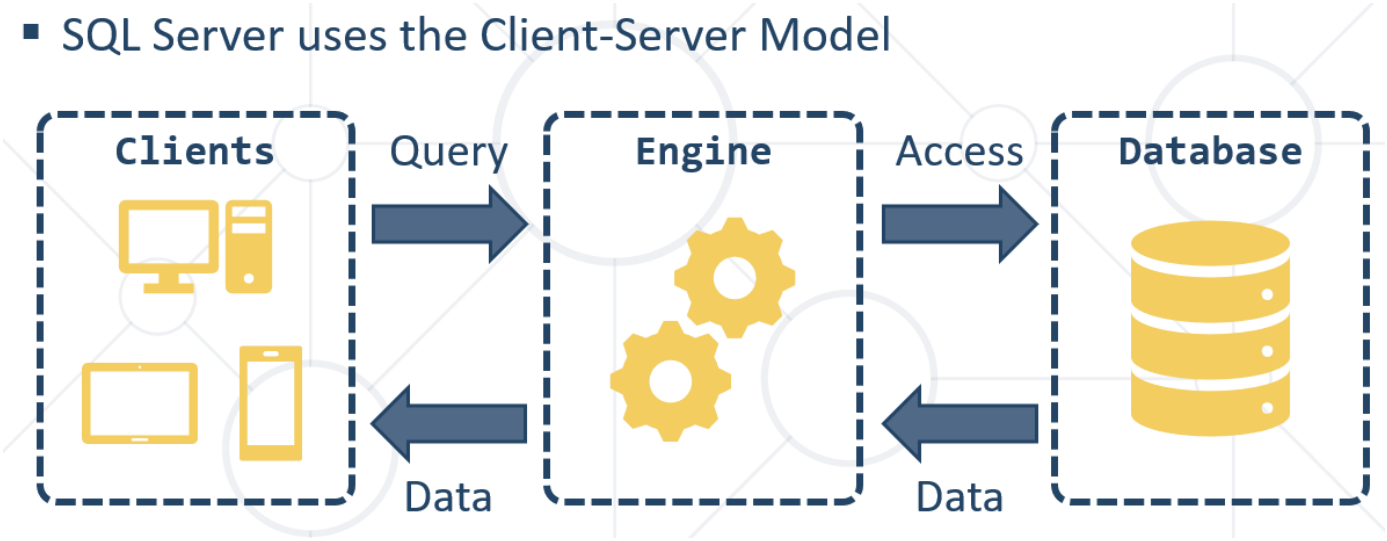
1.1. General info

- A database is an **organized** collection of **related** information
  - It imposes **rules** on the contained data
  - Access to data is usually provided by a "**system**" (DBMS) **database management**
  - Relational storage first proposed by Edgar Codd in 1970

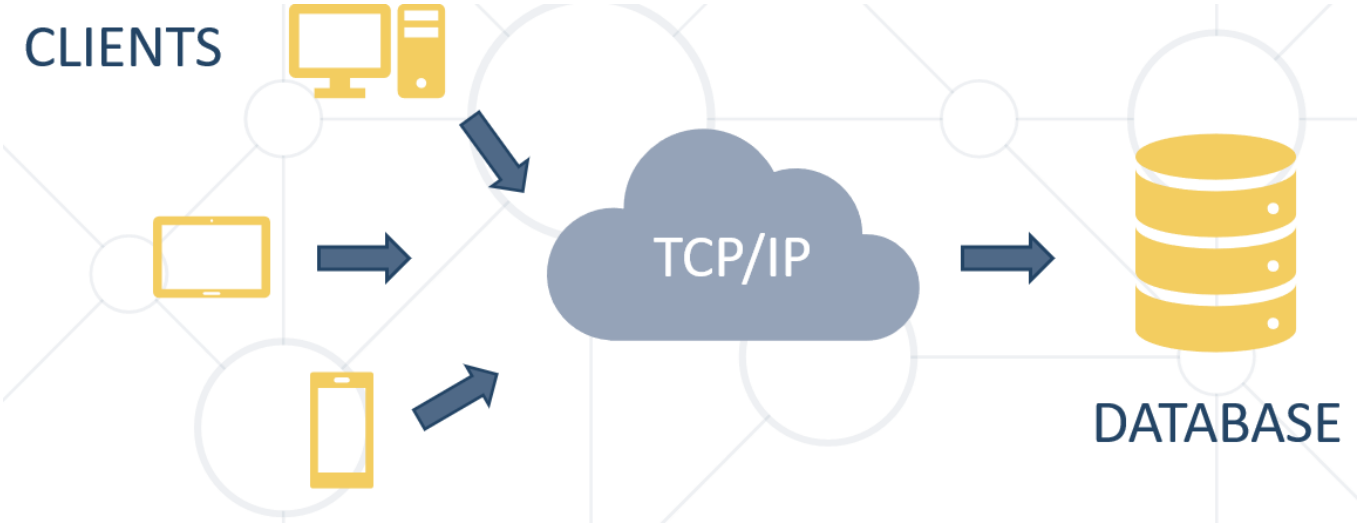
- **R**elational **D**ata **B**ase **M**anagement **S**ystem
  - Database **management**
  - It **parses requests** from the user and takes the **appropriate** action
  - The user **doesn't have direct access** to the stored data

- Data is presented by **relations** – collection of tables related by **common fields**
- MS SQL Server, DB2, Oracle and MySQL

1.2. Database Engine Flow



Client-Server Model

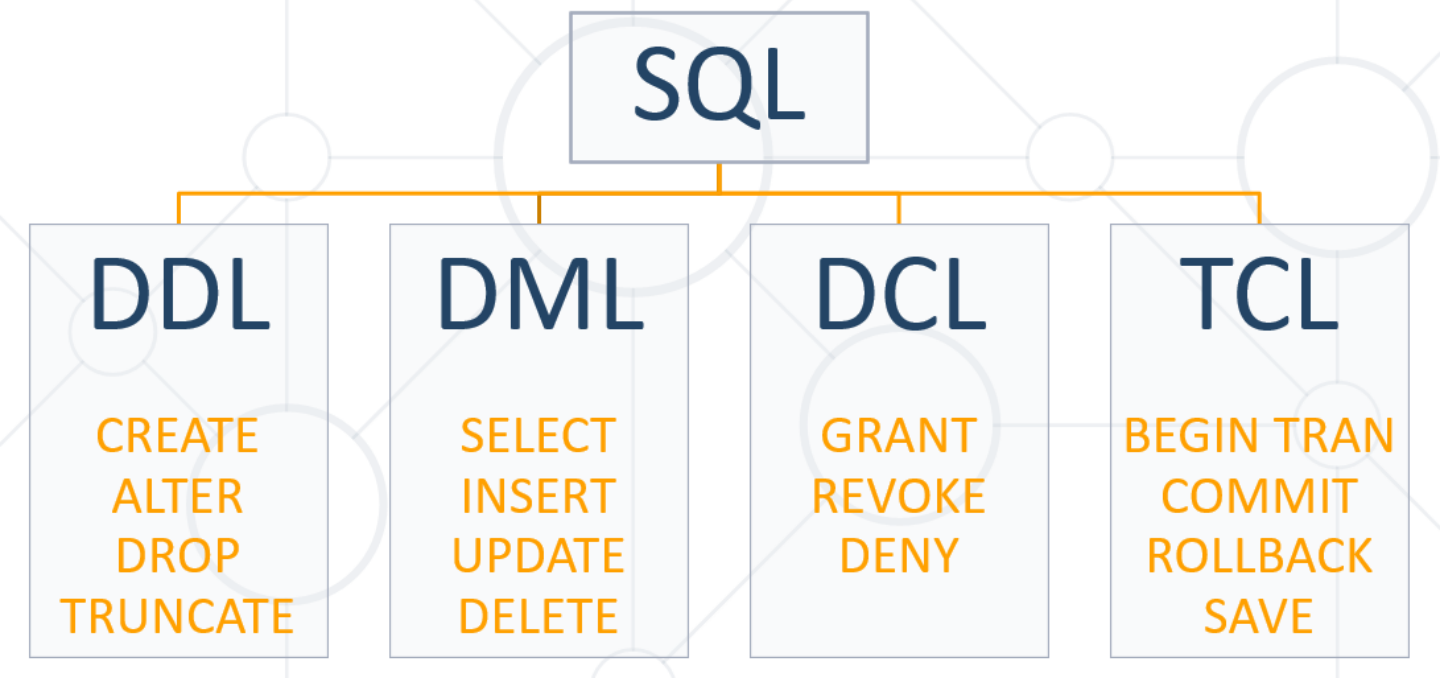


1.3. Structured Query Language = SQL

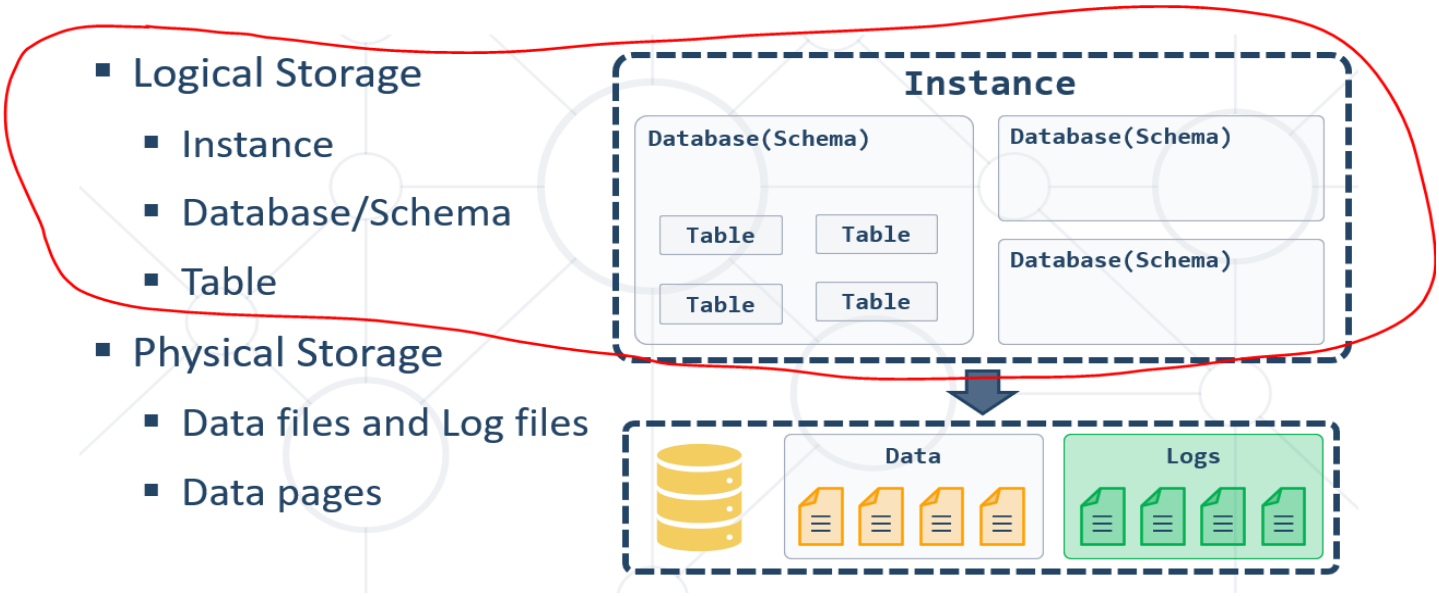
- Queries
- Clauses
- Expressions
- Predicates
- Statements

CRUD – Create, Read, Update, Delete

- Logically divided in four sections
  - **Data Definition Language**– describe the structure of our data = **DDL**
  - **Data Manipulation Language** – store and retrieve data = **DML**
  - **Data Control Language** – define who can access the data = **DCL**
  - **Transaction Control Language** – bundle operations and allow rollback = **TCL**



1.4. MySQL Server Architecture



1.5. Database Table Elements

- The table is the main **building block** of any database
- Each **row** is called a **record** or **entity**
- Columns (**fields**) define the **type** of data they contain

1.6. Table Relationships

- We split the data and introduce **relationships** between the tables to **avoid** repeating information
- Connection via **Foreign Key** in one table pointing to the **Primary Key** in another

user_id	first	last	registered
203	David	Rivers	05/02/2016
204	Sarah	Thorne	07/17/2016
205	Michael	Walters	11/23/2015

Primary Key

user_id	email
203	drivers@mail.cx
204	sarah@mail.cx
205	walters_michael@mail.cx
203	david@homedomain.cx

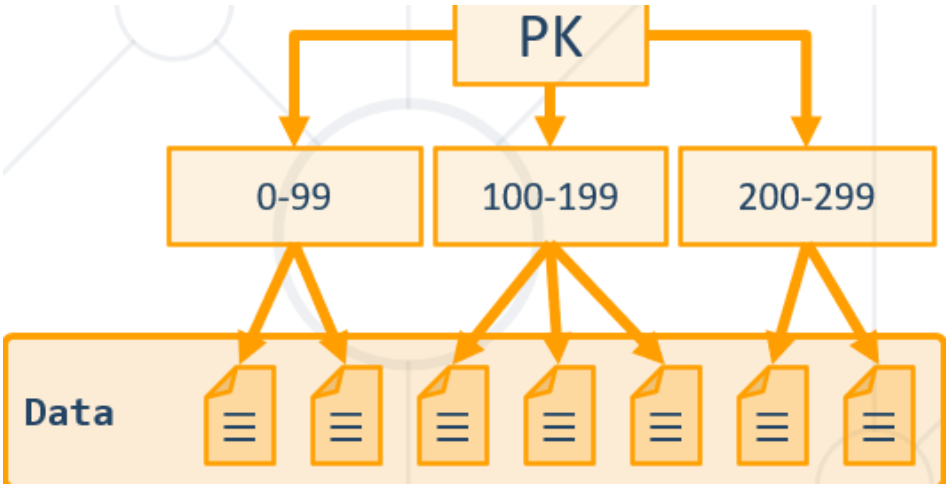
Foreign Key

1.7. Programmability

1.7.1. Indices

- Indices make data lookup faster
  - Clustered – bound to the **primary key**, physically sorts data
  - Non-Clustered – can be **any field**, references the primary index
- Structured as an **ordered tree**

Бинарно дърво – за по-бързо търсене в базата данни



1.7.2. Views

- Views are **prepared queries** for displaying **sections** of our data

```
CREATE VIEW v_employee_names AS
    SELECT employee_id,
           first_name,
           last_name
      FROM employees
```

```
SELECT * FROM v_employee_names
```

- Evaluated at **run time** – they do not increase performance

1.7.3. Procedures, Functions and Triggers

A database can further be customized with reusable code

- **Procedures** – carry out a predetermined **action**
  - E.g. get all employees with salary above 35000
- **Functions** – receive **parameters** and return a **result**
  - E.g. get the age of a person using their birthdate and current date
- **Triggers** – **watch** for activity in the database and **react** to it
  - E.g. when a record is deleted, write it to an archive

1.8. Data Types in MySQL Server

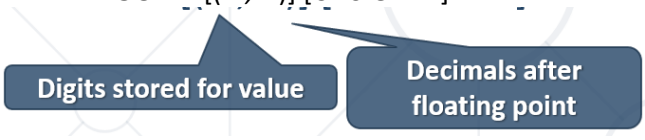
1.8.1. Numeric Data Types

- Numeric data types have certain range
- Their range can be changed if they are:
  - **Signed** - represent numbers both in the positive **and** negative ranges
  - **Unsigned -** represent numbers **only** in the positive range
- E.g. signed and unsigned INT:

Signed Range		Unsigned Range	
Min Value	Max Value	Min Value	Max Value
-2147483648	2147483648	0	4294967295

- **INT** [(*M*)] [UNSIGNED] INT(10) – число с 10 цифри

- **T**INYINT, **S**MALLINT, **M**EDIUMINT, **B**IGINT
- **DOUBLE** [(*M, D*)] [UNSIGNED]



  - E.g. DOUBLE(5, 2) – 999.99 – общо са 5 цифри, като след десетичната запетая са 2 цифри
- **DECIMAL** [(*M, D*)] [UNSIGNED] [**ZEROFILL**] – слага нули отпред ако е нужно

DECIMAL за по-голяма точност
M – общо брой цифри
D- от които след десетичната запетая

1.8.2 String Types

String column definitions include attributes that specify the **character set** or **collation**.

- **CHARACTER SET** (Encoding) - Determines the storage of each character (single or multiple bytes)

E.g. utf8, ucs2

- **CHARACTER COLLATION** – rules for encoding comparison - Determines the sorting order and case-sensitivity
  E.g. latin1_general_cs, Traditional_Spanish_ci_ai etc

- **Set and collation** can be defined at the database, table or column level

Non-unicode (just English, western languages)

- **CHAR** (M) - up to 255 characters
  - fixed-length character type (example CHAR(30) или CHAR(1))
- **VARCHAR**(M) **–** up to 65 535 characters
  - Variable max size
- **TEXT** – up to 65 535 characters
  - TINYTEXT, MEDIUMTEXT, LONGTEXT
- **BLOB - B**inary **L**arge **Ob**ject [(M)] - 65 535 ($2^{16} – 1$) characters – когато не пазим адреса/пътя към снимката, а пазим битовата символна версия от 1000+ символа на самата снимка
  - TINYBLOB, MEDIUMBLOB, LONGBLOB

TINYBLOB : L < 2^8 = 256 Bytes
BLOB : L < 2^16 = 65,536 Bytes
MEDIUMBLOB : L < 2^24 = 16,777,216 Bytes
LONGBLOB : L < 2^32 = 4,294,967,296 Bytes

fieldName Blob(size in bytes) -

The short answer **is**: **VARCHAR is** variable length, while **CHAR is** fixed length. **CHAR is** a fixed length string data type, so any remaining space **in the** field **is** padded with blanks. **CHAR** takes up 1 byte per character. ... **VARCHAR is** a variable length string data type, so it holds only the characters you assign to it.

Двойно повече място за Unicode (all languages worldwide) - Supports many client computers that are running different locales.
nchar/nvarchar - произлиза от national

	char	nchar	varchar	nvarchar
Character Data Type	Non-Unicode fixed-length	Unicode fixed-length can store both non-Unicode and Unicode	Non-Unicode variable length	Unicode variable length can store both non-Unicode and

		characters (i.e. Japanese, Korean etc.)		Unicode characters (i.e. Japanese, Korean etc.)
Maximum Length	up to 8,000 characters	up to 4,000 characters	up to 8,000 characters	up to 4,000 characters
Character Size	takes up 1 byte per character	takes up 2 bytes per Unicode/Non-Unicode character	takes up 1 byte per character	takes up 2 bytes per Unicode/Non-Unicode character
Storage Size	n bytes	2 times n bytes	Actual Length (in bytes)	2 times Actual Length (in bytes)
Usage	use when data length is constant or fixed length columns	use only if you need Unicode support such as the Japanese Kanji or Korean Hangul characters due to storage overhead	used when data length is variable or variable length columns and if actual data is always way less than capacity	use only if you need Unicode support such as the Japanese Kanji or Korean Hangul characters due to storage overhead
			query that uses a varchar parameter does an index seek due to column collation sets	query that uses a nvarchar parameter does an index scan due to column collation sets

1.8.3. Date Types

- **DATE -** for values with a date part but **no time part -** 'YYYY-MM-DD' or 'YY-MM-DD'
- **TIME -** for values with time but **no date part –** 'hh: mm: ss'
- **DATETIME -** values that contain both date **and** time parts - 'YYYY-MM-DD hh: mm: ss'
- **TIMESTAMP -** both date **and** time parts

- MySQL retrieves values for a given date type in a **standard output format**

E.g. as a string in either 'YYYY-MM-DD' or 'YY-MM-DD'

ВАЖНО
Когато сравняваме дата DATE с DATESTAMP, изречението „hired after 1/1/1999“ го тълкуваме
WHERE e.`hire_date` >= '1999-01-02'
защото имаме DATESTAMP 1999-12-12 **01:26**:00.000000

1.8.4. Boolean Types

`gender` **BOOLEAN**;

```
CREATE TABLE `people`(
`id` INT NOT NULL UNIQUE AUTO_INCREMENT PRIMARY KEY,
`name` VARCHAR(200) NOT NULL,
`picture` MEDIUMBLOB,
`height` DOUBLE(5,2),
`weight` DOUBLE(5,2),
`gender` CHAR(1) NOT NULL,
`birthdate` DATE NOT NULL,
`biography` LONGTEXT
);
```

1.9. DDL – Data Definition Language - Database Modelling – using GUI or via basic SQL queries

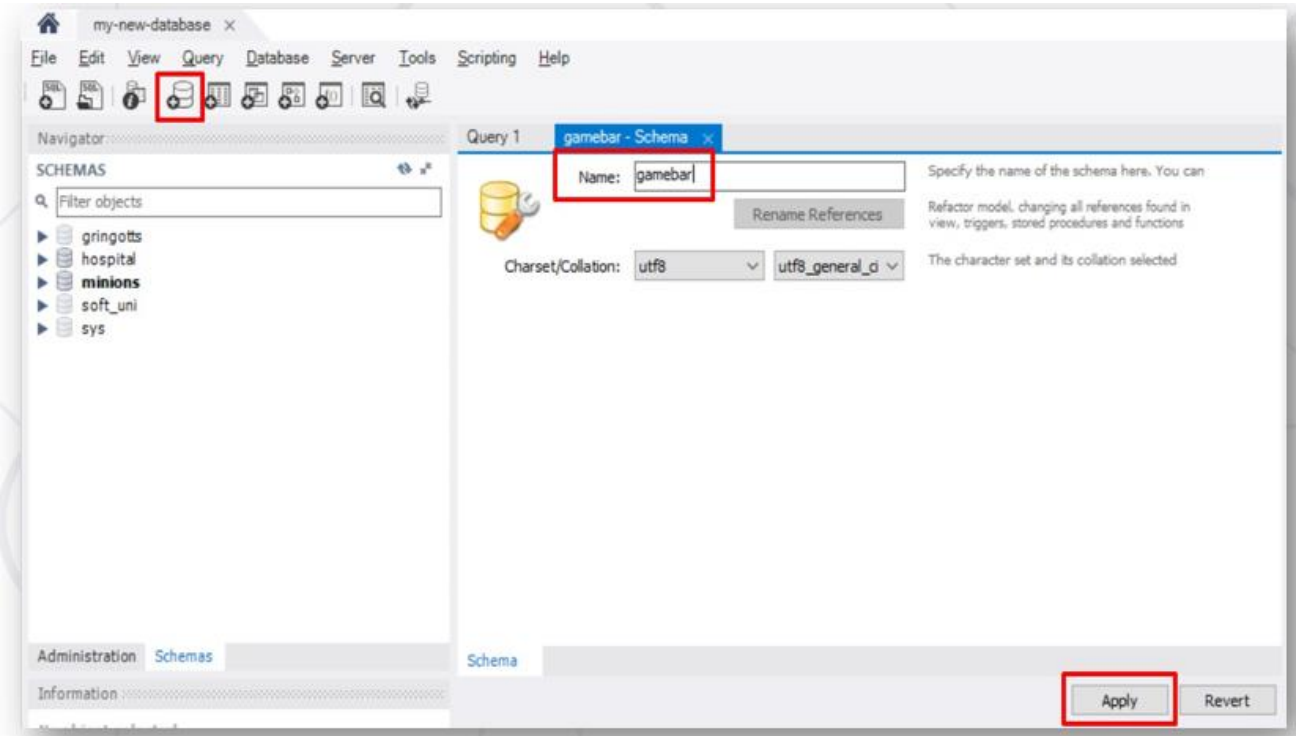
Да даваме Refresh от време на време

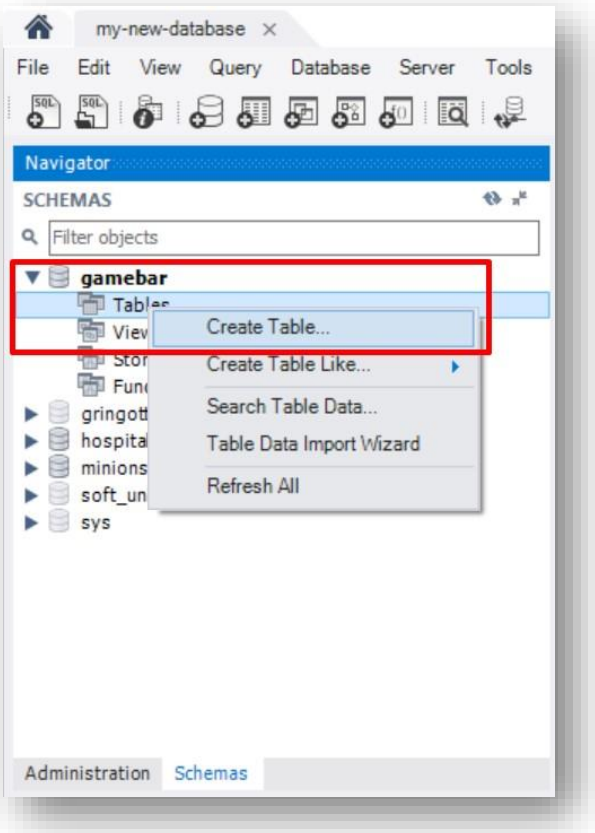
- **Working with IDEs – MySQL Workbench** Database ManagementSystem - we can use GUI Clients to **create** and **customize** tables

- Enables us:
  - To **create** a new database
  - To create **objects in the database** (tables, stored procedures, relationships and others)
  - To **change** the properties of objects
  - To **enter records** into the tables

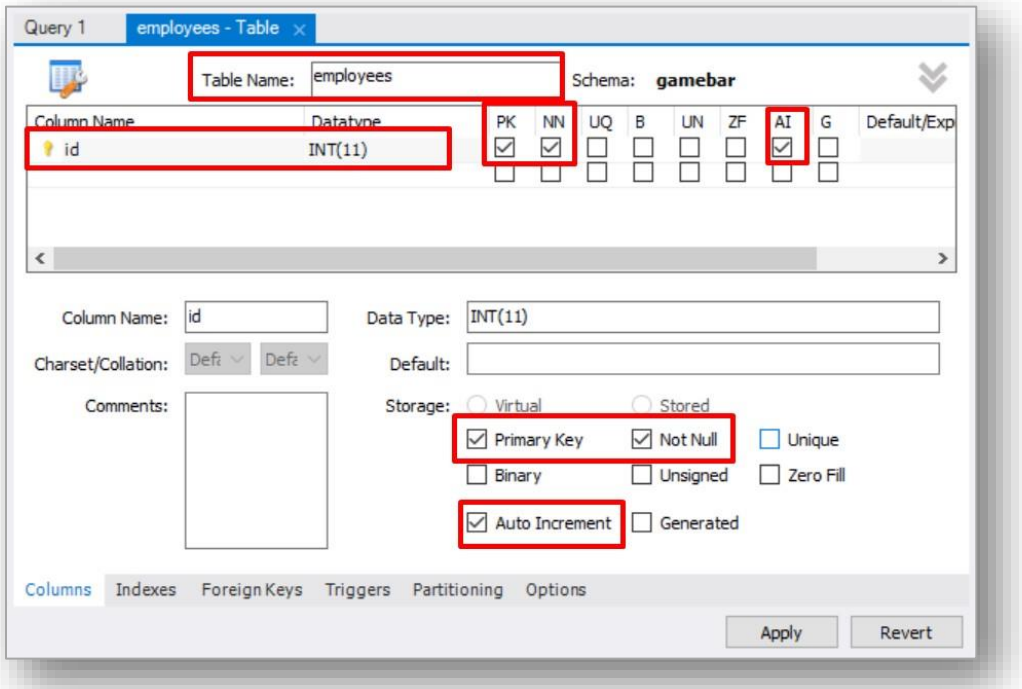
1.9.0. Creating a New Database

- Select **Create new schema** from the command menu

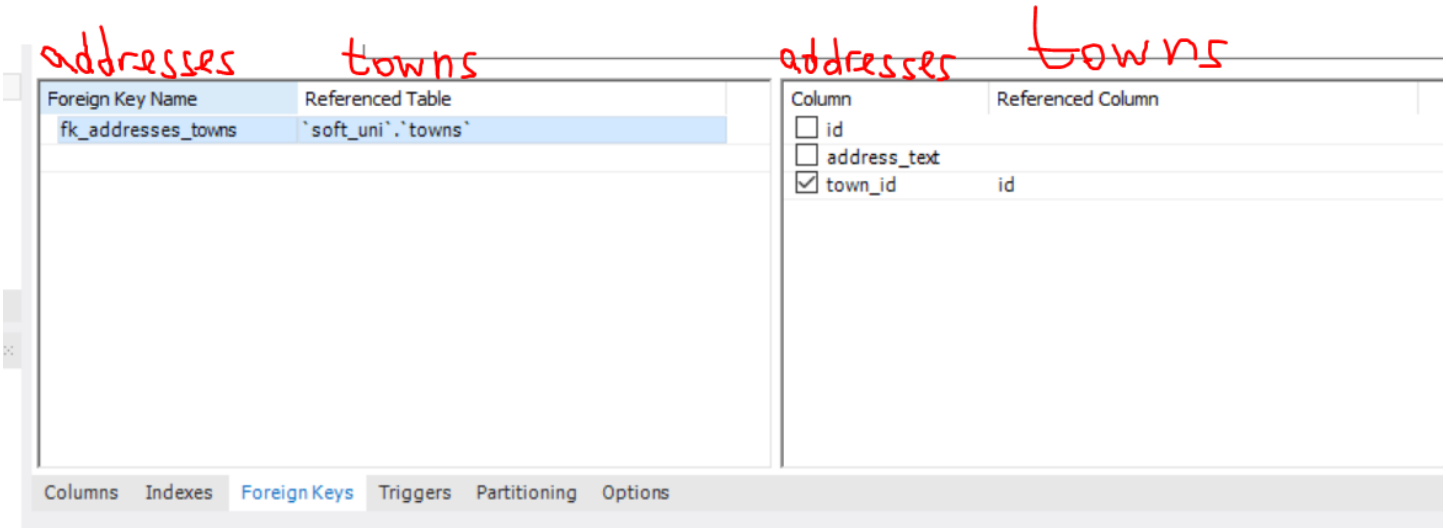




- **A Primary Key is used to uniquely identify and index records**



1.9.1. Foreign keys



- **Adding foreign keys**

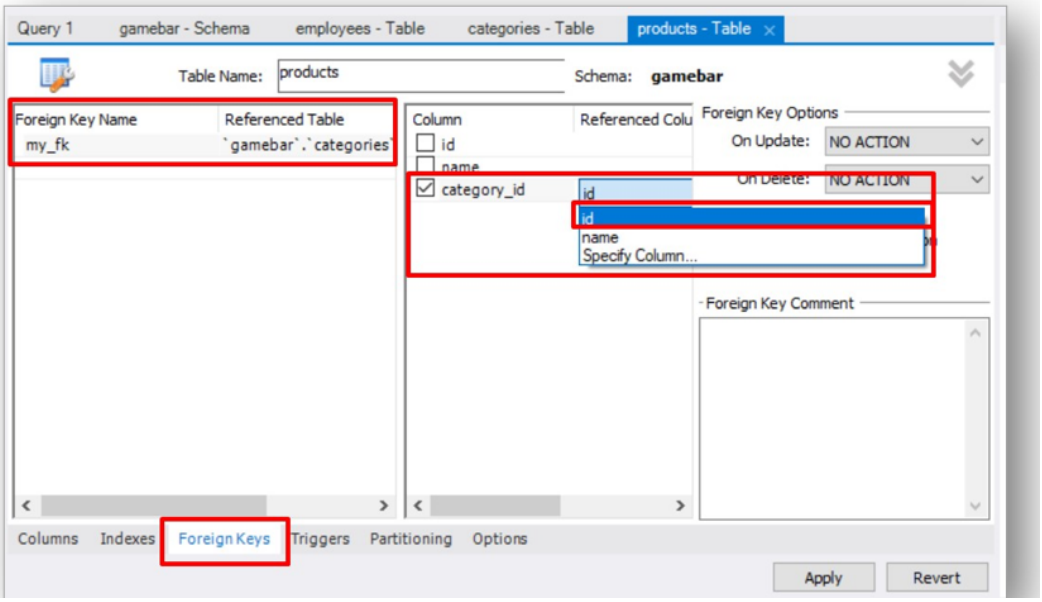
A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.
The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

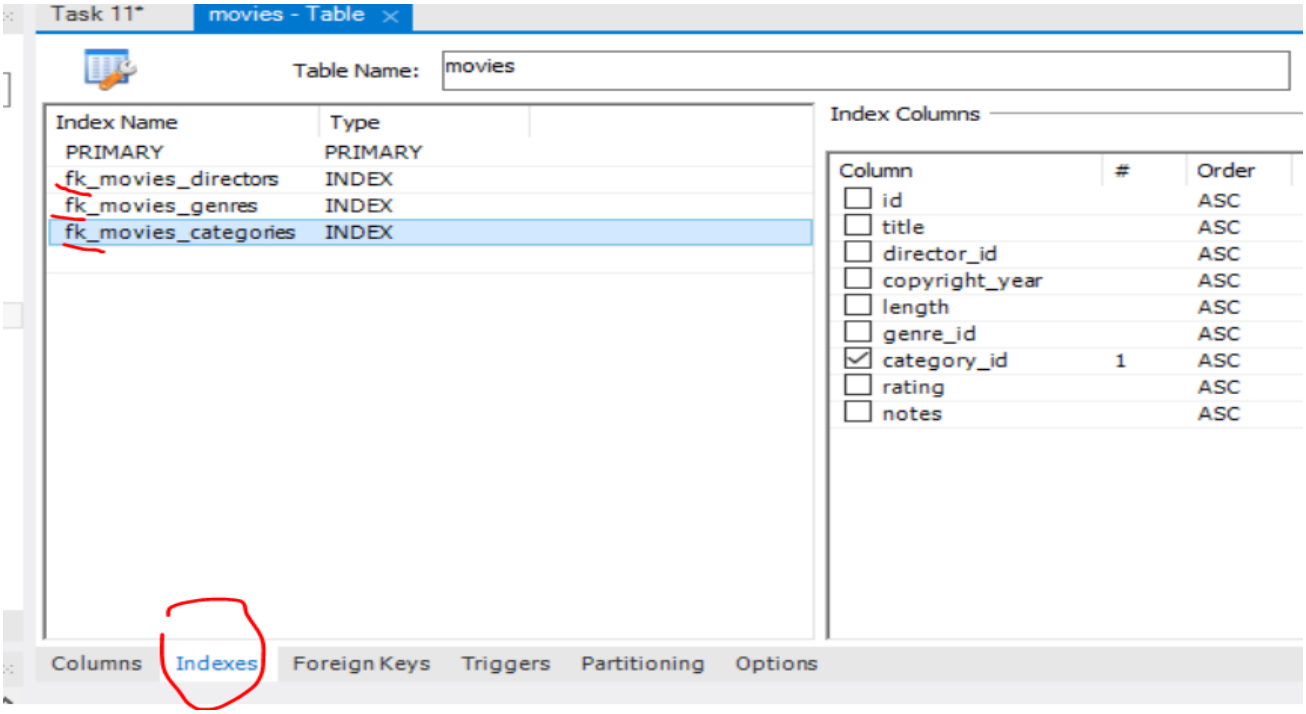
Конвенция при изписване на foreign key поле: fk_fromMinions_toTowns
В minions е чуждия ключ, а в Towns е primary ключ.

ВАЖНО: когато създаваме foreign keys, първо трябва да създадем таблицата, от която foreign key ще взема данни.

Foreign keys are created in the "**Foreign keys**" tab:

- **Reference table** – select the table from which you will choose a column to link your foreign key – "**categories**";
- **Columns** – select the column you want to be set as foreign key – "**category_id**";
- **Referenced columns** – select the column set to primary to link the foreign key – "**id**";






```
CREATE TABLE `Orders` (
    `OrderID` int NOT NULL,
    `OrderNumber` int NOT NULL,
    `PersonID` int,
    PRIMARY KEY (`OrderID`),

    CONSTRAINT `fk_source_target`
    FOREIGN KEY `Orders`(`PersonID`)
    REFERENCES `Persons`(`PersonID`)
);
```

SQL FOREIGN KEY on ALTER TABLE

```
ALTER TABLE `products`
ADD CONSTRAINT `fk_products_categories`
FOREIGN KEY `products`(`category_id`)
REFERENCES `categories`(`id`);
```

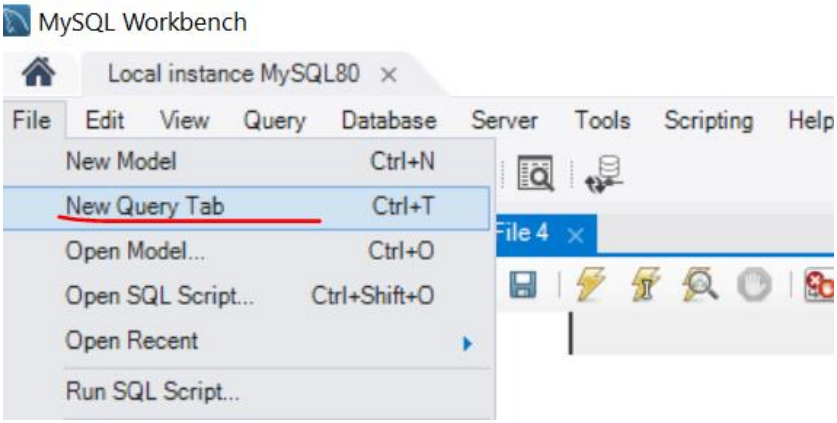
```
ALTER TABLE `minions`.`minions`
ADD CONSTRAINT `fk_minions_towns`
  FOREIGN KEY (`town_id`)
  REFERENCES `minions`.`towns` (`id`);
```

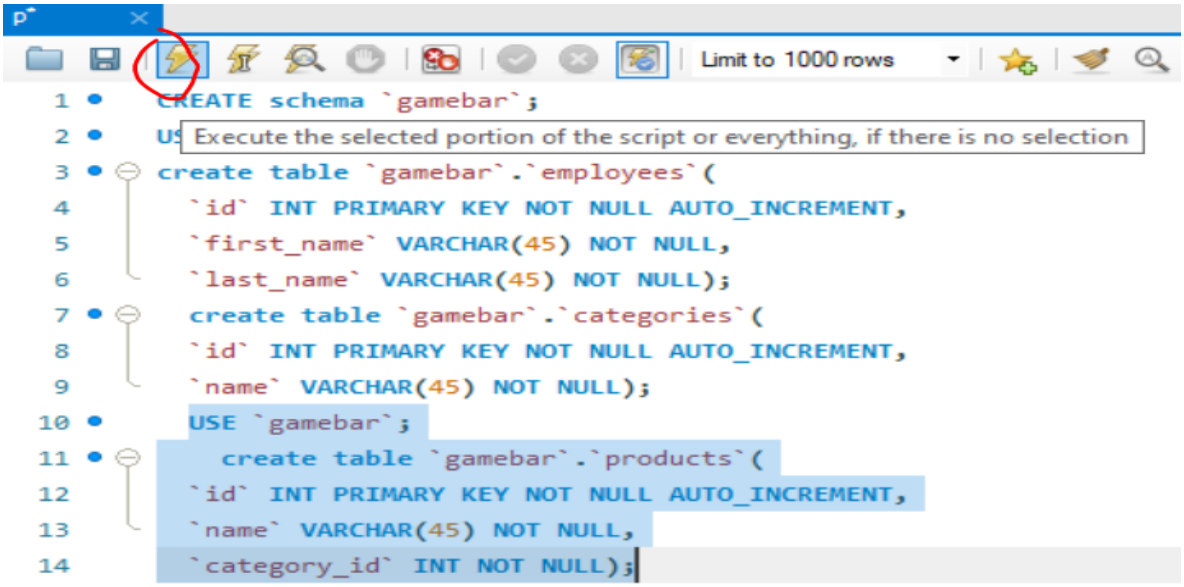
1.9.2. Where to run our SQL queries

Working with basic SQL queries

- We communicate with the database engine using SQL
- Queries provide greater **control** and **flexibility**

Queries are written in the "Query" tab.
Database creation





CREATE DATABASE `gamebar`; или **CREATE SCHEMA** `gamebar`;

Ако искаме да отворим съществуващ SQL script, то правилния начин и е цъкнем **File -> Open SQL script**

1.9.3. Table Creation in SQL:

В работната част gamebar, създай таблица employees – за графично, виж скрийншота по-горе

The command **USE** – ако имаме отворени няколко база данни, да знаем с коя работим
USE `gamebar`

CREATE TABLE `gamebar`.`employees` (
`id` **INT NOT NULL AUTO_INCREMENT**,
`name` **VARCHAR(45) NOT NULL**,
PRIMARY KEY (`id`));

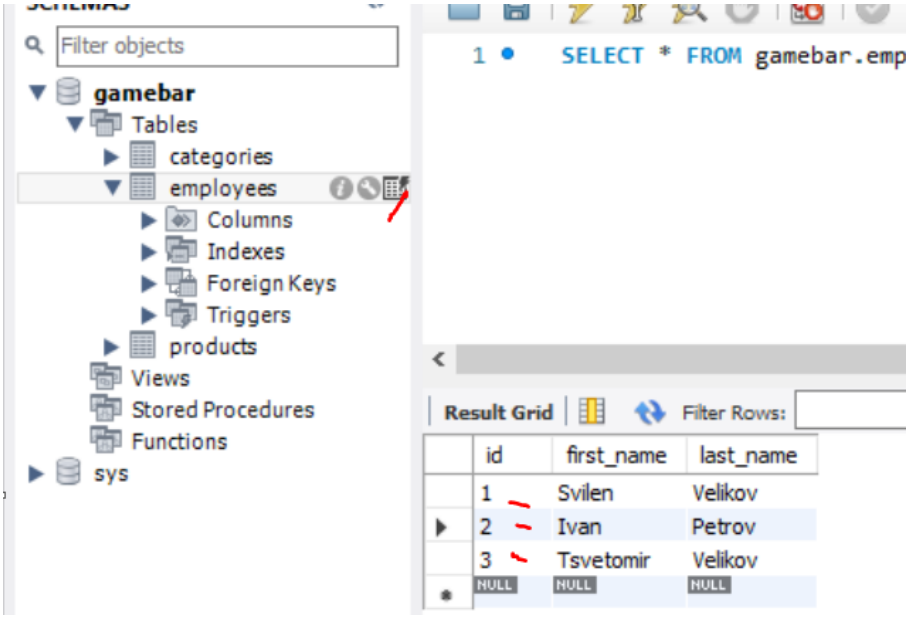
Или
create table `gamebar`.`categories`(
`id` **INT PRIMARY KEY NOT NULL AUTO_INCREMENT**,
`name` **VARCHAR(45) NOT NULL**);

Sugar syntaxis for AUTO_INCREMENT – задаване на първоначална стойност, от която да започне да инкрементира с единица:

```
CREATE TABLE models (
`model_id` INT AUTO_INCREMENT UNIQUE NOT NULL,
`name` VARCHAR(20) NOT NULL,
`manufacturer_id` INT NOT NULL
) AUTO_INCREMENT = 101;  //започни от 101 като първи запис
```

1.9.4. Add records in SQL:

Option 1 – през графични interface



Опция 2 – с SQL заявка и hardcore-нати стойности

INSERT INTO `gamebar`.`employees` (`id`, `first_name`, `last_name`) **VALUES** ('1', 'Svilen', 'Velikov');
INSERT INTO `gamebar`.`employees` (`id`, `first_name`, `last_name`) **VALUES** ('2', 'Ivan', 'Petrov');
INSERT INTO `gamebar`.`employees` (`id`, `first_name`, `last_name`) **VALUES** ('3', 'Tsvetomir', 'Velikov');

Или така:
INSERT INTO `towns` (`id`, `name`)
VALUES
(1, 'Sofia'),
(2, 'Plovdiv'),
(3, 'Varna');

Или ако вкарваме всичко:
INSERT INTO `towns` пропускаме скобите или слагаме само празни скоби
VALUES
(1, 'Sofia'),
(2, 'Plovdiv'),
(3, 'Varna');

Опция 3 – с SQL заявка, без VALUES и с функция за определяне/за автоматично попълване

Пример 1
INSERT INTO cards(card_number, card_status, bank_account_id)
(без тези скоби в judge
SELECT REVERSE(full_name), 'Active', id
FROM clients
WHERE id>=191 AND id<=200
); без тези скоби в judge

Пример 2
INSERT INTO `coaches`(`first_name`, `last_name`, `salary`, `coach_level`)

```
SELECT `first_name`, `last_name`, `salary`,
CHAR_LENGTH(`first_name`)
FROM `players`
WHERE `age` >= 45;
```

Пример 3

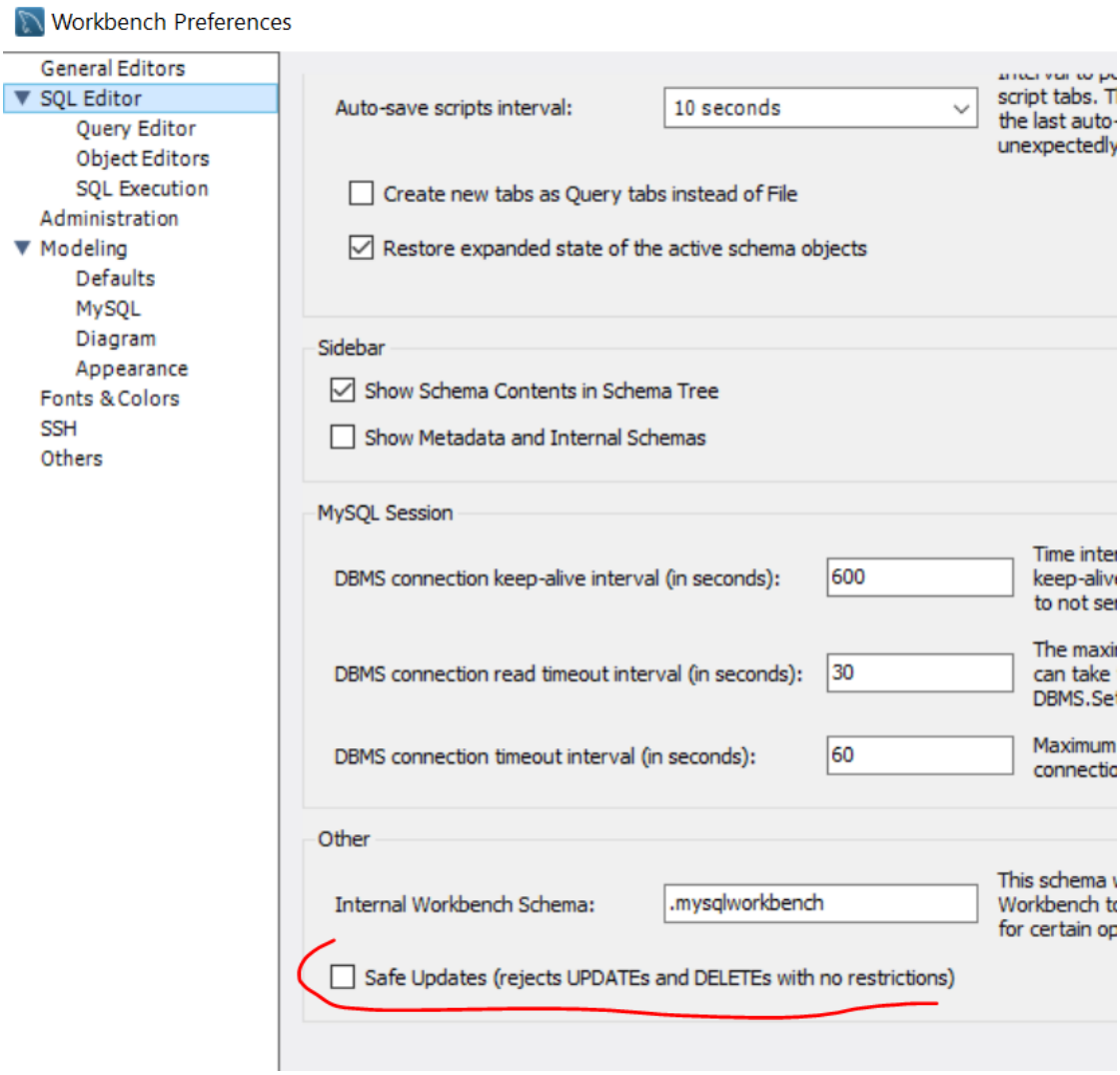
```
INSERT INTO cards(card_status, card_number, bank_account_id)
(    без тези скоби в judge
SELECT (
        CASE
                WHEN id BETWEEN 191 AND 199 THEN 'Active'
                WHEN id BETWEEN 200 AND 299 THEN 'Inactive'
                WHEN id BETWEEN 300 AND 500 THEN 'Deleted'
        END
) AS customs_status, REVERSE(full_name), id
FROM clients
); без тези скоби в judge
```

1.9.5. Correcting/Updating records in SQL

id	first_name	last_nam
1	Svilen	Velikov
2	Ivan	Ivanov
3	Tsvetomir	Velikov
NULL	NULL	NULL

```
UPDATE `gamebar`.`employees` SET `last_name` = 'Ivanov' WHERE (`id` = '2');
```

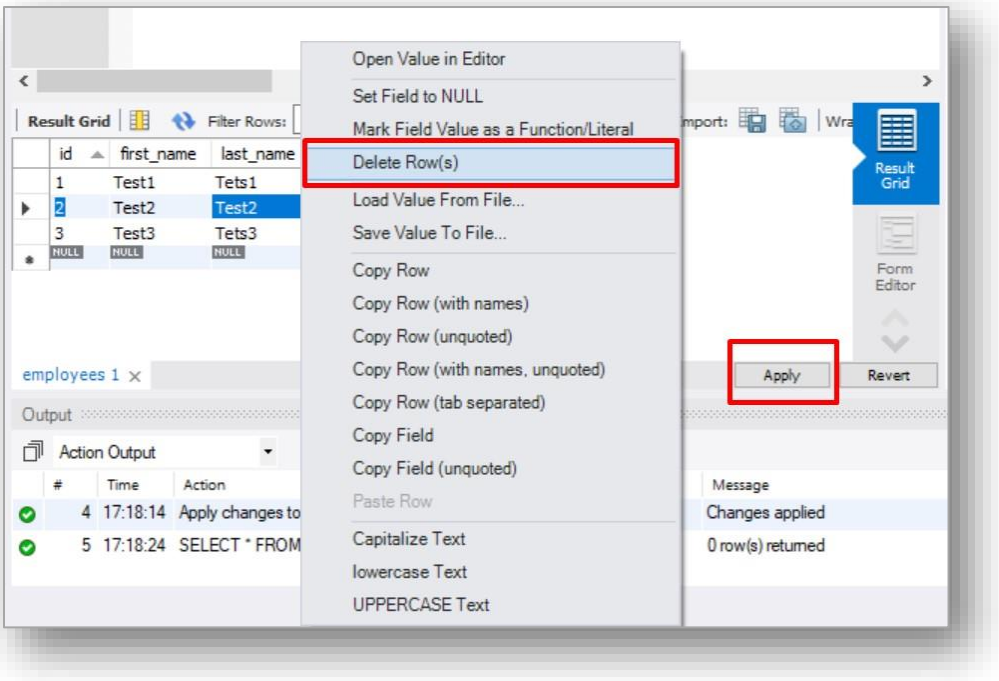
Възможност в WHERE да не участва ключовото поле: Preferences -> SQL Editor



1.9.6. Deleting data in SQL:

- Deleting structures is called **dropping**
    - You can drop **keys**, **constraints**, **tables** and entire **databases**
- Deleting all data in a table is called **truncating**
- Both of these actions **cannot be undone** – use with caution!

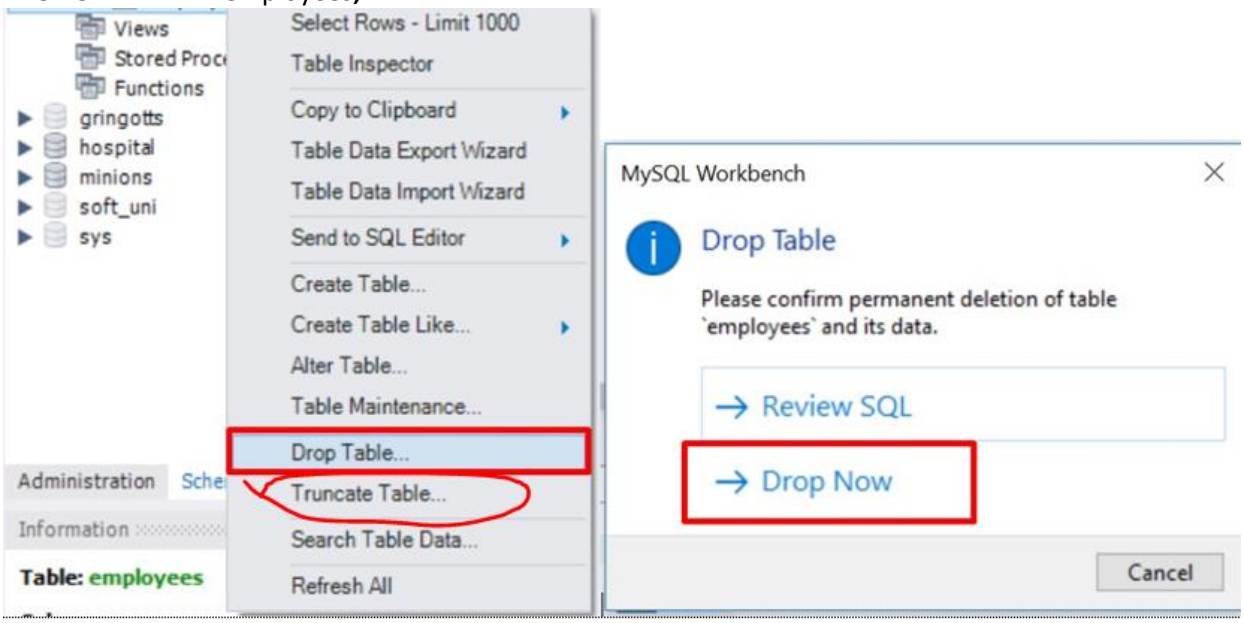
- **Deleteing row**



DELETE FROM `gamebar`.`employees` **WHERE** (`id` = '2');

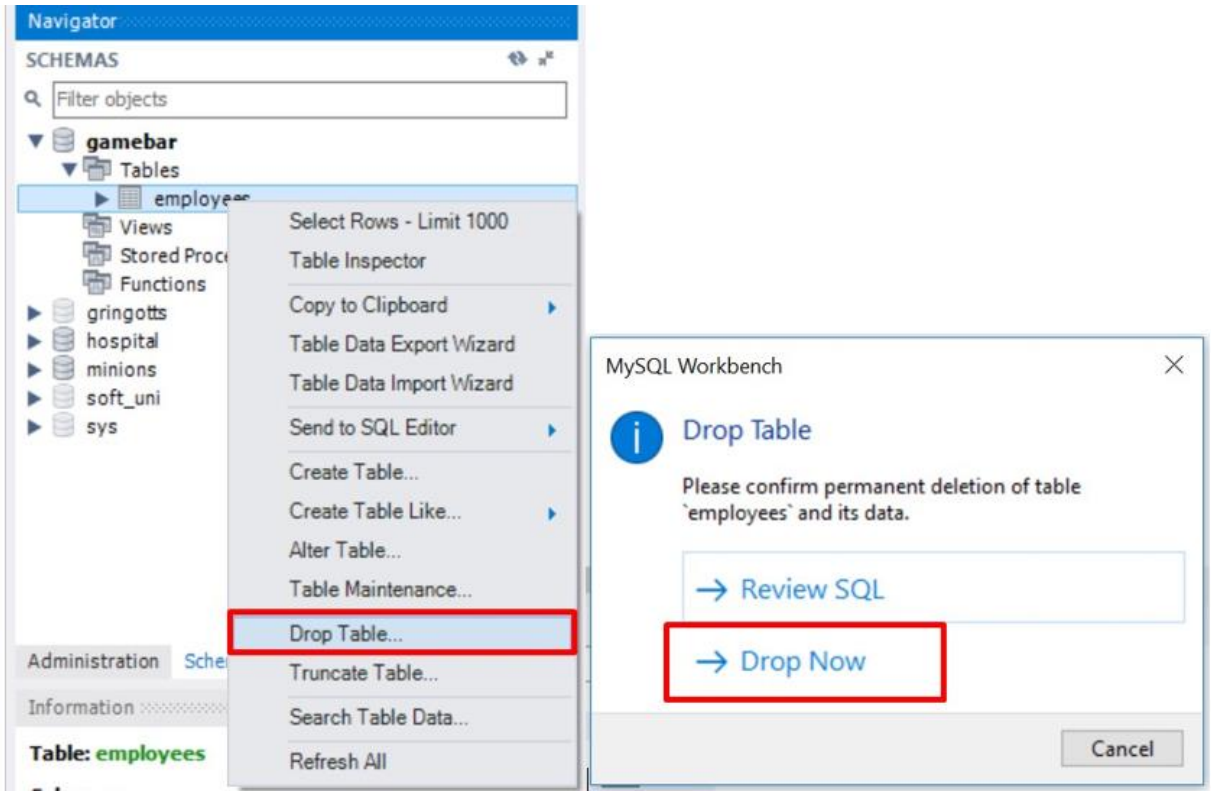
- **To delete all the entries in a table, but keep the table structure**

TRUNCATE TABLE employees**;**



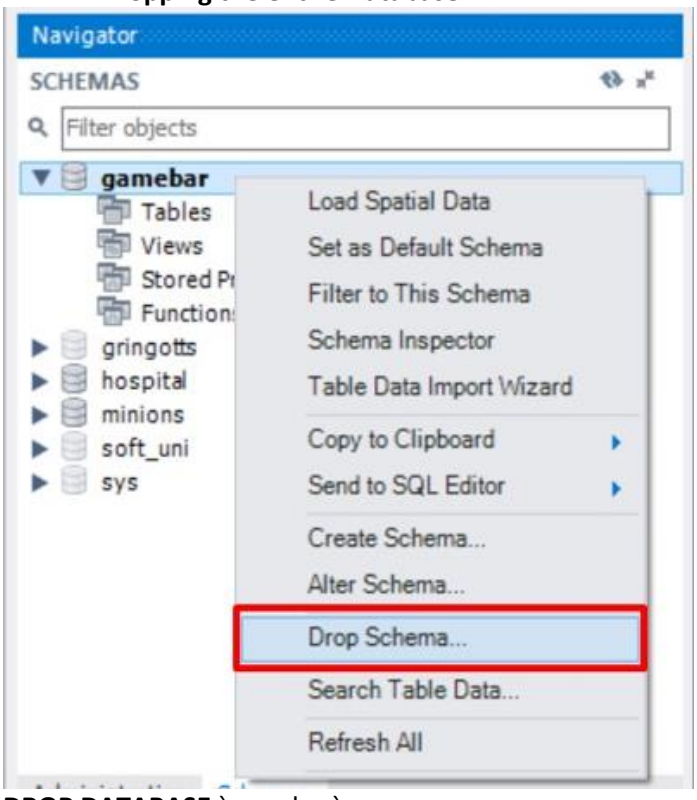
DROP Table `gamebar`.`employees`

- **Dropping table - delete data and structure**



DROP Table `gamebar`.`employees`

- **Dropping the entire Database**



DROP DATABASE `gamebar`

- **To remove a constraining rule from a column**
  - Primary keys, value constraints and unique fields

ALTER TABLE employess **DROP CONSTRAINT** pk_id**;**

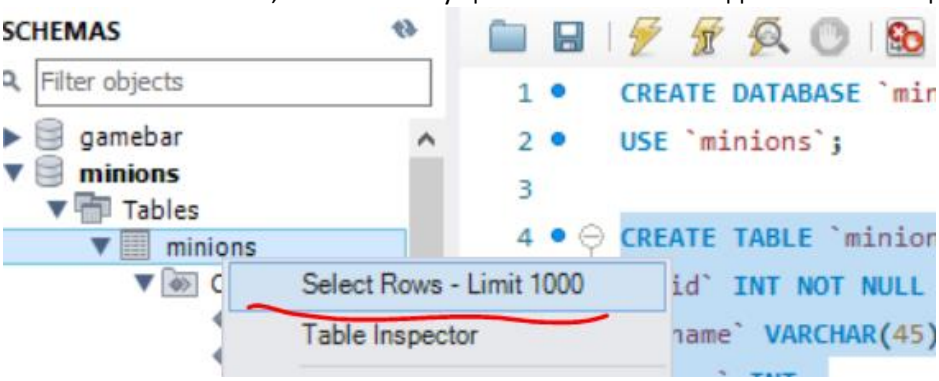
- **To remove DEFAULT value (**if not specified, revert to NULL**)**

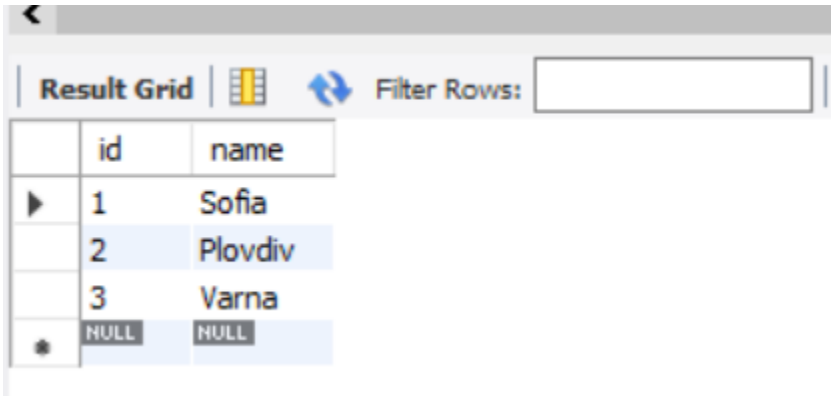
ALTER TABLE employess
ALTER COLUMN clients
DROP DEFAULT;

1.9.7. Retrieve Records in SQL:

- Get all information from a table

SELECT * FROM towns; - покажи текущите записи в базата данни от таблица towns.





- **You can limit the columns and number of records**

SELECT first_name, last_name **FROM** employees **LIMIT** 5; - ограничи до 5 записа

1.10. Table Customization

Primary Key
id INT PRIMARY KEY;

Not null – държим да има запис в това поле
id INT NOT NULL PRIMARY KEY;

Auto-Increment (Identity)
id INT AUTO_INCREMENT PRIMARY KEY;

Unique constraint – no repeating values in entire table

email VARCHAR(50) UNIQUE;

Default value – if not specified (otherwise set to **NULL**)
balance DECIMAL(10,2) DEFAULT 0;

1.11. Altering Tables

A table can be changed using the keywords ALTER TABLE

ALTER TABLE employees**;**

Add new column

ALTER TABLE employees **ADD** salary DECIMAL**; -** добавя колона salary от тип Decimal

ALTER TABLE `gamebar`.`employees`
ADD COLUMN `middle_name` **VARCHAR(45) NOT NULL** AFTER `last_name`;

ALTER TABLE `users`
ADD COLUMN `pk_users` VARCHAR(45) NOT NULL AFTER `id`;

Changing type of a column / changing name of a column

ALTER TABLE `minions`.`towns`
CHANGE COLUMN `town_id` `id` **INT NOT NULL** AUTO_INCREMENT **;**

Delete existing column – изтрива колона

ALTER TABLE people **DROP COLUMN** full_name**;**
ALTER TABLE `users` DROP COLUMN `pk_users`;

Modify data type of existing column

ALTER TABLE people **MODIFY COLUMN** email VARCHAR(100)**; -** колоната email става от нов тип

Add primary key to existing column

ALTER TABLE people **ADD CONSTRAINT PRIMARY KEY();** - Constraint name
PRIMARY KEY (id**);** **-** Column name (more than one for composite key)

ALTER TABLE `users` ADD CONSTRAINT PRIMARY KEY(`pk_users`);

Deleting primary key from a table

ALTER TABLE people
DROP PRIMARY KEY; **-** Column name (more than one for composite key)

ALTER TABLE `users`
DROP PRIMARY KEY;

Add constraint / Add unique constraint

ALTER TABLE people **ADD CONSTRAINT** uq_email - Constraint name
UNIQUE (email**)** ; - columns names

Това не копира обединени данни в колона `pk_users`, прави следното – задай ограничение pk_users, което да е primary key от id и username
ALTER TABLE `users`
DROP PRIMARY KEY,
ADD CONSTRAINT `pk_users`
PRIMARY KEY `users`(`id`, `username`);

ALTER TABLE `users`
DROP PRIMARY KEY,
ADD CONSTRAINT `pk_users`
PRIMARY KEY `users`(`id`),
CHANGE COLUMN `username` `username` VARCHAR(50) UNIQUE;

Set default value

ALTER TABLE people **ALTER COLUMN** balance **SET DEFAULT** 0;

- Set default value – вариант 2

ALTER TABLE `users`
CHANGE COLUMN `last_login_time` `last_login_time` **DATETIME NULL DEFAULT CURRENT_TIMESTAMP ;**
Старо **ново**

От типа данни на NOW()
ALTER TABLE `users`
CHANGE COLUMN `last_login_time` `last_login_time` DATETIME NULL DEFAULT NOW() ;

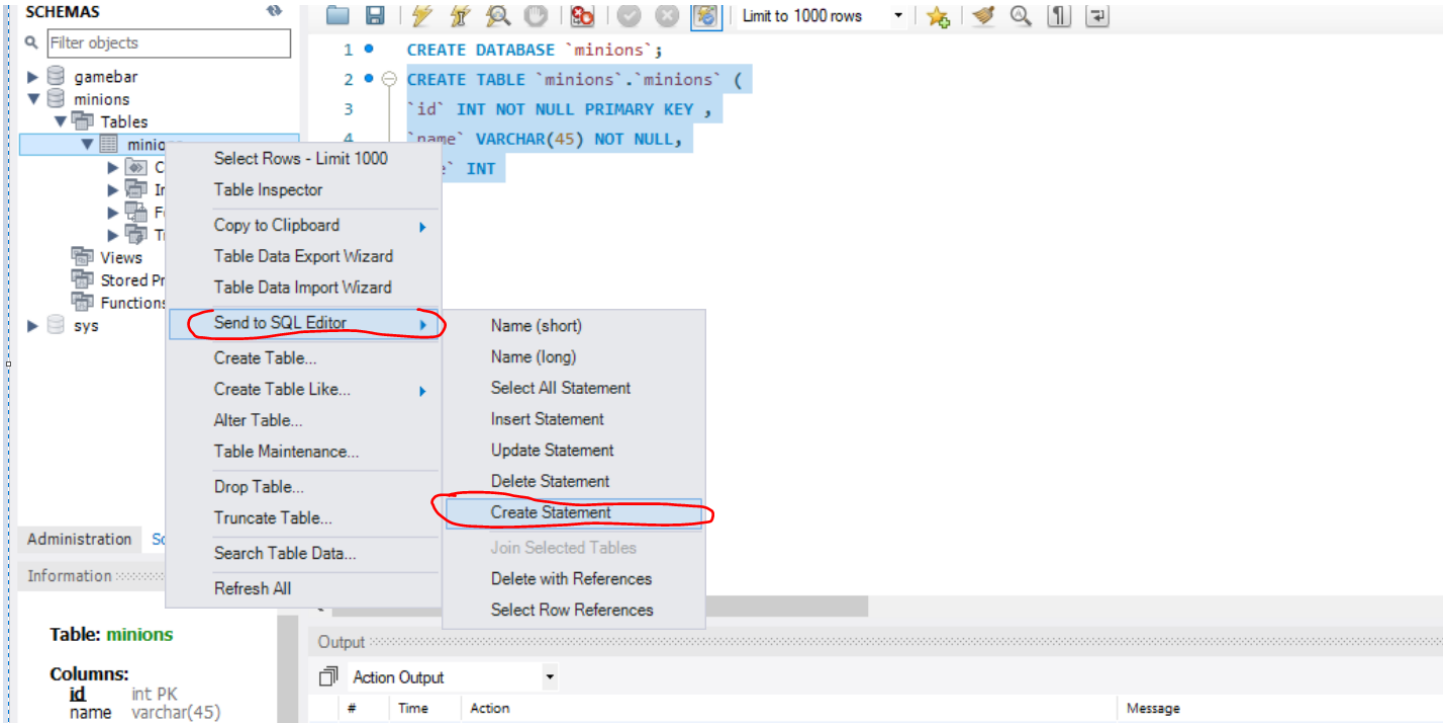
ALTER TABLE `users`
CHANGE COLUMN `username` `username` VARCHAR(30) NOT NULL DEFAULT 'Bai Peshu Starshi' ;

1.12. How to cheat – to see the SQL query

Create Statement – използваме го, за да създадем лесно SQL Заявка без да пишем всичко ръчно

Insert Statement – използваме го, за да видим SQL заявката, на това което сме създали

Update Statement – използваме го, за да обновим базата данни/да обновим поле/таблица



1.13. Advanced SQL queries

Сортира в alphabetic ред
SELECT `name` FROM `towns`
ORDER BY `name`; -

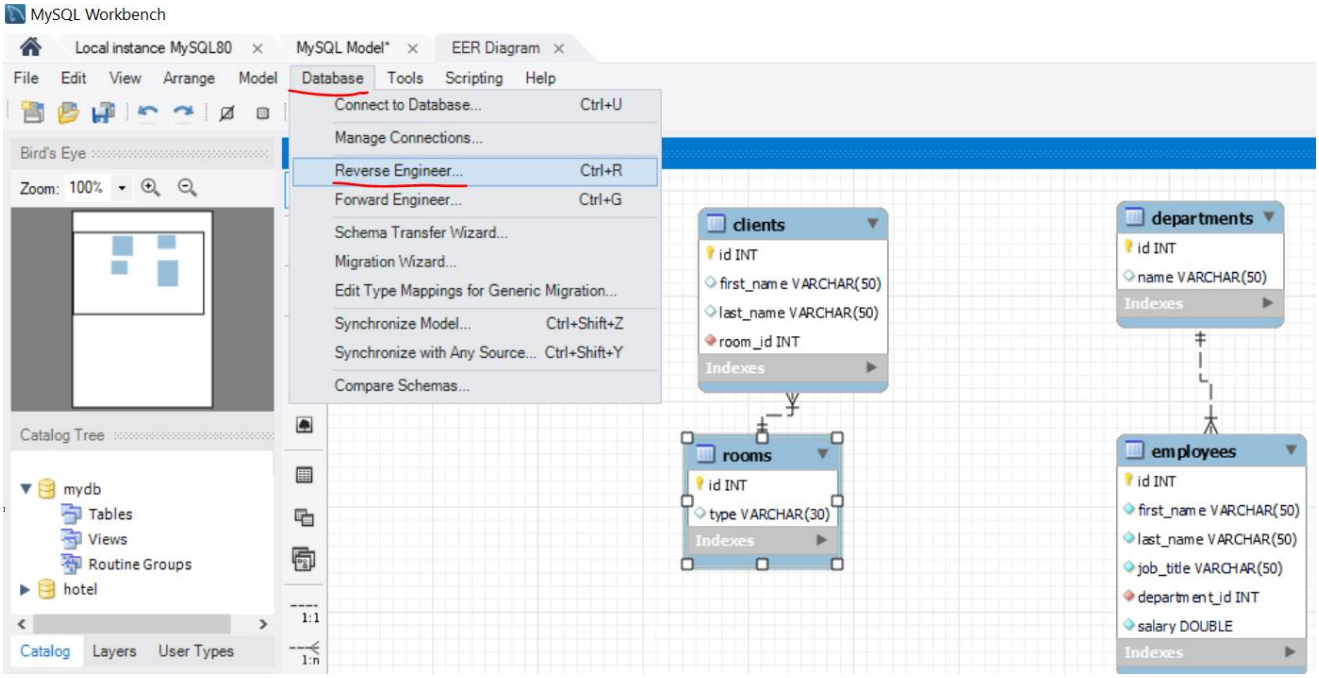
Сортира Double в низходящ ред
SELECT * **FROM** `employees`
ORDER BY `salary` **DESC;**
ORDER BY `salary` е същото като **ORDER BY** `salary` **ASC**

To show sorted only **some of the columns**
SELECT `first_name`, `last_name`, `job_title`, `salary` from `employees`
ORDER BY `salary` **DESC;**

Нанасяне на нова информация на даден ред за дадена колона – за всички записи
UPDATE `employees`
SET `salary` = `salary` * 1.1;
WHERE `id` > 0**;**

Нанасяне на нова информация на даден ред за дадена колона – за определен запис
UPDATE `employees`
SET `salary` = `salary` * 1.1;
WHERE `id` = 5**;**

1.14. E/R Diagram – диаграма на свързаността



2. BASIC CRUD (Create,Read,Update, Delete) OPERATIONS – DML – Data Manipulation Language

2.1. Query Basics

- Select first, last name and job title about employees:

```
SELECT `first_name`, `last_name`, `job_title` FROM `employees`;
```

- Select projects which start on 01-06-2003:

```
SELECT * FROM `projects` WHERE `start_date`='2003-06-01';
```

- Inserting data into table – можем да insert-нем определени колонки, но тези които изпускаме не трябва да са NOT NULL. А тези, които са AUTO_INCREMENT – сами се увеличават дори да не вкарваме данни за тях

```
INSERT INTO projects(`name`, `start_date`)
VALUES('Introduction to SQL Course', '2006-01-01');
```

Опция 3 – с SQL заявка и функция за определяне/за автоматично попълване

```
INSERT INTO cards(card_number,  card_status,  bank_account_id)
(
SELECT REVERSE(full_name), 'Active', id
FROM clients
WHERE id>=191 AND id<=200
);
```

- Update several cells for specific rows:

```
UPDATE `projects`
SET `end_date` = '2006-08-31', `id` = 3;
WHERE `start_date` = '2006-01-01';
```

- Update specific cells/columns for all rows/records:

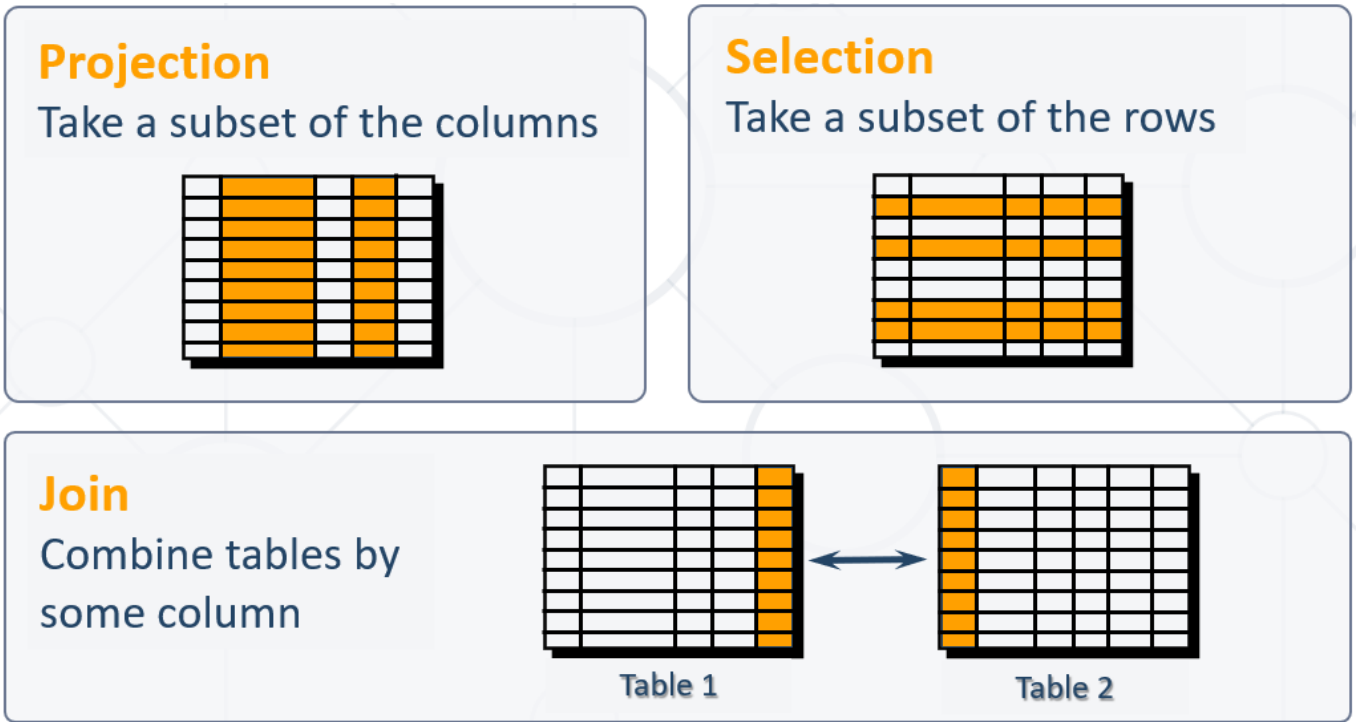
```
UPDATE `employees`
SET `salary` = `salary` * 1.1;
```

- Delete specific projects – изтрива целия ред

```
DELETE FROM `projects`
WHERE `start_date` = '2006-01-01';
```

2.2. Retrieving Data

Capabilities of SQL SELECT




```
SELECT  `id`, `first_name`, `last_name`, `job_title`
FROM `employees`
ORDER BY `id`;\
WHERE
LIMIT 3;
```

Всички колони плюс още колони

```
SELECT  *,`id`, `first_name`, `last_name`, `job_title`
FROM `employees`
ORDER BY `id`;\
WHERE
```

LIMIT 3;

Aliases(прякор/друго име) rename a table or a column heading – използваме задължително когато работим с повече от една таблица!!!

ВАЖНО – когато искаме да работим с колоната aliases – прякото име, то след AS използваме обикновени кавички '', но в последствие използваме специалните кавички `` - НЕЕЕ Е ТАКА, МОЖЕ ДА СИ ИЗПОЛЗВАМЕ САМО ТИЛДА КАВИЧКИ!

```
SELECT e.id AS 'No.',
e.first_name AS 'First Name',
e.last_name AS 'Last Name',
e.job_title AS 'Job Title'
FROM employees AS e
ORDER BY `Job title`;
```

Пример за Aliases когато работим едновременно с 2 таблици:

```
SELECT p.`peak_name`,
r.`river_name`,
LOWER(CONCAT(p.`peak_name`, SUBSTRING(r.`river_name`, 2))) AS `mix`
FROM `peaks` AS p, `rivers` AS r
WHERE RIGHT(LOWER(p.`peak_name`), 1) = LEFT(LOWER(r.`river_name`), 1)
ORDER BY `mix`;
```

```
SELECT
5+5 AS 'staticnumber',
`job_title` AS  'Job Title',
`id` AS 'No.'
FROM `employees`;
```

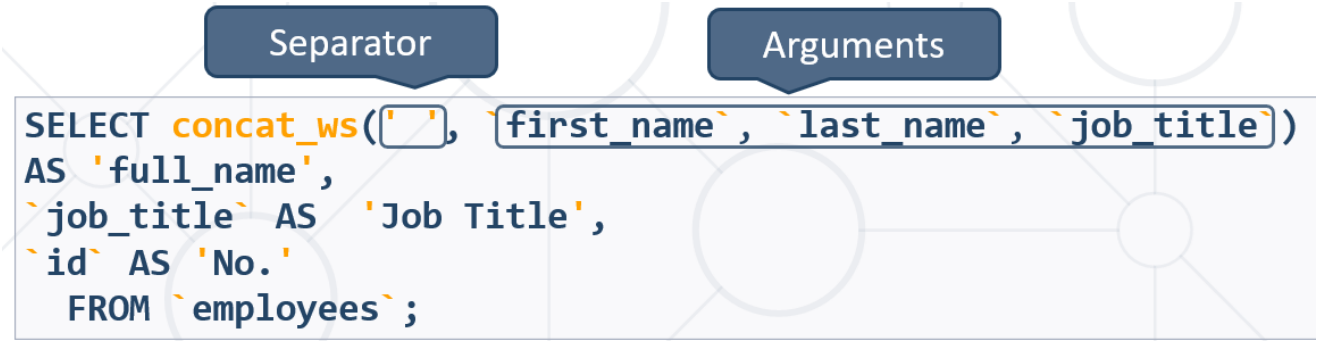
Concatenation – когато ги обединява в резултата от SELECT

concat() - returns the string that results from concatenating the arguments - предефинирана функция в MySQL

- String literals are enclosed in ['](**single quotes**)
- Table and column names containing special symbols use [`] (**backtick**)

```
SELECT concat(`first_name`,' ',`last_name`) AS 'full_name',
`job_title` AS  'Job Title',
`id` AS 'No.'
FROM `employees`;
```

*Another function of concatenation is **concat_ws()** - stands for concatenate with **separator** and is a special form of **CONCAT()** – с първия стринг лепим останалите*



- Skip any **NULL** values after the separator argument.

Concatennating with + : Add 2 strings together:

Не работи както трябва

Filtering the Selected Rows

- Use **DISTINCT** to eliminate duplicate results – to eliminate all the duplicate records and fetching only unique records.

```
SELECT DISTINCT `first_name`
FROM `employees`;
```

Ако има повтарящи се имена, то покажи само веднъж повтарящото се име

- You can filter rows by specific conditions using the **WHERE** clause

```
SELECT `last_name`, `department_id`
FROM `employees`
WHERE `department_id` = 1;
```

- Other **logical operators** can be used for better control

```
SELECT `last_name`, `salary`
FROM `employees`
WHERE `salary` <= 20000;
```

- Conditions can be combined using **NOT**, **OR**, **AND** and brackets

```
SELECT `last_name` FROM `employees`
WHERE NOT (`manager_id` = 3 OR `manager_id` = 4);
```

- Using **BETWEEN** operator to specify a range:

```
SELECT `last_name`, `salary` FROM `employees`
WHERE `salary` BETWEEN 20000 AND 22000; - работи включително
HAVING `max_salary` NOT BETWEEN 30000 AND 70000
```

- Using **IN / NOT IN** to specify a set of values:

```
SELECT `first_name`, `last_name`, `manager_id`
FROM `employees`
WHERE `manager_id` IN (109, 3, 16); - дали е измежду тези стойности
```

```
SELECT * FROM `towns`
WHERE LOWER(SUBSTRING(`name`,1,1)) NOT IN('r', 'b', 'd')
```

ORDER BY `name`;

Comparing with NULL

- **NULL** is a special value that means missing value
  - Not the same as **0** or a blank space
- Checking for **NULL** values

Проверка за различно по този начин **!=** и по този начин **<>** не можем да правим с NULL!!!

```
SELECT `last_name`, `manager_id`
FROM `employees`
WHERE `manager_id` = NULL; -ГРЕШНО!
```

```
SELECT `last_name`, `manager_id`
FROM `employees`
WHERE `manager_id` IS NULL;
```

```
SELECT `last_name`, `manager_id`
FROM `employees`
WHERE `manager_id` IS NOT NULL;
```

Sorting with ORDER BY – може по две условия, отделяме със запетая

- Sort rows with the **ORDER BY** clause
  - **ASC**: ascending order, default
  - **DESC**: descending order

```
SELECT `last_name`, `hire_date`
FROM `employees`
ORDER BY `hire_date` DESC, `last_name` ASC;
```

Ако `hire_date` съвпада, то сортирай по следващ критерий `last_name`

Сортираме по много условия – отделяме със запетая

```
SELECT * FROM `employees`
ORDER BY `salary` DESC, `first_name` ASC, `last_name` DESC, `middle_name` ASC, `employee_id`;
```

Views – все едно си запазваме предефинирана заявка

- Views are **virtual tables** made from others tables, views or joins between them
- Usage:
  - To simplify writing complex queries
  - To limit access to data for certain users

Views – Example 1

```
CREATE VIEW `v_hr_result_set` AS
SELECT
        CONCAT(`first_name`,' ',`last_name`) AS 'Full Name',
        `salary`
FROM `employees` ORDER BY `department_id`;

SELECT * FROM `v_hr_result_set`;
```

Example 2

```
CREATE VIEW `myview` AS
```

SELECT `first_name`, 5 **FROM** `employees`
ORDER BY `salary` **DESC**, `first_name` **ASC**, `last_name` **DESC**;

SELECT * FROM `myview`;
DROP VIEW `myview`; - заличи

2.3. Writing Data in Tables

*The SQL **INSERT** command*

INSERT INTO `towns` **VALUES** (33, 'Paris'); **- values for all columns сме длъжни да подадем**

INSERT INTO projects(`name`, `start_date`) - кажи колко колони ще нанасяш
VALUES ('Reflective Jacket', NOW())

Inserting data into table – можем да insert-нем определени колонки, но тези които изпускаме не трябва да са NOT NULL. А тези, които са AUTO_INCREMENT – сами се увеличават дори да не вкарваме данни за тях

Bulk data can be recorded in a single query, separated by comma

```
INSERT INTO `towns` (`id`, `name`)
VALUES
(1, 'Sofia'),
(2, 'Plovdiv'),
(3, 'Varna');
```

*You can use existing records to create a **new table – копира както структурата, така и данните***

Пример 1

CREATE TABLE `customer_contacts` - new table name
AS SELECT `customer_id`, `first_name`, `email`, `phone`
FROM `customers`; - from existing table

Пример 2

CREATE TABLE `workers` **AS**
SELECT `first_name` **FROM** `employees`;

CREATE TABLE auto_filled **AS**
SELECT e.`first_name`,
d.`name` AS 'dept_name'
FROM `employees` AS e
INNER JOIN `departments` AS d
ON e.`department_id` = d.`department_id`;

You can write into an existing table - – копира както структура, така и данни

Пример 1

INSERT INTO `projects`(`name`, `start_date`)
SELECT
CONCAT(`name`,' ', ' Restructuring'),
NOW()
FROM `departments`;

Копира ги/добвя ги като данни за нови елементи от таблицата .... ☹
INSERT INTO `users`(`pk_users`)

```
SELECT
CONCAT(`id`, '',`username`) FROM `users`;
```

Пример 2

```
CREATE TABLE `workers`;

INSERT INTO `workers`
SELECT `first_name` FROM `employees` WHERE `salary` < 1000;
```

2.4. Updating Existing Records – UPDATE & DELETE

Updating data

The SQL UPDATE command

```
UPDATE `employees`
SET `last_name` = 'Brown'
WHERE `employee_id` = 1;

UPDATE `employees`
  SET `salary` = `salary` * 1.10,
    `job_title` = CONCAT('Senior',' ', `job_title`)
 WHERE `department_id` = 3;
```

- Note: Don't forget the **WHERE** clause!

Deleting Data

- Deleting specific rows from a table
  - Note: Don't forget the **WHERE** clause!

```
DELETE FROM `employees`
WHERE `employee_id` = 1;
```

- Delete all rows from a table (**TRUNCATE** works faster than **DELETE)**

```
TRUNCATE TABLE `users`;
```

3.Built-in functions

https://dev.mysql.com/doc/refman/8.0/en/functions.html - функции, има и .xml functions и .json functions

3.1. String functions

- **SUBSTRING()** – extracts part of a string

SUBSTRING(*String*, *Position***) – позицията/броенето започва от 1, а не от 0-левия**
SUBSTRING(*String*, *Position*, *Length***)**
SUBSTRING(*String* FROM *Position* FOR *Length***)**

```
SELECT SUBSTRING('SoftUni', 2);  - връща  'oftUni'
SELECT SUBSTRING('SoftUni', 2, 3); - връща 'oft'
```

- **REPLACE** – replaces specific string with another
  - Performs a case-sensitive match

REPLACE(*String*, *Pattern*, *Replacement***)** Pattern - string to replace replacement – with what to replace

```
SELECT REPLACE(`title`, 'The', '***')
AS 'Title' FROM `books`
WHERE SUBSTRING(title, 1, 3) = 'The';
```

```
SELECT REPLACE(`title`, 'The', '***')
AS `title` FROM `books`
WHERE `title` LIKE 'The%'
ORDER BY `id` ASC;
```

```
SELECT `first_name`, `last_name` FROM `employees`
WHERE LOWER(`job_title`) NOT LIKE '%engineer%'
ORDER BY `employee_id`;
```

- **Кастване/конвертиране от число към стринг**

CAST (1 as **CHAR**)

- **Chaining на функции –** една функция в друга

- **LTRIM & RTRIM –** remove spaces from either side of string

LTRIM(*String***) –** от началото на стринга
RTRIM(*String***)** – от края на стринга

- **CHAR_LENGTH –** count number of characters

CHAR_LENGTH(*String***)**

```
SELECT `name` FROM `towns`
WHERE CHAR_LENGTH(`name`) IN(5,6)
ORDER BY `name` ASC;
```

- **LENGHT –** get number of used bytes (double for Unicode)

LENGTH(String**)**
Кирилицата заема по 2 байта, а латиницата по един

```
SELECT LENGTH('асц'); - връща 6
SELECT LENGTH('bdt'); - връща 3
```

- **LEFT & RIGHT** – get characters from beginning or end of string

LEFT(*String*, *Count***) – от края**
RIGHT(*String*, *Count***) – от началото**

```
SELECT `id`, `start`,
LEFT(`name`, 3) AS 'Shorthand'
FROM `games`;
```

- **LOWER & UPPER –** change letter casing – we use it for case insensitive search

LOWER(*String***)**
UPPER(*String***)**

- **REVERSE** – reverse order of all characters in string

REVERSE(*String***)**

- **REPEAT –** repeat string

REPEAT(*String, Count***)**

- **LOCATE –** locate specific pattern (substring) in string

LOCATE(*Pattern, String,[Position]***) -** If omitted(пропуснато), position begins at 1
SELECT **LOCATE**('Big', `title`) FROM `books`; - връща позицията на която се появява Big в полето `title`. **Ако не намери, връща 0**
SELECT LOCATE('@', 'chavdar.mitkov@softuni.bg'); - връща 15
SELECT LOCATE('@', 'chavdar.mitkov@softu@ni.bg', 16); - връща 22

SELECT `user_name`, SUBSTRING(`email`, LOCATE('@', `email`)+1) AS `email provider` FROM `users`
ORDER BY `email provider` ASC, `user_name` ASC;

- **INSERT** – insert substring at specific position

INSERT(*StringToInsertInto, Position, Length, Substring***)** като Length е броят символи за унищожение

SELECT INSERT(`title`, 1, 0, 'Ordered book: ') FROM `books`; **- вмъкни на позиция 1, без да триеш нищо(0) 'Ordedered book: ' пред всяко заглавие на книга**

SELECT *,
INSERT (`title`, LOCATE('Big', `title`), 3, 'Small') AS `newtitle` - новото заглавие ако съдържа Big, то го подмени с 'Small'
FROM `books`
WHERE `title` LIKE **'%Big%'; - да не започва или да не завършва с Big**

- **SUBSTRING_INDEX** – insert substring at specific position

SUBSTRING_INDEX(*string, delimiter, number*)

Parameter	Description
string	Required. The original string
delimiter	Required. The delimiter to search for
number	Required. The number of times to search for the delimiter. Can be both a positive or negative number. If it is a positive number, this function returns all to the left of the delimiter. **If it is a negative number, this function returns all to the right of the delimiter.**

```
SELECT
  user_name,
  SUBSTRING_INDEX(email, '@', -1) AS 'email provider'
FROM
  users;
```

3.2. Arithmetical Operators and Numeric Functions

Arithmetical Operators

Name	Description
DIV	Integer division
/	Division operator
-	Minus Operator
%, MOD	Modulo operator
+	Addition operator
*	Multiplication operator
- (arg)	Change sign of argument

Numeric Functions

Used primarily for numeric **manipulation** and/or mathematical **calculations**

- **PI** – get the value of Pi (15 –digit precision)

SELECT PI() +0.000000000000000

- **ABS** – absolute value

ABS(*Value***)**

- **SQRT** – square root

SQRT(*Value***)**

- **POW** – raise value to desired exponent

POW(*Value*, *Exponent***)**

- The **SUM()** function returns the total sum of a numeric column.

SELECT ROUND(SUM(`cost`), 2) AS `total_sum` FROM `books`;

Math Functions

- **CONV** – Converts numbers between different number bases

CONV(*Value*, *from_base, to_base***)**

- **ROUND** – obtain desired precision

ROUND(*Value*, *Precision***) –** Precision can be negative

SELECT **ROUND**(PI(), 2);

- **FLOOR** & **CEILING** – return the nearest integer

FLOOR(*Value***) - надолу**

CEILING(*Value***) - нагоре**

- **SIGN** – returns +1, -1 or 0, depending on value sign

SIGN(*Value***)**

- **RAND** – get a random value in range [0,1)
  - If **Seed** is not specified, one is assigned at random – за хеширане, връща винаги една стойност за даден Seed

RAND() – връща random
RAND(*Seed***) -** за хеширане, връща винаги една стойност за даден Seed

SELECT RAND('ssdfe'); винаги връща '0.15522042769493574'

3.3. Date Functions

- **EXTRACT** – extract a segment from a date as an integer

EXTRACT(*Part* ***FROM*** *Date***)**
SELECT
EXTRACT(**DAY** FROM `born`) AS `day`,
`born` FROM `authors`;

SELECT EXTRACT(**YEAR** FROM '2022-05-23'); 2022
SELECT EXTRACT(**month** FROM '2022-05-23'); 5
SELECT EXTRACT(**day** FROM '2022-05-23'); 23

- **Get direct DAY, YEAR, MONTH, etc. without the function extract**

Директно си подаваме DAY, YEAR, MONTH от дадена дата
DAY(p.`date`) = 10

- **TIMESTAMPDIFF** – find difference between two dates

TIMESTAMPDIFF(*Part, FirstDate, SecondDate***)**

- ***Part*** can be any part and format of date or time

SELECT
timestampdiff(**MONTH**, `born`, `died`) AS `months_lived`,
`born` FROM `authors`;

- **DATE_FORMAT** – formats the date value according to the format

SELECT DATE_FORMAT('2017/05/31', '%Y %b %D') **AS** 'Date'**;**
SELECT DATE_FORMAT('2017/05/31 23:13:00', '%Y %b %D, %h:%i:%s') AS 'Date';
2017 May 31st, 11:13:00

Хардкорнати стойности %b за месец и %i за минута

SELECT `first_name` FROM `employees`

WHERE `department_id` IN(3, 10) && DATE_FORMAT(`hire_date`, '%Y') BETWEEN 1995 AND 2005
ORDER BY `employee_id`;

Specifier	Description
%a	Abbreviated weekday name (Sun..Sat)
%b	Abbreviated month name (Jan..Dec)
%c	Month, numeric (0..12)
%D	Day of the month with English suffix (0th, 1st, 2nd, 3rd, ...)
%d	Day of the month, numeric (00..31)
%e	Day of the month, numeric (0..31)
%f	Microseconds (000000..999999)
%H	Hour (00..23)
%h	Hour (01..12)
%I	Hour (01..12)
%i	Minutes, numeric (00..59)
%j	Day of year (001..366)
%k	Hour (0..23)
%l	Hour (1..12)
%M	Month name (January..December)
%m	Month, numeric (00..12)
%p	AM or PM
%r	Time, 12-hour (***hh:mm:ss*** followed by AM or PM)
%S	Seconds (00..59)
%s	Seconds (00..59)
%T	Time, 24-hour (***hh:mm:ss***)
%U	Week (00..53), where Sunday is the first day of the week; WEEK() mode 0
%u	Week (00..53), where Monday is the first day of the week; WEEK() mode 1
%V	Week (01..53), where Sunday is the first day of the week; WEEK() mode 2; used with %X
%v	Week (01..53), where Monday is the first day of the week; WEEK() mode 3; used with %x
%W	Weekday name (Sunday..Saturday)
%w	Day of the week (0=Sunday..6=Saturday)
%X	Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V
%x	Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v
%Y	Year, numeric, four digits
%y	Year, numeric (two digits)
%%	A literal % character
%***x***	***x***, for any "***x***" not listed above

- **NOW** – obtain current date and time

SELECT NOW();

- **DATE_ADD**(`some_date`, **INTERVAL** *stepValue typeStep*)

SELECT `product_name`, `order_date`,
DATE_ADD(`order_date`, **INTERVAL** 3 **DAY**) AS 'pay_due'
FROM `orders`;

MINUTE
HOUR
DAY
WEEK
MONTH
QUARTER
YEAR

- Сравнение по дата

SELECT `deposit_group`, `is_deposit_expired`,
AVG(`deposit_interest`) AS `average_interest`
FROM `wizzard_deposits`
WHERE `deposit_start_date` > '1985-01-01' когато формата на датата е този
GROUP BY `deposit_group`, `is_deposit_expired`
ORDER BY `deposit_group` DESC, `is_deposit_expired` ASC;

https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html

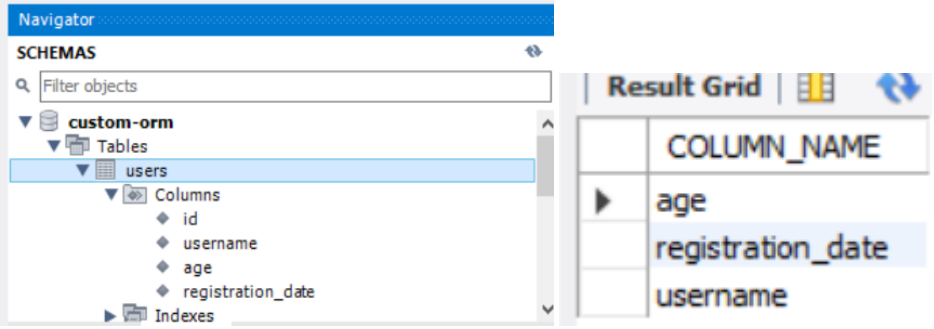
3.4. Wildcards

Системни команди / System commands – като сървър (а не като клиент)

USE **INFORMATION_SCHEMA**;
SELECT * from **statistics**;
SHOW **tables**;

Вземи information данни за дадена таблица

SELECT `COLUMN_NAME` **FROM** `information_schema`.`columns`
WHERE `TABLE_SCHEMA` = 'custom-orm' **AND** `COLUMN_NAME` != 'id' **AND** `TABLE_NAME` = 'users';



Used to substitute any other character(s) in a string

- **'%'** - represents zero, one, or multiple characters

    - '_' - represents a single character
    - Can be used in combinations

- Used with **LIKE** operator in a **WHERE** clause
    - Similar to **Regular Expressions**

```
SELECT * FROM `books`
WHERE `title` LIKE '____Big%';
```

```
SELECT `user_name`, `ip_address` FROM `users`
WHERE `ip_address` LIKE '___.1%.%.___'     три символа.1няколко символа.няколко символа.три символа
ORDER BY `user_name` ASC;
```

- Find any values that start with "a"

```
WHERE CustomerName LIKE 'a%';
```

- Find any values that have "r" in second position

```
WHERE CustomerName LIKE '_r%';
```

- Finds any values that starts with "a" and ends with "o"

```
WHERE ContactName LIKE 'a%o';
```

- Supported characters also include – **част от regex нещата искат да опишат тук**
    - **** – specify prefix to treat special characters as normal – да го escape-нем
    - [**charlist**] – specifying which characters to look for
        - [!**charlist**] – **excluding** characters

```
SELECT * FROM `books`
WHERE `title` LIKE '____Big\%'; - обикновен процент
```

```
SELECT `first_name`, `last_name` FROM `employees`
WHERE LOWER(`job_title`) NOT LIKE '%engineer%'
ORDER BY `employee_id`;
```

3.5. Regex

```
SELECT * FROM `customers`
WHERE `city` REGEXP '[a-c]%';    -  a, b or c  - връща 0 или 1 ца – дали има или няма match
```

Using regular expression

- **REGEXP** - pattern matching using regular expressions

```
SELECT `employee_id`, `first_name`, `last_name`
FROM `employees`
WHERE `first_name` REGEXP '^\[^K\]{3}\$';  връща 0 или 1 ца – дали има или няма match
```

Пример:

```
DELIMITER $$$$
CREATE FUNCTION ufn_is_word_comprised(setOfLetters VARCHAR(45), word VARCHAR(45))
RETURNS BIT   //връща нула или единица, както и при BOOLEAN
DETERMINISTIC
BEGIN
```

```
        RETURN word REGEXP(concat('^[', setOfLetters, ']+$')); //поне един път трябва да има всяк
END;
$$$$

SELECT ufn_is_word_comprised('oistmiahf', 'Sofia');
SELECT ufn_is_word_comprised('oistmiahf', 'halves');
```

- **REGEXP_SUBSTR**(*expr*, *pat*[, *pos*[, *occurrence*[, *match_type*]]])

Returns the substring of the string *expr* that matches the regular expression specified by the pattern *pat*, NULL if there is no match. If *expr* or *pat* is NULL, the return value is NULL. REGEXP_SUBSTR() takes these optional arguments:

  - *pos*: The position in *expr* at which to start the search. If omitted, the default is 1.
  - *occurrence*: Which occurrence of a match to search for. If omitted, the default is 1.
  - *match_type*: A string that specifies how to perform matching. The meaning is as described for REGEXP_LIKE().

```
SELECT REGEXP_SUBSTR(`title`, ' [a-zA-Z]{2} ') AS `match`, `title` FROM `books`;

SELECT REGEXP_SUBSTR(`title`, '[a-zA-Z]+') AS `match`, `title` FROM `books`;
```

- REGEXP_REPLACE(*expr*, *pat*, *repl*[, *pos*[, *occurrence*[, *match_type*]]])

Replaces occurrences in the string *expr* that match the regular expression specified by the pattern *pat* with the replacement string *repl*, and returns the resulting string. If *expr*, *pat*, or *repl* is NULL, the return value is NULL.

```
SELECT
  user_name,
  REGEXP_REPLACE(email, '.*@', '') AS 'email provider'
FROM
  users;
```

3.6. Условни конструкции

Използване на IFNULL функцията - Return the specified value IF the expression is NULL, otherwise return the expression:

IFNULL(`middle_name`), '') - ако е NULL, то го замести с празен стринг, иначе върни полето

IF condition – 1 – as a function – връща резултат

IF(*condition, value_if_true, value_if_false*)

```
SELECT
`name` AS `game`,
/*DATE_FORMAT(start, '%k') AS `P`,*/
IF(DATE_FORMAT(start, '%k') >= 0 && DATE_FORMAT(start, '%k') < 12, 'Morning',
        IF(DATE_FORMAT(start, '%k') >= 12 && DATE_FORMAT(start, '%k') < 18, 'Afternoon', 'Evening')) AS `Part of the
Day`,
IF(`duration` <= 3 , 'Extra Short',
```

```
        IF(`duration` <= 6, 'Short',
                IF(`duration` <= 10, 'Long', 'Extra Long'))) AS `Duration`
FROM `games`;
```

IF condition – 2 – as a statement – не връща стойност

```
IF search_condition THEN statement_list;
    [ELSEIF search_condition THEN statement_list] ...;
    [ELSE statement_list];
END IF
```

```
DELIMITER %%
CREATE PROCEDURE usp_raise_salary_by_id(id int)
BEGIN
        START TRANSACTION;
        IF((SELECT count(employee_id) FROM employees WHERE employee_id like id)<>1)
        THEN ROLLBACK;
        ELSE
                UPDATE employees AS e SET salary = salary + salary*0.05
                WHERE e.employee_id = id;
        END IF;
END %%
```

```
DECLARE result DECIMAL;
IF(salary_emp < 30000) THEN SET result := 'Low'; //при DECLARE използваме := за присвояване
   ELSEIF (salary_emp <= 50000) THEN SET result := 'Average';  ELSEIF слято трябва да е
   ELSE SET result := 'High';
END IF;
```

CASE condition - as a function – връща резултат

Пример 1:

```
SELECT
CASE `author_id`
        WHEN 1 THEN 'Recommended for beginners'   //Ако 1, върни еди какво си
        WHEN 7 THEN 'Recommended for advanced'
        ELSE 'All audiences'
END
AS `my_preference`
FROM `books`
WHERE SUBSTRING(title, 1, 3) = 'The';
```

Пример 2:

```
SELECT `name` AS 'game',
(CASE /*без променлива тук може*/
        WHEN HOUR(`start`) BETWEEN 0 AND 11 THEN 'Morning'
        WHEN HOUR(`start`) BETWEEN 12 AND 17 THEN 'Afternoon'
   ELSE 'Evening'
END)   AS   'Part of the Day',
(CASE /*без променлива тук може*/
        WHEN `duration` BETWEEN 0 AND 3 THEN 'Extra Short'
```

```
        WHEN `duration` BETWEEN 4 AND 6 THEN 'Short'
        WHEN `duration` BETWEEN 7 AND 10 THEN 'Long'
   ELSE 'Extra long'
END)   AS  'Duration'
FROM `games`;
```

Пример 3:

```
SELECT
CASE
        WHEN `age` BETWEEN 0 AND 10 THEN '[0-10]'
        WHEN `age` BETWEEN 11 AND 20 THEN '[11-20]'
        WHEN `age` BETWEEN 21 AND 30 THEN '[21-30]'
        WHEN `age` BETWEEN 31 AND 40 THEN '[31-40]'
        WHEN `age` BETWEEN 41 AND 50 THEN '[41-50]'
        WHEN `age` BETWEEN 51 AND 60 THEN '[51-60]'
        WHEN `age` >= 61 THEN '[61+]'
        ELSE 'OUT of RANGE'
END AS `age_group`,
COUNT(`id`) AS `wizard_count`
FROM `wizzard_deposits`
GROUP BY `age_group`
ORDER BY `age_group`;
```

3.7. Цикли - WHILE statement

Пример с функция, където използваме **WHILE statement**

```
DELIMITER %%
CREATE FUNCTION ufn_IsWordComprised(setOfLetters VARCHAR (50), word VARCHAR (50))
RETURNS INT
deterministic
BEGIN
   DECLARE index_letter INT;
   DECLARE length_word INT;
   DECLARE letter CHAR(1);

   SET index_letter := 1;
   SET length_word := CHAR_LENGTH(word);

   WHILE (index_letter <= length_word)
   DO
      SET letter := SUBSTRING(word, index_letter, 1);

      IF (LOCATE(letter, setOfLetters) > 0) THEN SET index_letter := index_letter + 1;
      ELSE
        RETURN 0;
         END IF;
   END WHILE;

   RETURN 1;
END;
%%

SELECT ufn_IsWordComprised('oistmiahf', 'Sofia');
```

3.8. Цикли - LOOP statement

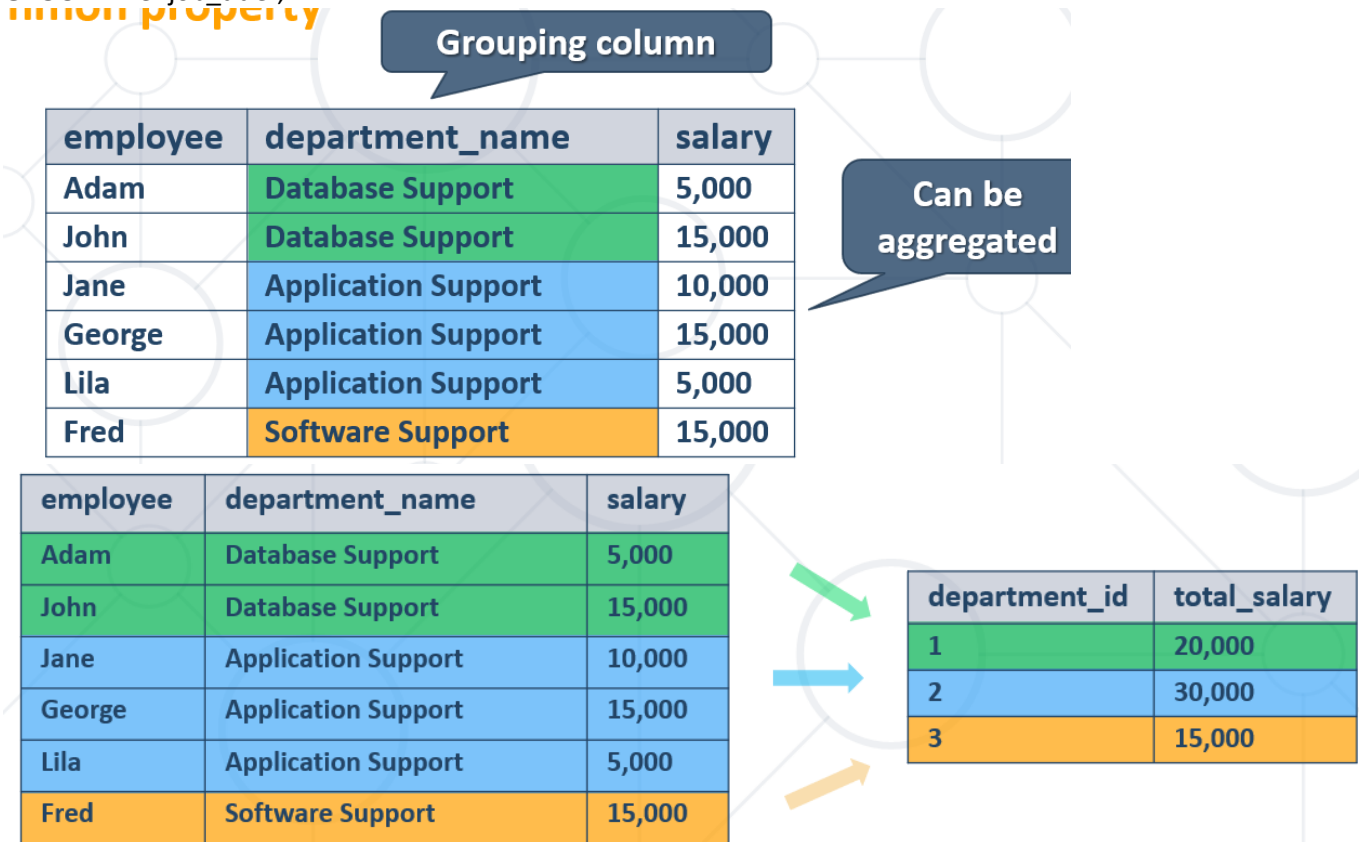
```
LOOP
   ...
   -- terminate the loop
   IF condition THEN
      LEAVE [label];
   END IF;
   ...
END LOOP;
```

4. Data Aggregation

4.1. Grouping - Consolidating Data Based On Criteria

- Grouping allows taking data into **separate groups** based on a **common property**

```
SELECT e.`job_title`, count(employee_id)
FROM `employees` AS e
GROUP BY e.`job_title`;
```



employee	department_name	salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000

employee	department_name	salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000

department_id	total_salary
1	20,000
2	30,000
3	15,000

- **Групиране по 2 критерия**

```
SELECT
   `deposit_group`,
   `magic_wand_creator`,
   MIN(`deposit_charge`) AS 'min_deposit_charge'
FROM
   `wizzard_deposits`
```

```
GROUP BY `deposit_group` , `magic_wand_creator`
ORDER BY `magic_wand_creator` ASC , `deposit_group`;
```

- With **GROUP BY** you can get each separate group and use an **"aggregate" function** over it (like Average, Min or Max)

4.2. Aggregate Functions

- Used to operate over **one** or **more** groups performing **data analysis** on every one
  - MIN, MAX, **AVG**, COUNT etc.
- They usually **ignore NULL** values

COUNT - counts the values (not nulls) in one or more columns based on grouping criteria

- Note that when we use **COUNT** we will ignore any employee with **NULL** salary.

```
SELECT `department_id`, COUNT(`first_name`) AS `Number of employees`
FROM `employees`
GROUP BY `department_id`
ORDER BY `department_id` ASC, `Number of employees` ASC;
```

Използваме звезда за по-мързеливо и за да ни брои всички елементи

```
SELECT `department_id`, COUNT(*) AS `Number of employees`
FROM `employees`
```

SUM - sums the values in a column

- If any department has no salaries **NULL** will be displayed.

```
SELECT e.`department_id`,
SUM(e.`salary`) AS 'TotalSalary'
FROM `employees` AS e
GROUP BY e.`department_id`;
```

MAX/MIN - takes the maximum value in a column.

```
SELECT e.`department_id`,
MAX(e.`salary`) AS 'Max Salary'
FROM `employees` AS e
GROUP BY e.`department_id`;
```

AVG calculates the average value in a column.

```
SELECT e.`department_id`,
  ROUND(AVG(e.`salary`),2) AS 'Average Salary'
FROM `employees` AS e
GROUP BY e.`department_id`
ORDER BY e.`department_id`;
```

Използване на AVG в секцията ORDER BY

```
SELECT w.`deposit_group`
FROM `wizzard_deposits` AS w
```

```
GROUP BY w.`deposit_group`
ORDER BY AVG(w.`magic_wand_size`)
LIMIT 1;
```

4.3. HAVING - Using Predicates While Grouping

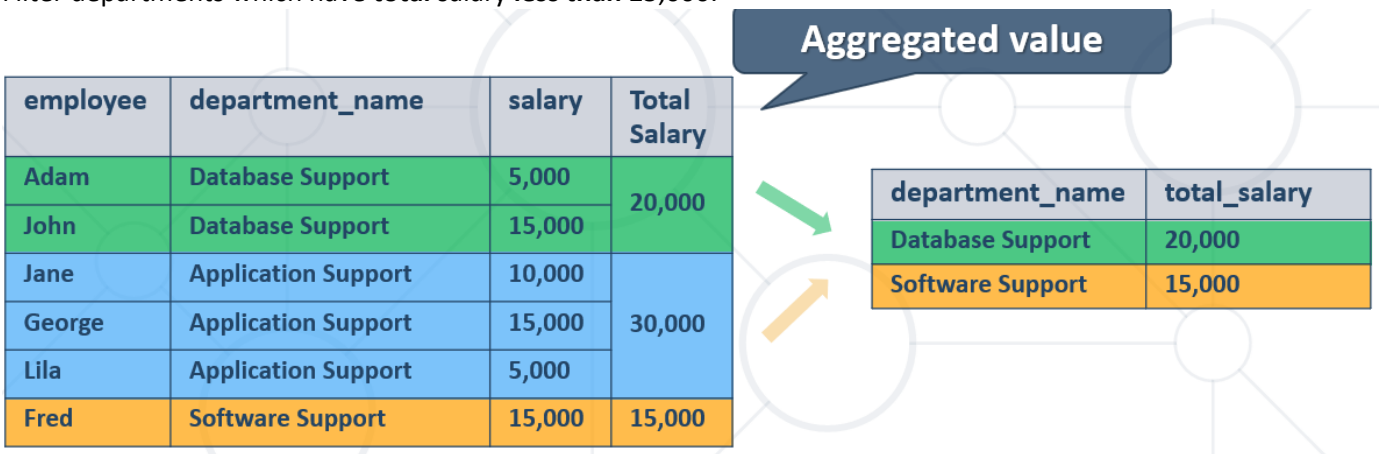
Having Clause

- The **HAVING** clause is used to filter data based on **aggregate** values.
  - We cannot use it **without** grouping **before** that
- Any Aggregate functions in the "**HAVING**" clause and in the "**SELECT**" statement are executed one time only
- **Unlike HAVING, the WHERE clause filters rows before the aggregation**

```
SELECT `deposit_group`,
SUM(`deposit_amount`) AS `total_sum`
FROM `wizzard_deposits`
WHERE `magic_wand_creator` = 'Ollivander family'
GROUP BY `deposit_group`
HAVING `total_sum` < 150000
ORDER BY `total_sum` DESC;
```

```
SELECT `deposit_group`,
*
FROM `wizzard_deposits`
WHERE `magic_wand_creator` = 'Ollivander family'
GROUP BY `deposit_group`
HAVING  SUM(`deposit_amount`) < 150000
ORDER BY `total_sum` DESC;
```

Filter departments which have **total** salary **less than** 25,000.




```
SELECT `department_id`,
SUM(`salary`) AS `TotalSalaryOfDepartment`
FROM `employees`
GROUP BY `department_id`
HAVING `TotalSalaryOfDepartment` < 120000;
```

4.4. MySQL OFFSET and LIMIT is used to specify which row should be fetched first.

```
SELECT e.`department_id`,
e.`salary` AS `third_highest_salary`
```

FROM `employees` AS e
WHERE (SELECT ine.`employee_id` FROM `employees` AS ine връща employee_id когато
 WHERE ine.`department_id` = e.`department_id` департамента съвпада
 GROUP BY ine.`salary` когато е третата най-висока заплата
ORDER BY `salary` DESC **LIMIT** 1 **OFFSET** 2 – **изкарай само един резултат, започвайки да търсиш след втория**
) = e.`employee_id` когато има съвпадение по employee_id
GROUP BY e.`department_id`
ORDER BY e.`department_id` ASC;

LIMIT 2, 1; - започни да търсиш след втория запис, и ограничи до 1 запис изхода

4.5. Debug mode – EXPLAIN SELECT ..

EXPLAIN SELECT *, **SUBSTRING**(`title`, 1, 4) **FROM** `books` **LIMIT** 20 **OFFSET** 11;
OFFSET Отмести/започни от 11тия запис нататък

4.6. Вложени агрегиращи заявки

```
SELECT e.`first_name`, e.`last_name`, e.`department_id`
FROM `employees` AS e
WHERE e.`salary` > (
        SELECT
        AVG(inn.`salary`) FROM `employees` AS inn          - намери средната заплата
        WHERE inn.`department_id` = e.`department_id`   - ако средната заплата след групиране отговаря на запла-
        GROUP BY inn.`department_id`                       тата на всеки пореден служител от съотв. департамент
)
ORDER BY e.`department_id`, e.`employee_id`
LIMIT 10;
```

Друго решение на същата задача – използваме сега само един Alias:

```
SELECT `first_name`, `last_name`, `department_id`
FROM `employees` AS e
WHERE e.salary > (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id GROUP BY
department_id)
ORDER BY `department_id`, `employee_id`
LIMIT 10
```

ВАЖНО – при update и използване на нестнати операции

```
UPDATE employees_clients AS ec
SET ec.employee_id =
(
        SELECT ec.employee_id – от същата таблица ec реално
        GROUP BY ec.employee_id
        ORDER BY COUNT(ec.employee_id) ASC, ec.employee_id ASC
        LIMIT 1
)
WHERE ec.employee_id = ec.client_id;
```

4.7. Невложени заявки вършещи работа като вложени заявки

```
SELECT
   SUM(`hw`.`deposit_amount` - `gw`.`deposit_amount`) AS 'sum_difference'
FROM
```

```
    `wizzard_deposits` AS `hw`,
    `wizzard_deposits` AS `gw`
WHERE
    `gw`.`id` - `hw`.`id` = 1;
```

5. Table relations – видове връзки

5.1. Database design

Steps in database design

1. Identification of Entities

- Entity tables represent objects from the real world
  - Most often they are nouns in the specification

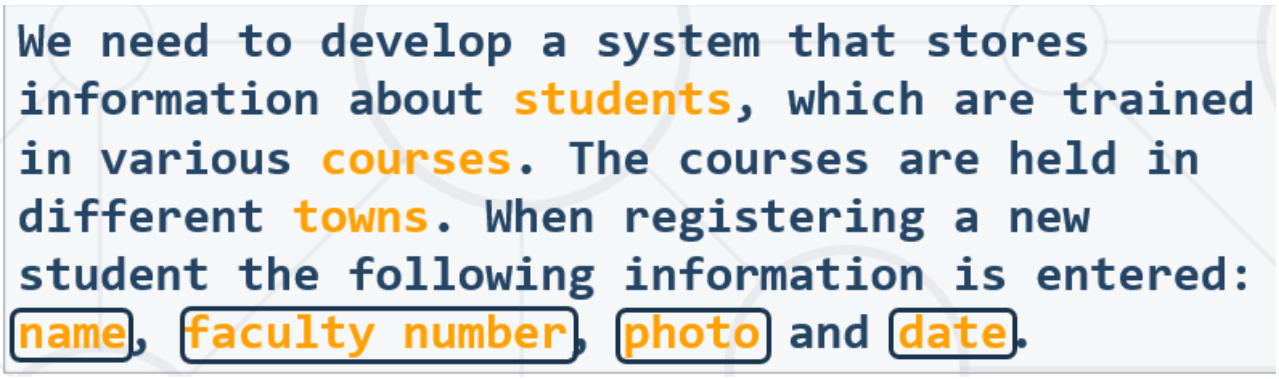
For example:

We need to develop a system that stores information about **students**, which are trained in various **courses**. The courses are held in different **towns**. When registering a new student the following information is entered: name, faculty number, photo and date.

Entities: **Student**, **Course**, **Town**

2. Define Table Columns

- Columns are clarifications for the entities in the text of the specification, for example:



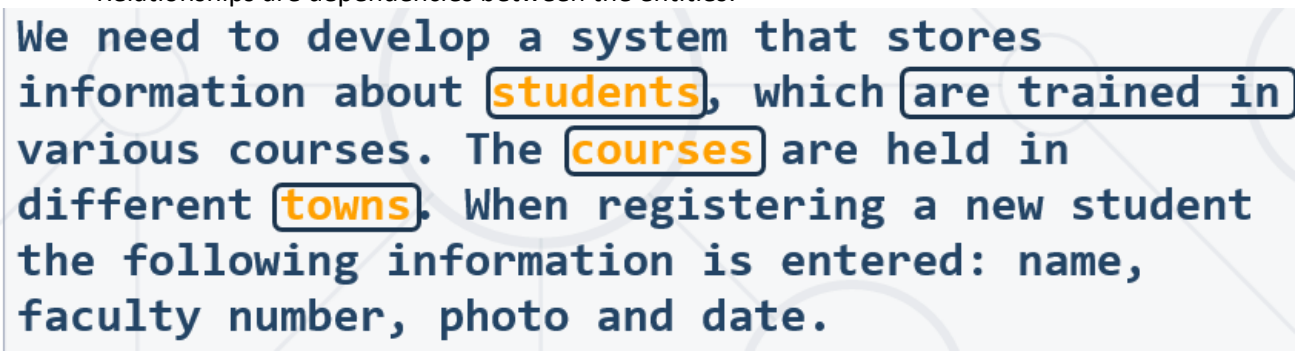
- Students have the following characteristics:
  - Name, faculty number, photo, date of enlistment and a list of courses they visit

3. Defining primary keys

- Always define an additional column for the primary key
  - Don't use an existing column (for example SSN)
  - Can be an integer number
  - **Must be** declared as **a PRIMARY KEY**
  - Use **AUTO_INCREMENT** to implement auto-increment
  - **Put the primary key as a first column**
- Exceptions
  - Entities that have well known ID, e.g. countries (BG, DE, US) and currencies (USD, EUR, BGN)

4. Modelling relationships

- Relationships are dependencies between the entities:



  - "Students are trained in courses" – **many-to-many** relationship.
  - "Courses are held in towns" – **many-to-one** (or many-to-many) relationship

5. Defining constraints

6. Filling test data

5.2. Table relations

- **Relationships** between tables are based on interconnections: **PRIMARY KEY** / **FOREIGN KEY**

Foreign Key

```
CREATE TABLE `Orders` (
    `OrderID` int NOT NULL,
    `OrderNumber` int NOT NULL,
```

```
    `PersonID` int,
    PRIMARY KEY (`OrderID`),

CONSTRAINT `fk_source_target`
    FOREIGN KEY `Orders`(`PersonID`) REFERENCES `Persons`(`PersonID`)
);
```

SQL FOREIGN KEY on ALTER TABLE

```
ALTER TABLE `products`
ADD CONSTRAINT `fk_products_categories` FOREIGN KEY `products`(`category_id`)
REFERENCES `categories`(`id`);
```

- The **foreign key** is an **identifier** of a record located in another table (usually its primary key)
- By using relationships we avoid repeating data in the database
- Relationships have multiplicity:

One-to-many – e.g. mountains / peaks - от едната страна на foreign key полето трябва да е UNIQUE

Много модели на един производител в случая

```
CREATE TABLE `manufacturers`(
   `manufacturer_id` INT AUTO_INCREMENT PRIMARY KEY NOT NULL,    - реално е UNIQUE това поле
   `name` VARCHAR(45),
   `established_on` DATE
);
```

Foreign key е от Many страната на релацията.

```
CREATE TABLE `models`(
        `model_id` INT UNIQUE NOT NULL,
   `name` VARCHAR(45),
   `manufacturer_id` INT NOT NULL  - а това поле не е UNIQUE
);

ALTER TABLE `models`
ADD CONSTRAINT `fk_models_manufacturers`
FOREIGN KEY `models`(`manufacturer_id`)
REFERENCES `manufacturers`(`manufacturer_id`);

INSERT INTO `manufacturers` (`name`, `established_on`)
VALUES
('BMW', '1916-03-01'),
('Tesla', '2003-01-01'),
('Lada', '1966-05-01');

INSERT INTO `models` (`model_id`, `name`, `manufacturer_id`)
VALUES
(101, 'X1', 1),
(102, 'i6', 1),
(103, 'Model S', 2),
(104, 'Model X', 2),
```

```
(105, 'Model 3', 2),
(106, 'Nova', 3);
```

One-to-one – e.g. example driver / car – и от двете страни на foreign key полето трябва да е UNIQUE

```
CREATE TABLE `people`(
   `person_id` INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
   `first_name` VARCHAR(45),
   `salary` DECIMAL(10, 2) NOT NULL,
   `passport_id` INT UNIQUE NOT NULL – това поле е UNIQUE
);


CREATE TABLE `passports`(
   `passport_id` INT UNIQUE NOT NULL,    -  и това поле също е UNIQUE
   `passport_number` VARCHAR(45) UNIQUE
);

ALTER TABLE `people`
ADD CONSTRAINT `fk_people_passports`
FOREIGN KEY `people`(`passport_id`)
REFERENCES `passports`(`passport_id`);


INSERT INTO `people` (`first_name`, `salary`, `passport_id`)
VALUES
('Roberto', 43300.00, 102),
('Tom', 56100.00, 103),
('Yana', 60200.00, 101);

INSERT INTO `passports` (`passport_id`, `passport_number`)
VALUES
(101, 'N34FG21B'),
(102, 'K65LO4R7'),
(103, 'ZE657QP2');
```

Всяко поле AUTO_INCREMENT и всяко поле PRIMERY KEY реално е **UNIQUE**

Many-to-many – e.g. student / course – изпълнява се с composite primary key и mapping table

Mapping table and composite primary key - пример за many-to-many relations

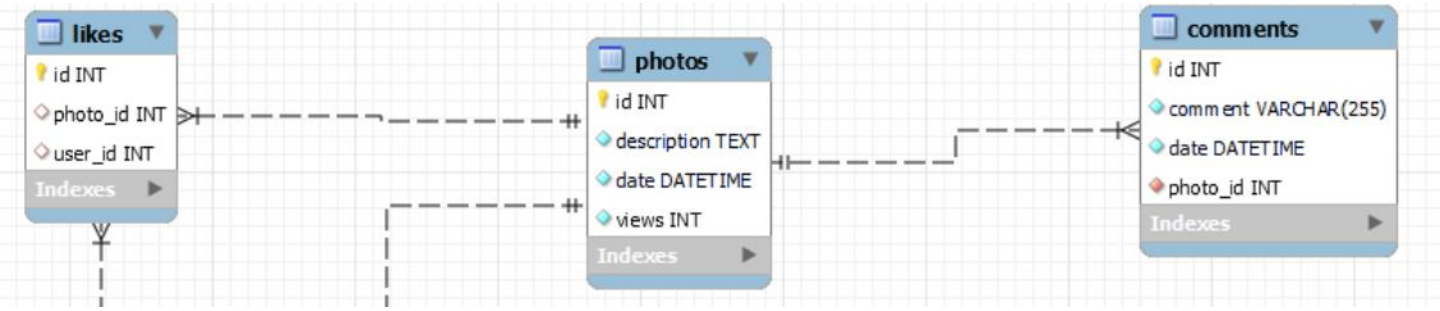
```
CREATE TABLE `students`(
          `student_id` INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
   `name` VARCHAR(45)
);

CREATE TABLE `exams` (
`exam_id` INT NOT NULL PRIMARY KEY,
`name` VARCHAR(30)
);
```

Важно: за да работи foreign key, трябва да има primary key в таблиците

```
CREATE TABLE `students_exams`(
`student_id` INT,
`exam_id` INT,

CONSTRAINT pk_students_exams
PRIMARY KEY (`student_id`, `exam_id`),

CONSTRAINT fk_students_exams_students
FOREIGN KEY (`student_id`)
REFERENCES `students`(`student_id`),

CONSTRAINT fk_students_exams_exams
FOREIGN KEY (`exam_id`)
REFERENCES `exams`(`exam_id`)
);
```

Без композитен ключ, използваме DISTINCT за да няма повторения

#08. Count Likes and Comments



В случая, като съединим 3 таблици, и за всеки различен лайк на снимка има примерно по два коментара.

Как процедираме – пускаме първо заявката със звезда, и виждаме какво става.

```
SELECT *
FROM `photos` AS p
LEFT JOIN `likes` AS l
ON l.`photo_id` = p.`id`
LEFT JOIN `comments` AS c
ON c.`photo_id` = p.`id`;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	description	date	views	id	photo_id	user_id	id	comment	date	photo_id
1	Nullam sit amet turpis el...	2019-09-22 14:48:03	0	46	1	46	19	amet erat nulla ...	2019-06-...	1
1	Nullam sit amet turpis el...	2019-09-22 14:48:03	0	46	1	46	94	lacus purus aliq...	2019-05-...	1
1	Nullam sit amet turpis el...	2019-09-22 14:48:03	0	52	1	10	19	amet erat nulla ...	2019-06-...	1
1	Nullam sit amet turpis el...	2019-09-22 14:48:03	0	52	1	10	94	lacus purus aliq...	2019-05-...	1
1	Nullam sit amet turpis el...	2019-09-22 14:48:03	0	57	1	18	19	amet erat nulla ...	2019-06-...	1
1	Nullam sit amet turpis el...	2019-09-22 14:48:03	0	57	1	18	94	lacus purus aliq...	2019-05-...	1

```
SELECT p.`id`, COUNT(DISTINCT l.`id`) AS `likes_count`, COUNT(DISTINCT c.`id`) AS `comments_count`
FROM `photos` AS p
LEFT JOIN `likes` AS l
ON l.`photo_id` = p.`id`
LEFT JOIN `comments` AS c
ON c.`photo_id` = p.`id`
GROUP BY p.`id`
ORDER BY `likes_count` DESC, `comments_count` DESC, p.`id` ASC;
```

Self-referencing

```
CREATE TABLE `teachers`(
`teacher_id` INT PRIMARY KEY,
`name` VARCHAR(20),
`manager_id` INT,

CONSTRAINT fk_teachers_teachers
FOREIGN KEY `teachers`(`manager_id`)
REFERENCES `teachers`(`teacher_id`)
);

INSERT INTO `teachers` (`teacher_id`, `name`)
VALUES
(101, 'John'),
(102, 'Maya'),
(103, 'Silvia'),
(104, 'Ted'),
(105, 'Mark'),
(106, 'Greta');

UPDATE `teachers` SET `manager_id` = null WHERE (`name` = 'John');
UPDATE `teachers` SET `manager_id` = 106 WHERE (`name` = 'Maya');
UPDATE `teachers` SET `manager_id` = 106 WHERE (`name` = 'Silvia');
UPDATE `teachers` SET `manager_id` = 105 WHERE (`name` = 'Ted');
UPDATE `teachers` SET `manager_id` = 101 WHERE (`name` = 'Mark');
UPDATE `teachers` SET `manager_id` = 101 WHERE (`name` = 'Greta');
```

Друг пример за self-referencing

```
SELECT e.`employee_id`, e.`first_name`, m.`employee_id` AS 'manager_id', m.`first_name` AS 'manager_name'
FROM `employees` AS e
JOIN `employees` AS m
ON e.`manager_id` = m.`employee_id`;
```

5.3. JOIN - Retrieving Related Data

Joins

- Table relations are useful when combined with JOINS
- With JOINS we can get data from two tables **simultaneously**
  - JOINS require at least two tables and a "**join condition**"

Example:

```
SELECT * FROM table_a
 JOIN table_b ON     - добави таблица b към таблица a
  table_b.common_column = table_a.common_column;     да са равни
```

За използването на JOIN не е необходимо използването на външен ключ (foreign key)

```
SELECT v.`driver_id`, v.`vehicle_type`,
CONCAT(c.`first_name`, ' ', c.`last_name`) AS 'driver_name'
FROM `vehicles` AS v
JOIN `campers` AS c    -  добави таблица campers към таблица  vehicles
ON v.`driver_id` = c.`id`;
```

5.4. Cascade Operations

- Cascading allows when a change is made to certain entity, this change to apply to all related entities

Ако изтриеш нещо, изтрий всичко по веригата.
Ако update-неш нещо, то го update-ни навсякъде по веригата.

CASCADE DELETE

- **CASCADE** can be either **DELETE** or **UPDATE**.
- Use **CASCADE DELETE** when:
  - The related entities are **meaningless** without the "main" one
- Do **not** use **CASCADE DELETE** when:
  - You make "**logical delete**"
  - You preserve **history**
- Keep in mind that in more complicated relations it won't work with **circular** references

Пример 1:

- Write a query to create a one-to-many relationship
- When an mountains gets removed from the database, all of his peaks are deleted too

```
CREATE TABLE `mountains`(
`id` INT PRIMARY KEY AUTO_INCREMENT,
`name` VARCHAR(20) NOT NULL
);

CREATE TABLE `peaks`(
`id` INT PRIMARY KEY AUTO_INCREMENT,
`name` VARCHAR(20) NOT NULL,
`mountain_id` INT,
CONSTRAINT `fk_mountain_id`
FOREIGN KEY(`mountain_id`)
REFERENCES `mountains`(`id`)
ON DELETE CASCADE
);
```

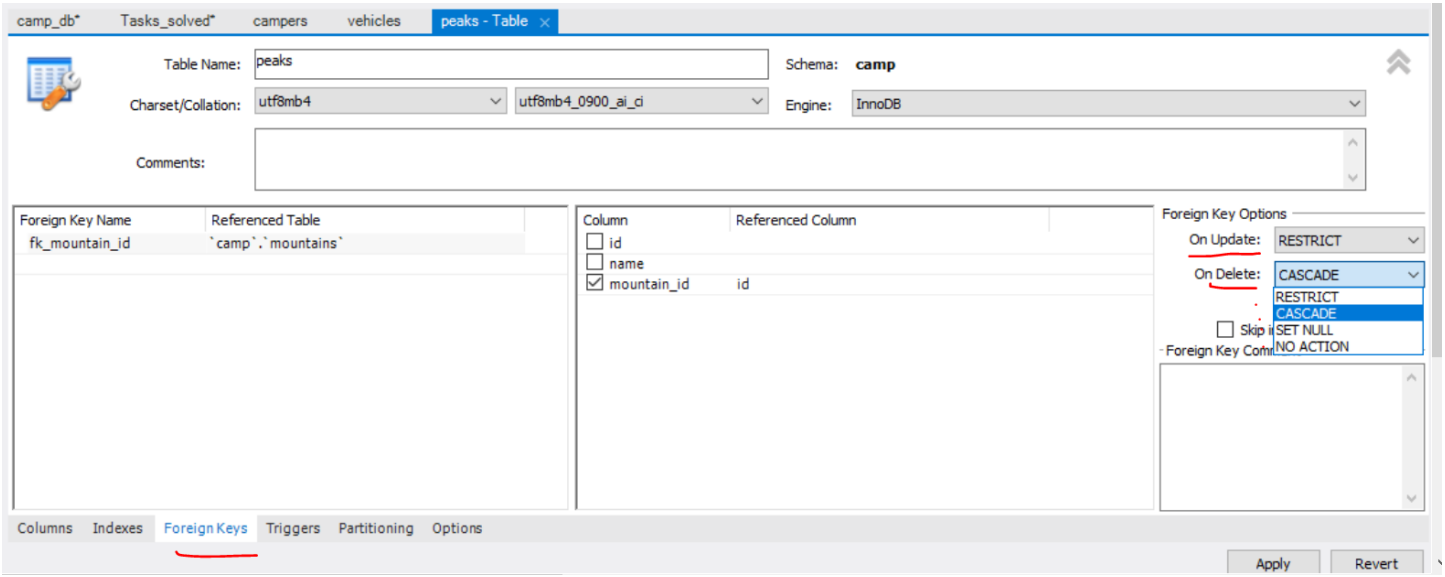
Пример 2:

```
CREATE TABLE drivers(
  driver_id INT PRIMARY KEY,
  driver_name VARCHAR(50)
);
CREATE TABLE cars(
  car_id INT PRIMARY KEY,
  driver_id INT,
  CONSTRAINT fk_car_driver FOREIGN KEY(driver_id)
  REFERENCES drivers(driver_id) ON DELETE CASCADE
);
```

CASCADE UPDATE

- Use **CASCADE UPDATE** when:
  - The primary key is **NOT** identity (not **auto-increment**) and therefore it **can** be changed
  - Best used with **UNIQUE** constraint
- Do **not** use **CASCADE UPDATE** when:
  - The primary is identity (**auto-increment**)
- Cascading can be avoided using triggers or procedures

```
CREATE TABLE drivers(
 driver_id INT PRIMARY KEY,
 driver_name VARCHAR(50)
);
CREATE TABLE cars(
 car_id INT PRIMARY KEY,
 driver_id INT,
 CONSTRAINT fk_car_driver FOREIGN KEY(driver_id)
 REFERENCES drivers(driver_id) ON UPDATE CASCADE
);
```



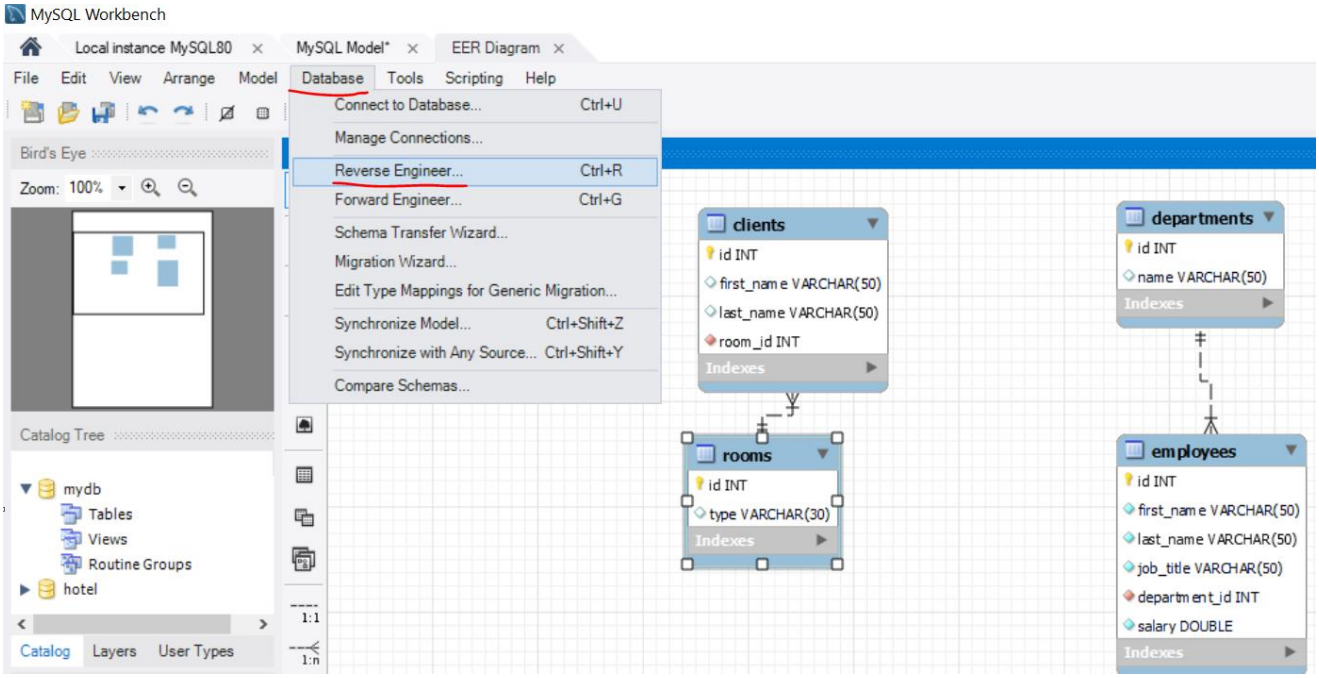
5.5. E/R Diagram

Понякога първо си чертаем E/R диаграмата, и след това пишем заявките

Relational Schema

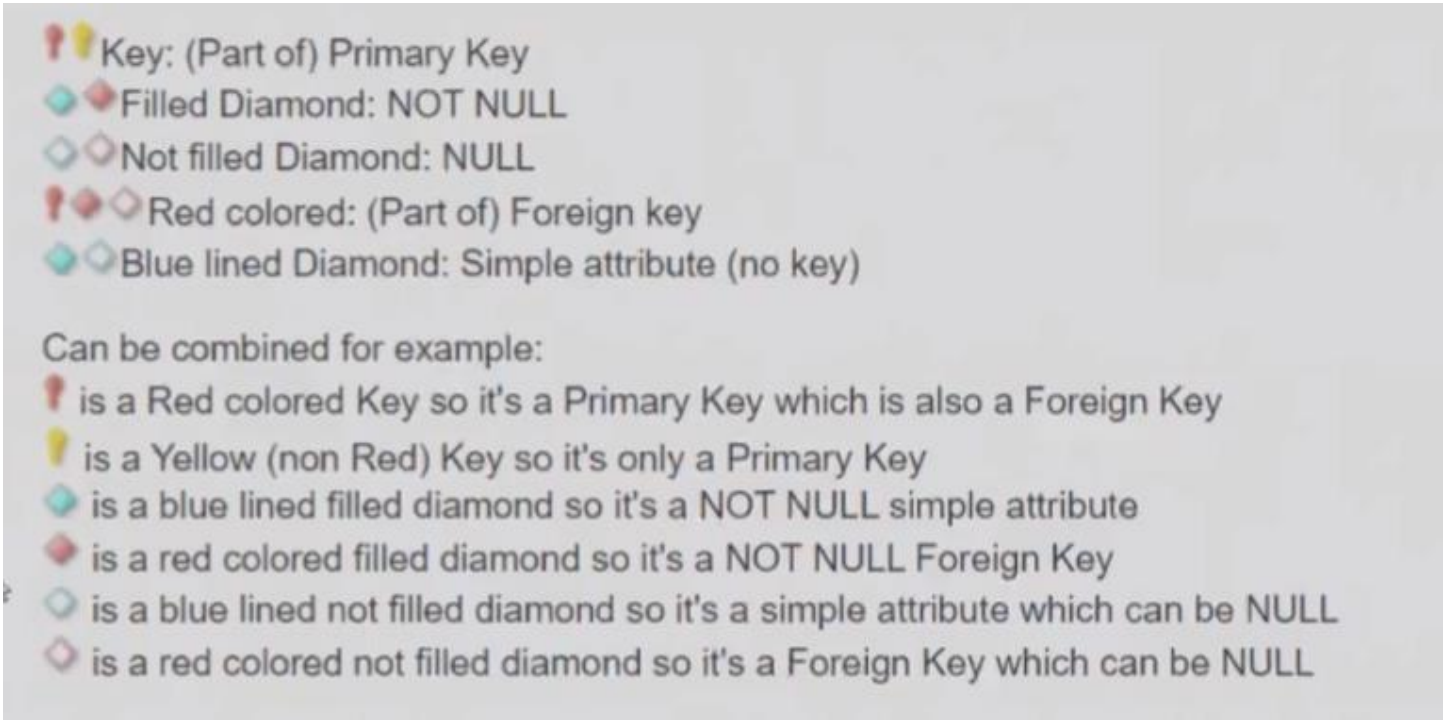
- **Relational schema** of a DB is the collection of:
  - The schemas of all tables
  - Relationships between the tables
  - Any other database objects (e.g. constraints)
- The relational schema describes the **structure** of the database
  - Doesn't contain data, but **metadata**
- Relational schemas are **graphically** displayed in Entity / Relationship diagrams (**E/R Diagrams**)

Reverse engineering

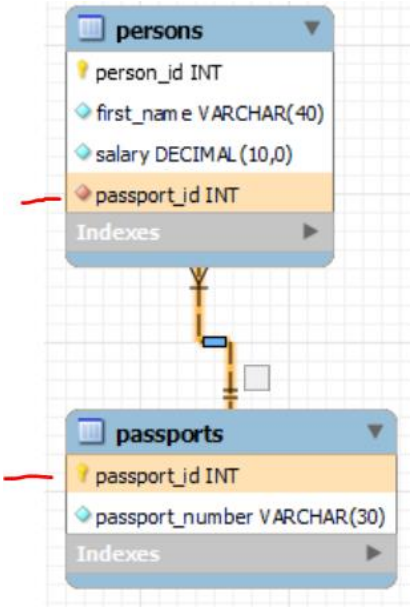


Добра практика е да слагаме префикс на името на всяка таблица, която създаваме. Например:
my_Products
my_Clients
my_personnel

или
wp_posts
wp_links
wp_commentmeta
wp_comments
wp_users



В persons таблицата пишем дефиницията за foreign key



Като посочим стрелката, и ни показва кои полета са свързани.

Как тълкуваме стрелката - в случая таблица persons (parent) има foreign key passport_id, който сочи към таблица passports (child дете) с ключ passport_id (или passport_id на persons сочи към passport_id на passports).

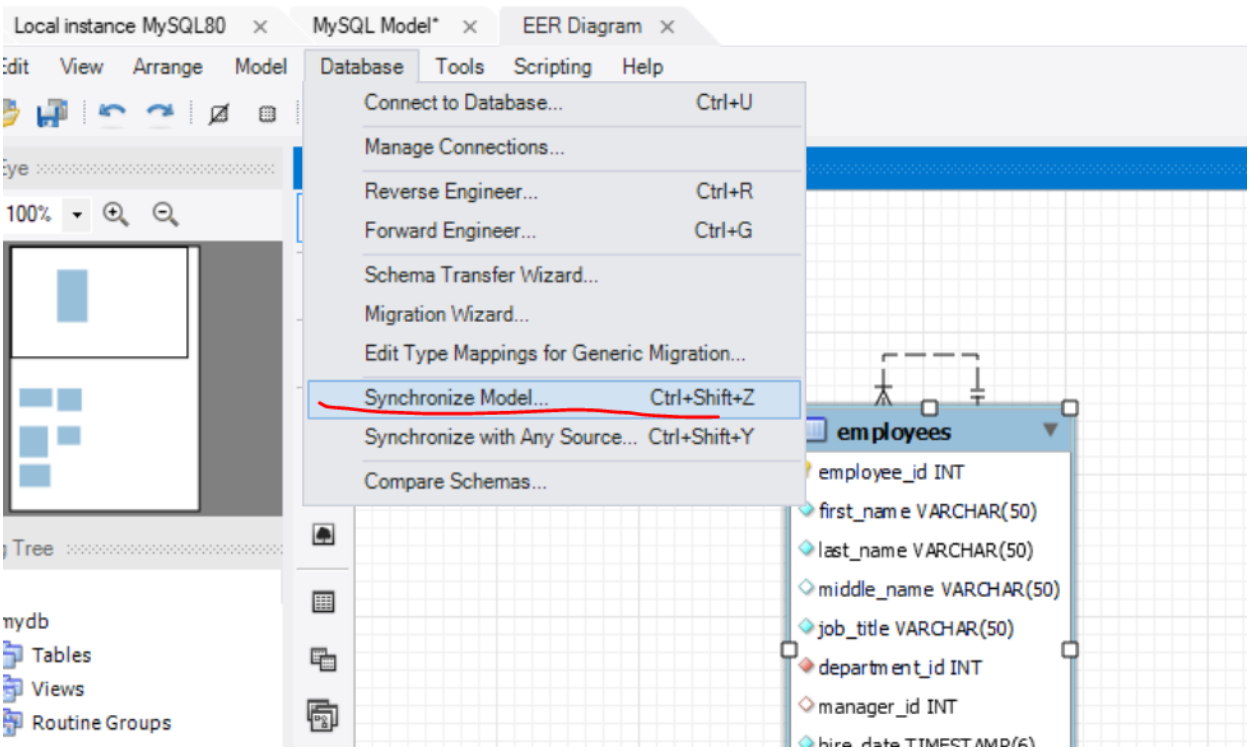
Какво означава линията:

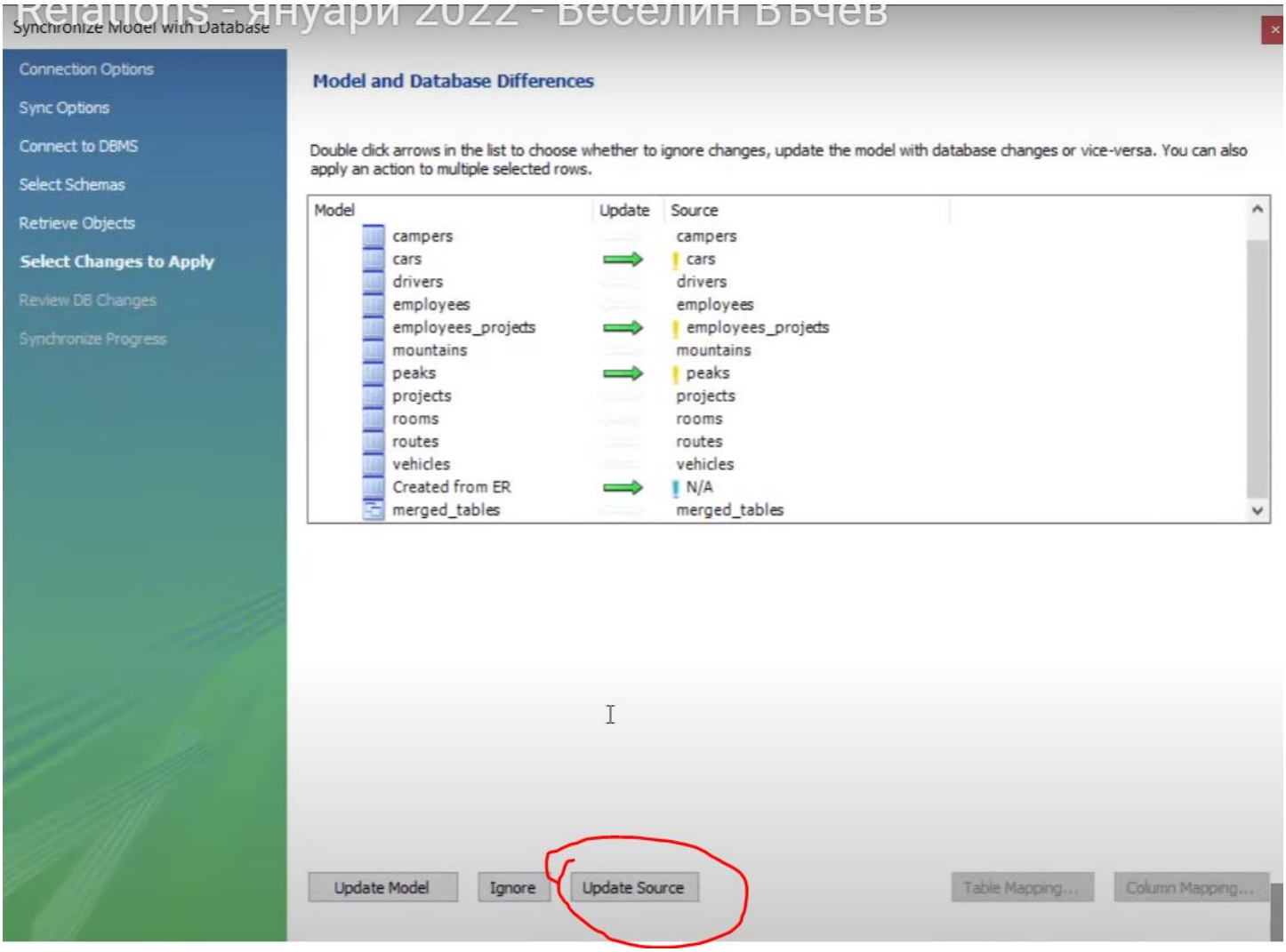
- **Плътна линия означава в двойката баща-дете, че и двете са задължителни/и двете са primary keys**
- **Прекъсната линия означава, че в двойката parent-child не са задължителни и двете**

Syncronize model – от E/R диаграма към създаване на база дани

Каквото сме си нарисували на диаграма, го направи на истинска база

QL Workbench





6. JOINS

6.1. Gathering Data From Multiple Tables

Можем да правим JOIN и на таблици без foreign keys

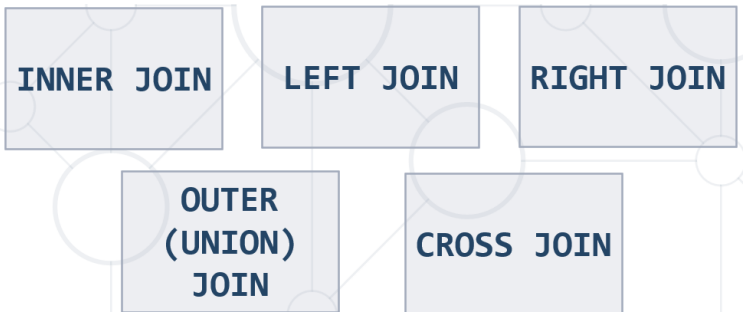
- Sometimes you need data from several tables:

Cartesian product – всяко от едната таблица с всяко от другата таблица

- Each row in the first table is paired with **all** the rows in the second table
  - When there is **no relationship** defined between the two tables

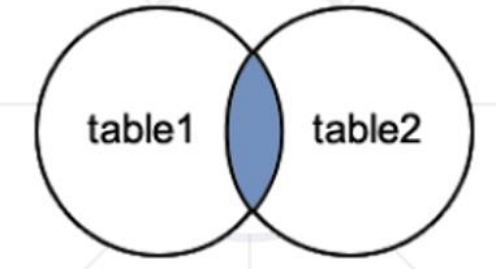
6.2. JOINS – used to collect data from two or more tables

- Types:



INNER JOIN = JOIN = взема сечението

- Produces a set of records which **match in both tables**

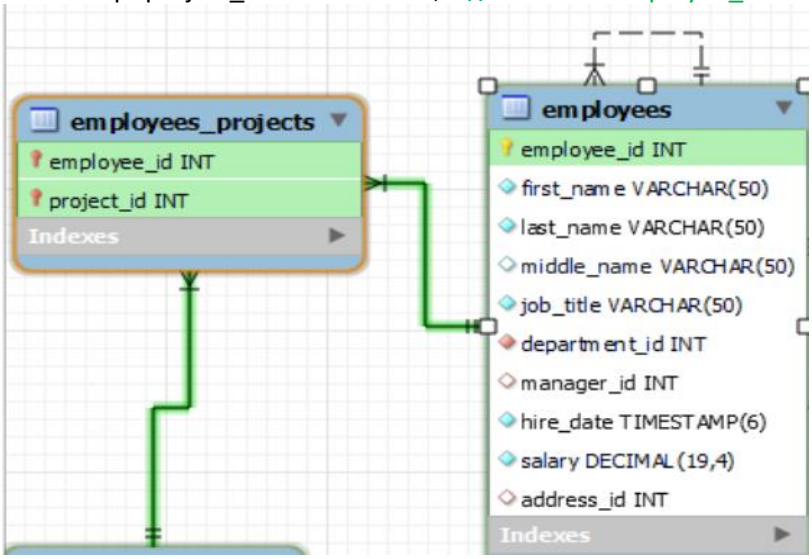



```
SELECT e.`first_name`, d.`name` AS 'dept_name'
FROM `employees` AS e                    - таблица 1 = e
INNER JOIN `departments` AS d            - таблица 2 = d
ON e.`department_id` = d.`department_id`;
```

```
//Селектирай служителите, само които имат проекти
SELECT e.`employee_id`, e.`first_name`, p.`name` AS `project_name`
FROM `employees` AS e
INNER JOIN `employees_projects` AS ep
ON e.`employee_id` = ep.`employee_id`;
```

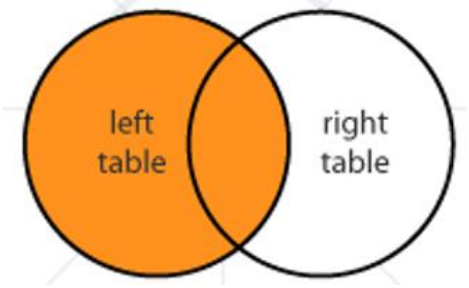
Или можем и така:

```
SELECT e.`employee_id`, e.`first_name`, p.`name` AS `project_name`
FROM `employees` AS e
LEFT JOIN `employees_projects` AS ep   //вземи всичко отляво (таблицата employees като първа таблица)
ON e.`employee_id` = ep.`employee_id`
WHERE ep.`project_id` IS NOT NULL;    //когато за employee_id от employees ИМА ep.`project_id`
```



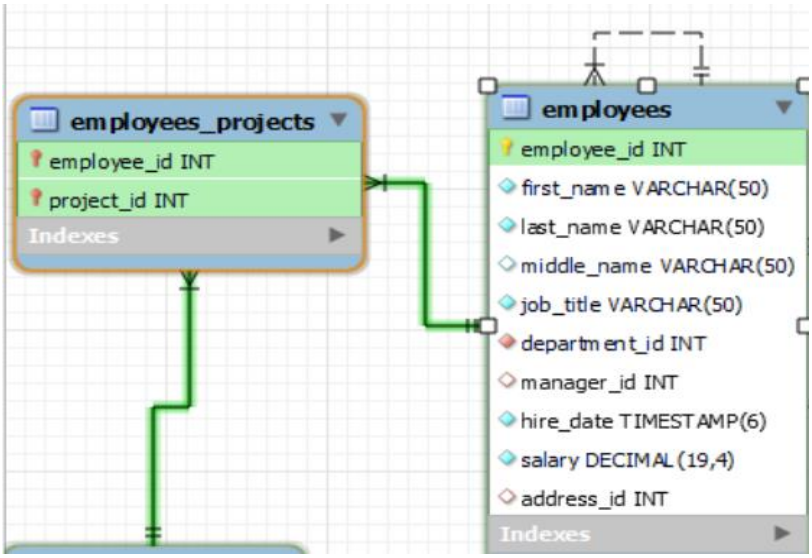
LEFT JOIN

- Matches every entry in **left** table regardless of match in the **right**



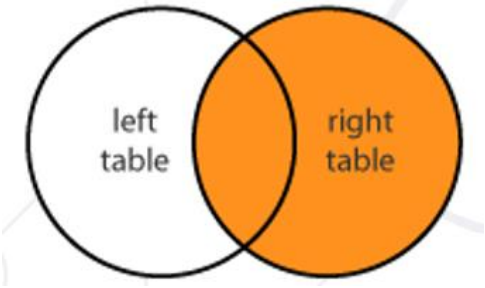
Example, see powerpoint presentation

Тези колеги, които не са в нито един проект
SELECT e.`employee_id`, e.`first_name`
FROM `employees` **AS** e
LEFT JOIN `employees_projects` **AS** ep //вземи всичко отляво (таблицата employees като първа таблица)
ON e.`employee_id` = ep.`employee_id`
WHERE ep.`project_id` **IS NULL** //когато за employee_id от employees няма ep.`project_id`
ORDER BY e.`employee_id` DESC
LIMIT 3;



RIGHT JOIN

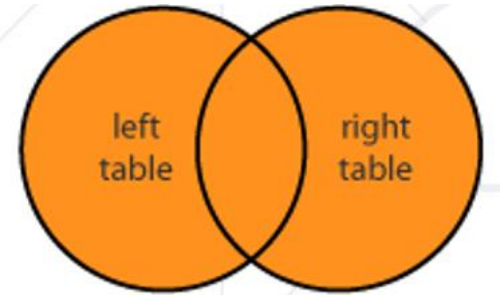
- Matches every entry in **right** table regardless of match in the **left**



Example, see powerpoint presentation

OUTER (FULL JOIN)

- Returns all records in both tables regardless of **any** match
  - Less useful than **INNER**, **LEFT** or **RIGHT JOINs** and it's **not implemented in MySQL**
  - We can use **UNION** of a **LEFT** and **RIGHT JOIN**



OUTER JOIN – взема всичко без сечението, но не е имплементирано в MySQL

UNION of LEFT and RIGHT JOIN

SELECT students.name, courses.name
FROM students
LEFT JOIN courses
ON students.course_id = courses.id

UNION

SELECT students.name, courses.name
FROM students
RIGHT JOIN courses
ON students.course_id = courses.id

Example, see powerpoint presentation

CROSS JOIN

- Produces a set of associated rows of two tables
  - Multiplication of each row in the first table with each in second
  - The result is a **Cartesian** product, when there's **no condition** in the **WHERE** clause

6.3. Subqueries – една заявка в друга заявка

- Subqueries – SQL query inside a larger one
- Can be nested in **SELECT, INSERT, UPDATE, DELETE**
  - Usually added within a **WHERE** clause

```
SELECT COUNT(e.employee_id) AS `count`
FROM employees AS e
WHERE e.salary >
(
SELECT AVG(salary) AS 'average_salary' FROM employees
);
```

Конкатениране на повече от две таблици

```
SELECT e.`first_name`, e.`last_name`, t.`name` , adr.`address_text`
FROM `employees` AS e
JOIN `addresses` AS adr
ON e.`address_id` = adr.`address_id`
JOIN `towns` AS t
ON adr.`town_id` = t.`town_id`
ORDER BY e.`first_name` ASC, e.`last_name` ASC
LIMIT 5;
```

WHERE и ORDER BY ги слагаме след всички JOIN

```
SELECT c.`country_code`, m.`mountain_range`, p.`peak_name`, p.`elevation`
FROM `countries` AS c
JOIN `mountains_countries` AS mc
ON c.`country_code` = mc.`country_code`
JOIN `mountains` AS m
ON mc.`mountain_id` = m.`id`
JOIN `peaks` AS p
ON m.`id` = p.`mountain_id`
WHERE c.`country_code` = 'BG' AND p.`elevation` > 2835
ORDER BY p.`elevation` DESC;
```

Когато трябва да визуализираме променена стойност на дадена клетка

```
SELECT e.`employee_id`, e.`first_name`,
(
CASE
        WHEN YEAR(p.`start_date`) > 2004
  THEN NULL                                          - променената стойност е NULL
  ELSE p.`name`
END
)
AS 'project_name'
FROM `employees` AS e
JOIN `employees_projects` AS ep
ON e.`employee_id` = ep.`employee_id`
JOIN `projects` AS p
ON p.`project_id` = ep.`project_id`
WHERE e.`employee_id` = 24
ORDER BY `project_name`;
```

#15. *Continents and Currencies

```
#групирана по два критерия връща за всеки континент и за всяка валута, то по колко пъти се използва дадена валута
SELECT contr.`continent_code`, contr.`currency_code`,
  COUNT(*) AS `currency_usage`
FROM `countries` AS contr
GROUP BY contr.`continent_code` , contr.`currency_code`
HAVING `currency_usage` > 1  #всички валути, които се използват в даден континент в повече от една държава
AND
#от всички валути колко пъти са използвани, сравни само тази коя е използвана най-много пъти
`currency_usage` = (SELECT          #c.`currency_code`,
    COUNT(*) AS `most_used_currency`
```

```
            FROM `countries` AS c
            WHERE c.`continent_code` = contr.`continent_code`
            GROUP BY c.`currency_code`
            ORDER BY `most_used_currency` DESC
            LIMIT 1)
ORDER BY contr.`continent_code` ASC, contr.`currency_code` ASC;
```

6.4. More hacks

UPDATE JOIN

```
#Task 3 - Update - version in which we have employees with 0 clients – вложена заявка
UPDATE employees_clients AS ec
SET ec.employee_id =
(
        SELECT COUNT(e.id) FROM employees AS e
        LEFT JOIN (SELECT * from employees_clients) AS emcl    - нова инстанция на същата таблица
        ON emcl.employee_id = e.id
        GROUP BY e.id
        ORDER BY COUNT(e.id) ASC, e.id ASC
        LIMIT 1
)
WHERE ec.employee_id = ec.client_id;
```

Или така също става

```
UPDATE `products` AS prr
 JOIN `categories` AS c
 ON prr.`category_id` = c.`id`
 JOIN `reviews` AS r
 ON r.`id` = prr.`review_id`
 SET prr.`price` = prr.`price` * 0.7
 WHERE c.`name` = 'Phones and tablets' AND r.`rating` < 4;
```

DELETE JOIN

Вариант 1 - Работи

```
DELETE emp FROM employees AS emp – изтрий от таблица emp
LEFT JOIN employees_clients AS ec
ON ec.employee_id = emp.id
WHERE ec.client_id IS NULL;   -   тези, служители, които нямат клиенти, при JOIN имат cliend_id  да е NULL
```

Вариант 2

```
DELETE FROM employees WHERE id =   – изтрий от таблица emp
(
        SELECT emp.id FROM (SELECT * FROM employees) AS emp  - нова инстанция на същата таблица
        LEFT JOIN employees_clients AS ec
        ON ec.employee_id = emp.id
        WHERE ec.client_id IS NULL      -   тези, служители, които нямат клиенти, при JOIN имат cliend_id  да е NULL
)
```

GROUP BY plus JOIN

```
SELECT CONCAT(emp.first_name, ' ', emp.last_name) AS 'name',
```

```
emp.started_on,
COUNT(ec.employee_id) AS count_of_clients
FROM employees AS emp
LEFT JOIN employees_clients AS ec
ON ec.employee_id = emp.id
GROUP BY ec.employee_id
ORDER BY count_of_clients DESC, emp.id ASC
LIMIT 5;
```

6.4. Indices и балансирано бинарно дърво

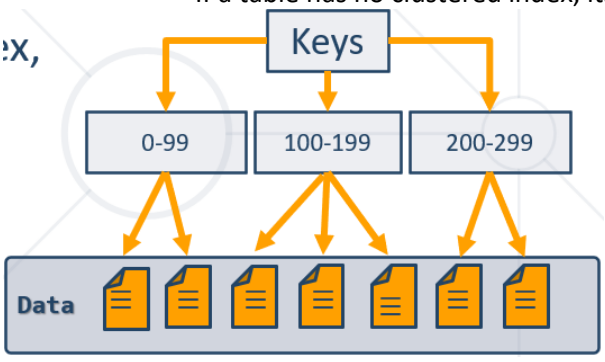
- Structures associated with a table or view that speeds retrieval of rows
  - Usually implemented as **B-trees**
- Indices can be built-in the table (**clustered**) or stored externally (**non-clustered**)
- Adding and deleting records in indexed tables is slower!
  - Indices should be used for big tables only (e.g. 50 000 rows)

Когато имаме много на брой данни, в даден момент се налага преиндексиране на базата данни, за да се постигне балансирано бинарно дърво.
Небалансираното дърво забавя много операциите.

```
id PRIMARY KEY -> B-tree
```

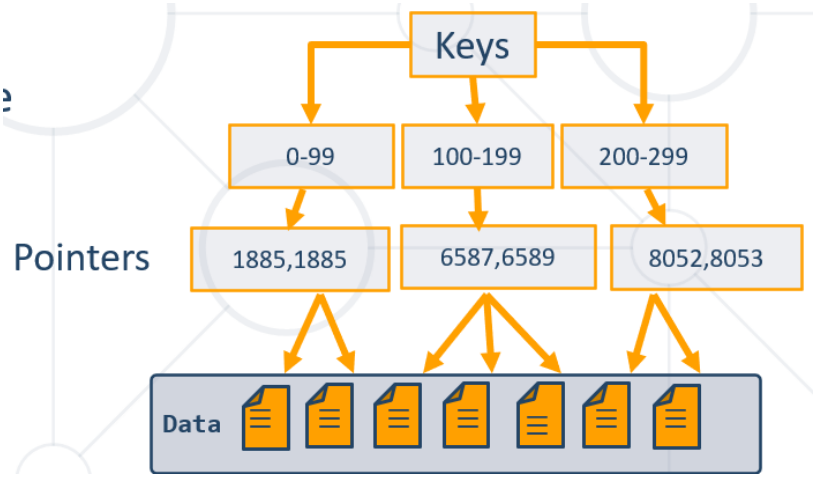
Clustered Indices

- **Clustered index determine the order of data**
  - Very useful for fast execution of **WHERE**, **ORDER BY** and **GROUP BY** clauses
- Maximum 1 clustered index per table
  - If a table has no clustered index, its data rows are stored in an **unordered structure** (heap)



Non-Clustered Indices

- Useful for fast retrieving a **single record** or a **range** of records
  - Each **key value entry** has a pointer to the data row that contains the key value
- Maintained in a separate structure in the DB



Синтаксис на индекс
CREATE INDEX
ix_users_first_name_last_name
ON `users`(`first_name`, `last_name`);

7. Database Programmability

Да избягваме в практиката функции и процедури в MySQL

Важно: ПЪРВО разписваме проста заявка, и след това я копираме/слагаме в структурата на съответната функция/процедура/тригер

We can optimize with User-defined Functions.
Transactions improve security and consistency.
Stored Procedures encapsulate repetitive logic.
Triggers execute before certain events on tables.

7.1. User-Defined Functions

Encapsulating Custom Logic

- Extend the functionality of a MySQL Server
  - **Modular=functional** programming – write **once**, call it **any number** of times
  - Faster execution – doesn't need to be reparsed and reoptimized with each use
  - Break out complex logic into **shorter code blocks**
- Functions can be:
  - Scalar – return **single value** or **NULL**
  - Table-Valued – return a **table**

Само при използване на **DECLARE** използваме за присвояване/задаване на стойност **:=**

DRY – Don't Repeat Yourself принцип

Creating Functions

DELIMITER $$$$ - **начало на разделител**
CREATE FUNCTION ufn_count_employees_by_town(`town_name` **VARCHAR(20)**)
RETURNS DOUBLE - **какъв тип връща функцията, незадължителен елемент**
DETERMINISTIC – при едни и същи входни данни един и същи резултат връща

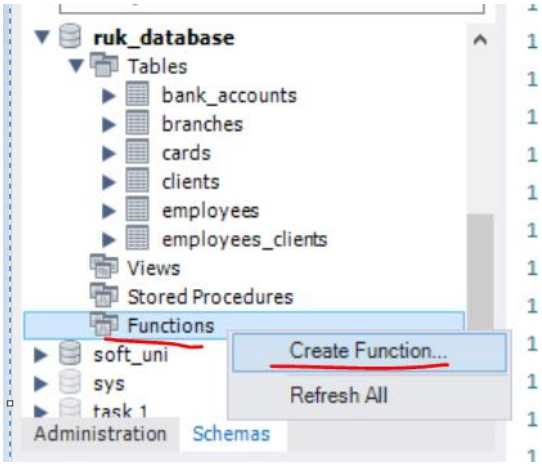
```
BEGIN
        DECLARE e_count DOUBLE;   - декларирване на променлива
        SET e_count := (SELECT COUNT(employee_id) FROM employees AS e    -  за присвояване
        INNER JOIN addresses AS a ON a.address_id = e.address_id
        INNER JOIN towns AS t ON t.town_id = a.town_id
        WHERE t.name = town_name);

        RETURN e_count;   - каква стойност връща функцията, незадължителен елемент
END;
$$$$ - край на разделител

DELIMITER $$$$
CREATE FUNCTION ufn_count_employees_by_town(`town_name` VARCHAR(20))
RETURNS INT
DETERMINISTIC
BEGIN
        DECLARE e_count INT;
        SET e_count := (SELECT COUNT(e.`employee_id`) FROM `employees` AS e
        INNER JOIN `addresses` AS a ON a.`address_id` = e.`address_id`
        INNER JOIN `towns` AS t ON t.`town_id` = a.`town_id`
        WHERE t.`name` = `town_name`);

        RETURN e_count;


Или директно връщаме резултата
        RETURN(
        (SELECT COUNT(e.`employee_id`) FROM `employees` AS e
        INNER JOIN `addresses` AS a ON a.`address_id` = e.`address_id`
        INNER JOIN `towns` AS t ON t.`town_id` = a.`town_id`
        WHERE t.`name` = `town_name`));

END;
$$$$
```

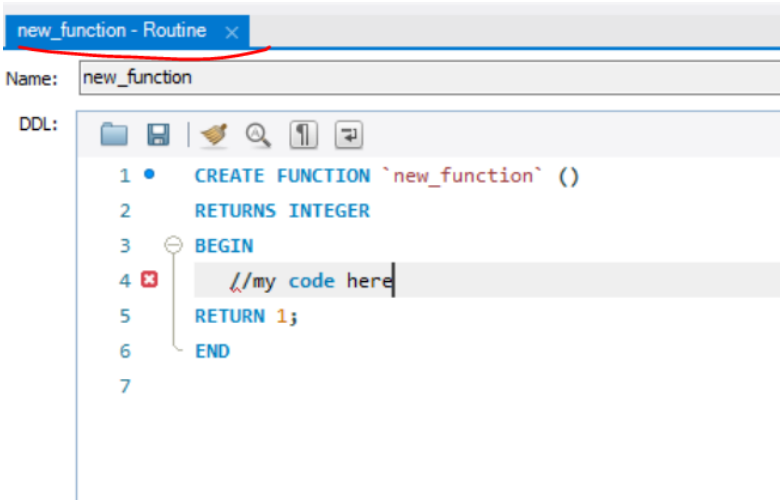
Deterministic vs. non-deterministic functions

Executing and Dropping Stored Functions

И така изпълняваме функцията

```
SELECT ufn_count_employees_by_town('Sofia');


SELECT ufn_get_salary_level(51000.0);
DROP FUNCTION ufn_get_salary_level;
```





7.2. Stored Procedures

Sets of Queries Stored On DB Server

- Stored procedures are logic removed from the application and placed on the database server.
  - Can greatly cut down traffic on the network
  - Improve the security of the database server
  - Separate data access routines from the business logic
- Stored procedures are accessed by programs using different platforms and API's.
- **Нямаме return при процедури**

Creating Stored Procedures

```
DELIMITER %%
CREATE PROCEDURE usp_select_employees_by_seniority()
BEGIN
  SELECT *
  FROM employees
  WHERE ROUND((DATEDIFF(NOW(), hire_date) / 365.25)) < 15;
END %%
```

Executing and Dropping Stored Procedures

- Executing a stored procedure by **CALL**

```
CALL usp_select_employees_by_seniority();
```

- **DROP PROCEDURE**

DROP PROCEDURE usp_select_employees_by_seniority;

Defining Parameterized Procedures

- To define a parameterized procedure use the syntax:

CREATE PROCEDURE usp_procedure_name
(parameter_1_name parameter_type,
parameter_2_name parameter_type,…)

Пример 1:

```
DELIMITER $$
CREATE PROCEDURE usp_select_employees_by_seniority(min_years_at_work INT)
BEGIN
 SELECT first_name, last_name, hire_date,
  ROUND(DATEDIFF(NOW(),DATE(hire_date)) / 365.25,0) AS 'years'
 FROM employees
 WHERE ROUND(DATEDIFF(NOW(),DATE(hire_date)) / 365.25,0) > min_years_at_work
 ORDER BY hire_date;
END $$

CALL usp_select_employees_by_seniority(15);
```

Пример 2:

```
Task 2 -Employees Promotion
DELIMITER $
CREATE PROCEDURE usp_raise_salaries (dept_name VARCHAR(45))
BEGIN
/*business logic*/
        UPDATE `employees` AS e
  JOIN `departments` AS d
  ON d.`department_id` = e.`department_id`
  SET e.`salary` = e.`salary` * 1.05
  WHERE d.`name` = dept_name;

  SELECT e.`first_name`, e.`salary` FROM `employees` AS e
  JOIN `departments` AS d
  ON d.`department_id` = e.`department_id`
  WHERE d.`name` = dept_name
  ORDER BY e.`first_name`, e.`salary`;

END $

CALL usp_raise_salaries('Finance');
```

Пример 3:

```
DELIMITER %%%
CREATE PROCEDURE usp_get_towns_starting_with (starts_with VARCHAR(20))
BEGIN
        SELECT `name` FROM `towns`
        WHERE `name` LIKE concat(starts_with, '%')  -  тук конкатенираме
  ORDER BY `name`;
```

END; %%%

CALL usp_get_towns_starting_with('S');

Returning Values Using OUTPUT Parameters

```
DELIMITER $$
CREATE PROCEDURE usp_add_numbers (first_number INT, second_number INT, OUT result INT)
BEGIN
  SET result = first_number + second_number;
END $$
```

Все едно процедурата работи като функция:

```
SET @answer=0;
CALL usp_add_numbers(5, 6,@answer);
SELECT @answer;
```

Функции и процедури рядко се използват – те са бизнес логика, която не е редно да стои при базата данни на сървъра.

7.3. Transactions – a kind of a stored procedure

- Една процедура има една или няколко SQL заявки, които се изпълняват една след друга.
- При транзакции, може да се създаде логика, която да изпълнява една или друга заявка или всички заявки, ИЛИ никоя от заявките да не се изпълни
- Transaction is a **sequence of actions SQL queries** (database operations) executed as a whole
  - Either **all** of them complete successfully or **none** of them
- Example of transaction
  - A bank transfer from one account into another (withdrawal + deposit)
    - If either the withdrawal or the deposit fails **the whole operation is cancelled**
- Транзакцията се пише вътре в процедура(stored procedure)
- Ако имаме много заявки, и ако някоя се чупи, то можем да зададем ROLLBACK и на всички изпълнени до момента транзакции

ACID model is used by InnoDB engine на MySQL

Transactions Behavior

- Transactions guarantee the **consistency** and the **integrity** of the database.
  - All changes in a transaction are temporary
  - Changes are persisted when **COMMIT** is executed – всичко е временно докато не commit-нем

COMMIT е зададен като default - да

  - At any time all changes can be canceled by **ROLLBACK**
- All of the operations are executed as a whole.

Само намаля в случая:

```
START TRANSACTION;
    UPDATE `employees` SET `salary` = `salary` - 1000 WHERE `employee_id` = 1;
```

Отново само намаля в случая:

```
START TRANSACTION;
	UPDATE `employees` SET `salary` = `salary` - 1000 WHERE `employee_id` = 1;
COMMIT;
```

Намалянето не се извършва:

```
START TRANSACTION;
	UPDATE `employees` SET `salary` = `salary` - 1000 WHERE `employee_id` = 1;
ROLLBACK;
```

#Task 3

```
DELIMITER %%
CREATE PROCEDURE usp_raise_salary_by_id(id int)
BEGIN
	DECLARE does_exist INT;
	START TRANSACTION;
	 UPDATE employees SET salary = salary *1.05 WHERE employee_id = id; - update-ни всички, но не записвай нищо още реално

	SET does_exist := (SELECT COUNT(*) FROM employees WHERE employee_id = id);
	IF (does_exist = 1)
	THEN COMMIT; - сега ги презапиши!!!
	ELSE
		ROLLBACK;
	END IF;
END %%

CALL usp_raise_salary_by_id(1);
```

Modern DBMS servers have built-in transaction support

- Implement "ACID" transaction
- hgfds

- ACID means:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

#13. Withdraw Money

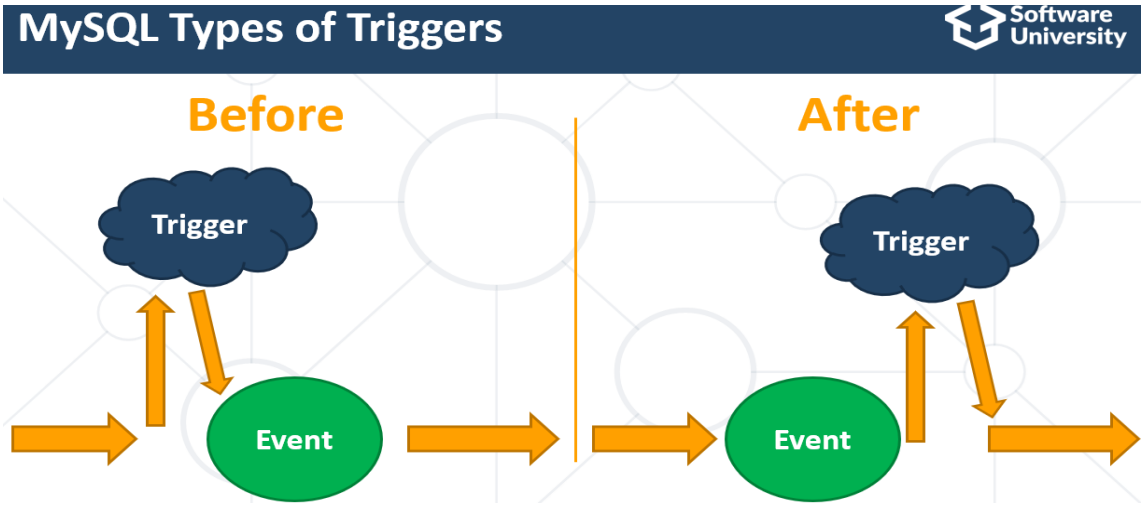
```
DELIMITER $$$
```

```
CREATE PROCEDURE usp_withdraw_money(account_id INT, money_amount DECIMAL(19, 4))
BEGIN
DECLARE bal DECIMAL(19, 4);
START TRANSACTION;
        SET bal := (SELECT a.`balance` FROM `accounts` AS a WHERE a.`id` = account_id) - money_amount;

        UPDATE `accounts`
   SET `balance` = `balance` - money_amount
   WHERE `id` = account_id;

   IF (money_amount > 0 AND bal > 0.0000) THEN COMMIT;
   ELSE ROLLBACK;
   END IF;
END;
$$$

DROP PROCEDURE usp_withdraw_money;

CALL usp_withdraw_money(1, 20);
```

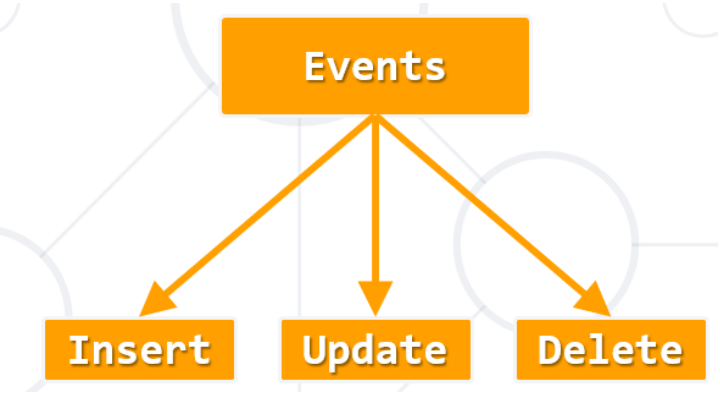
7.4. Triggers – като Event Listener

Event-listener – чакаме нещо да се случи, и тогава прави нещо

- Triggers - small programs in the database itself, activated by the database events application layer
  - UPDATE, DELETE or INSERT queries
  - Called in case of specific **event**
- We do not call triggers **explicitly**
- Triggers are **attached** to a table



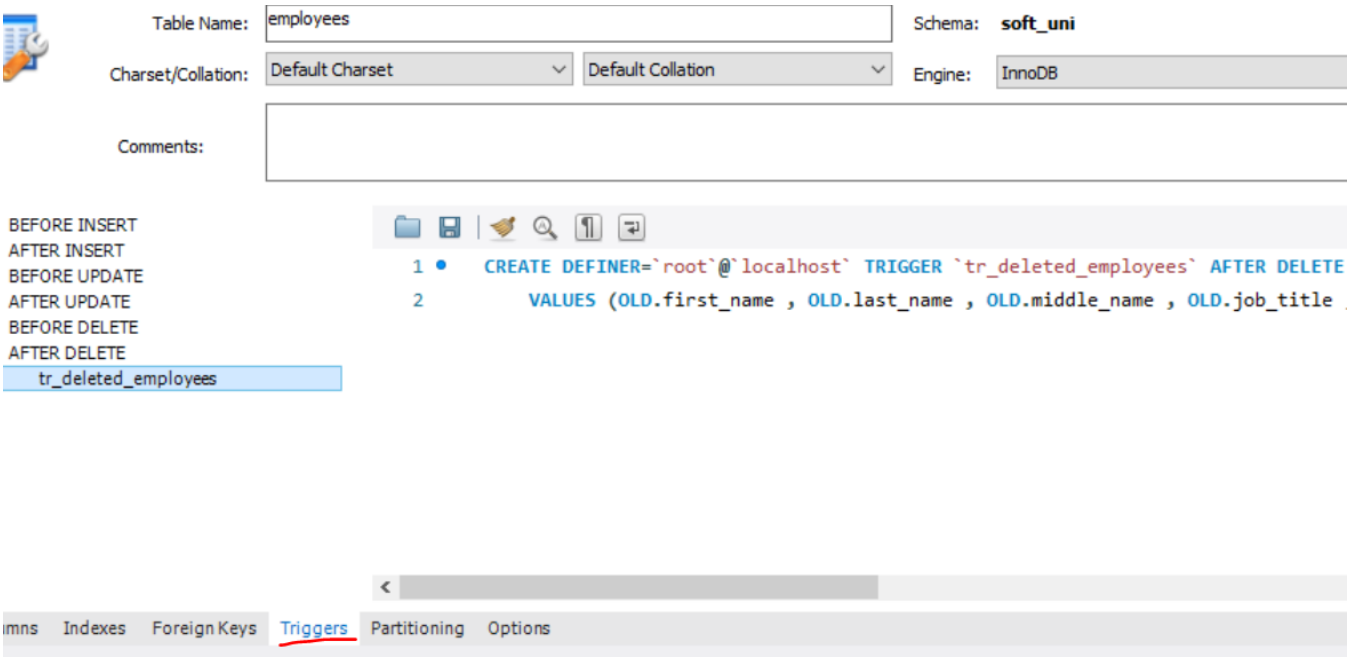
There are three different events that can be applied within a trigger:



The OLD and NEW keywords allow you to access columns before/after trigger action
OLD – **before trigger action – използва се при AFTER events**
NEW – **after trigger action – използва се при BEFORE events**

After Delete

```
CREATE TABLE deleted_employees(
        employee_id INT PRIMARY KEY AUTO_INCREMENT,
        first_name VARCHAR(20),
        last_name VARCHAR(20),
        middle_name VARCHAR(20),
        job_title VARCHAR(50),
        department_id INT,
        salary DOUBLE
);

CREATE TRIGGER tr_deleted_employees
AFTER DELETE
ON employees
FOR EACH ROW
INSERT INTO deleted_employees     (first_name,last_name,middle_name,job_title,department_id,salary)
VALUES(OLD.first_name, OLD.last_name, OLD.middle_name, OLD.job_title, OLD.department_id, OLD.salary);
```



Before Insert

CREATE TRIGGER trigger_employee
BEFORE INSERT
ON `employees`
FOR EACH ROW
BEGIN
END;

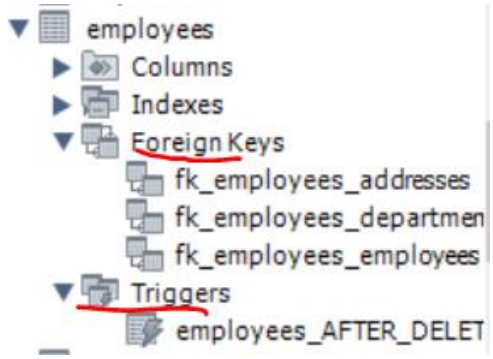
After Update

CREATE DEFINER = CURRENT_USER **TRIGGER** `employee_AFTER_UPDATE` **AFTER UPDATE**
ON `employees`
FOR EACH ROW
INSERT INTO addresses_archive (old_salary, new_salary) VALUES (**OLD.salary**, **NEW.salary**)
INSERT INTO LOGS addresses_archive (old_salary, new_salary) VALUES (**OLD.salary**, **NEW.salary**)

За по-лесно, може да ползваме и този прозорец

#15. Log Accounts Trigger
CREATE TABLE `logs`(
`log_id` INT PRIMARY KEY AUTO_INCREMENT,

```
`account_id` INT NULL,
`old_sum` DECIMAL(19, 4) NOT NULL,
`new_sum` DECIMAL(19, 4) NOT NULL);


DELIMITER $$
CREATE TRIGGER `tr_balance_updated`
AFTER UPDATE ON `accounts`
FOR EACH ROW
BEGIN  -  пишем го след FOR EACH ROW
        IF OLD.`balance` != NEW.`balance` THEN
        INSERT INTO `logs`(`account_id`, `old_sum`, `new_sum`) VALUES (OLD.`id`, OLD.`balance`, NEW.`balance`);
        END IF;
END;$$

DROP TRIGGER `tr_balance_updated`;

Тази проверка е излишна, освен разбира се ако има промяна не в баланса, а в други данни
#IF OLD.`balance` != NEW.`balance` THEN
#END IF;
```




```
CREATE TABLE IF NOT EXISTS `accounts` (
  `id` int(11) NOT NULL,
  `account_holder_id` int(11) NOT NULL,
  `balance` decimal(19,4) DEFAULT '0.0000',
  PRIMARY KEY (`id`),
```

```
CREATE TABLE logs(
        log_id INT PRIMARY KEY AUTO_INCREMENT,
        account_id INT,
        old_sum DECIMAL(19, 4),
        new_sum DECIMAL(19, 4)
);
```

CREATE DEFINER=`root`@`localhost` **TRIGGER** `accounts_AFTER_UPDATE`
AFTER UPDATE
ON `accounts` **FOR EACH ROW**
INSERT INTO **`logs`** (`account_id`, `old_sum`, `new_sum`) - **вкарай в новата таблица `logs`**
VALUES (OLD.`id`, OLD.`balance`, NEW.`balance`); - **вземи данни от таблица `accounts` където правим промени**

7.5. Глобални настройки / стойности

За смяна на root паролата

```
ALTER USER 'root'@'localhost' IDENTIFIED BY '';
flush privileges;
```

За проверка свързана с външни ключове

```
#Игнорирай външните ключове, за да можем да си трием спокойно :)
SET FOREIGN_KEY_CHECKS = 0;

#Имай в предвид външните ключове
SET FOREIGN_KEY_CHECKS = 1;
```

Общи

```
SET GLOBAL log_bin_trust_function_creators = 1;  //да се доверявам на функциите ако аз съм ги създал
SET SQL_SAFE_UPDATES = 0;   //да махнем safe updates


SELECT @@sql_mode;

SET sql_mode = 'ONLY_FULL_GROUP_BY';   - активен този mode
SET sql_mode = '';- премахни only full group by – да го използваме в Judge, но няма ефект ☹
```

Има промяна в Judge – Judge работи вече само в mode 'ONLY_FULL_GROUP_BY'
Когато групираме по PRIMARY KEY, няма проблеми обаче!!!
Друг вариант е да използваме вложени заявки и да не използваме GROUP BY!!!

GROUP_CONCAT
ANY_VALUE(muhaha) **AS** `bla_bla_bla`;

8. Нормализация на базите данни

https://bg.myservername.com/database-normalization-tutorial

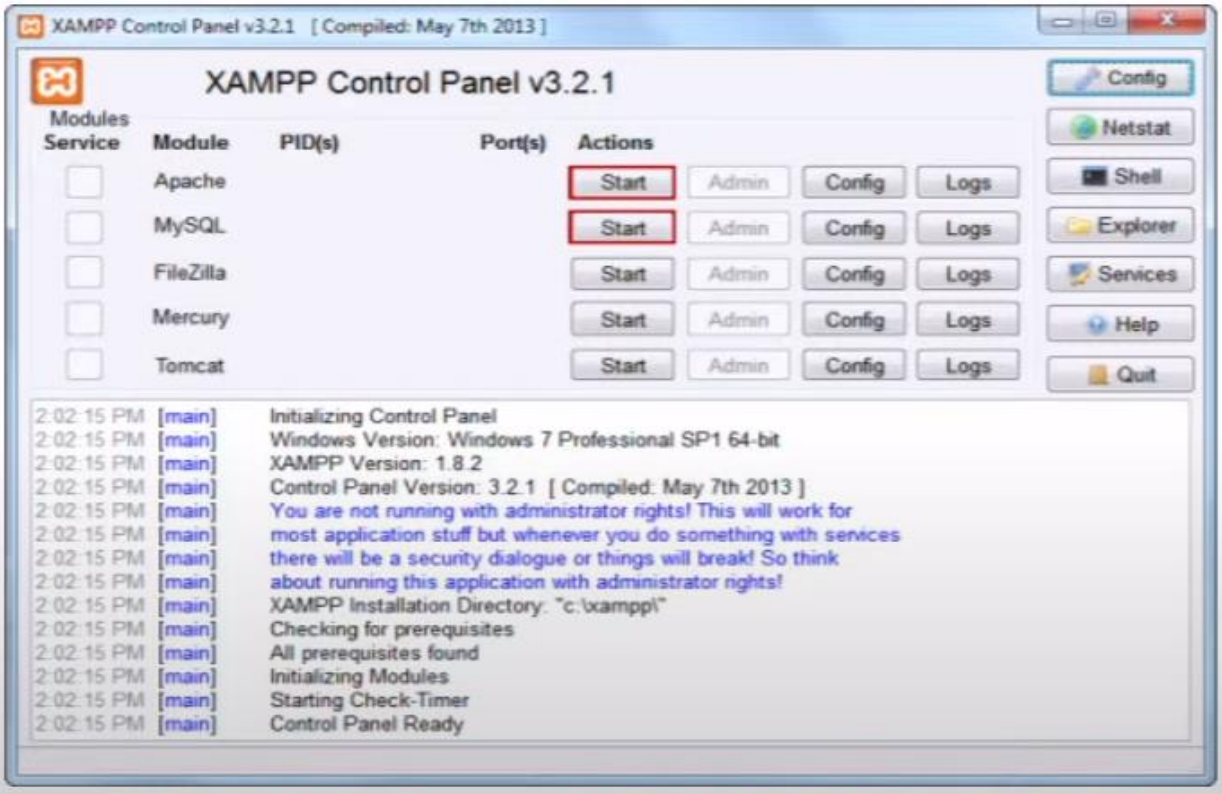
9. Other

Ако искаме, можем да ползваме и XAMPP или MariaDB или Heidi SQL вместо MySQL

Using XAMPP Control Panel

By B Lingafelter Feb 13, 2014 testserver, workflow

1. Open the XAMPP Control Panel. If you don't have a Desktop or Quick Launcl **XAMPP Control Panel**.



2. Click **Start** button next to Apache. **Note: Do NOT mark the Service check b**