I. Databases

What is database

- Organize collection of data
- Two types of databases:
  - Operational database - collect, modify and maintain data on daily basis.
  - Analytical database - mainly for historical and time-dependant data
- Based on two mathematical theories
  - Set theory
  - First-order predicate

Relational database Model

Based on the concept of "relation":

- **Relation/Table:** a relation (table) is a set of tuples
- **Tuple/Row:** a tuple is an individual row in a relation. Each tuple represents a single record
- **Attribute/Column:** An attribute is a named column of a relation
- **Degree:** The degree of a relation is the number of attributes it contains
- **Cardinality:** Cardinality refers to the number of tuples in a relation - колко реда има таблицата ни
- **Relational Algebra:** Procedural query language for the relational model

Domain:

- Refers to the set of allowable values that a column (attribute) can contain.
- It's the type of data that can be stored in a certain column
- Example: a domain of a `salary` column could be all real numbers, or the domain of a `gender` column could be 'male', 'female', 'prefer not to say'

Relation:

- Table in a relational database
- Set of tuples that have the same attributes
- A relation is defined by its name and the set of attributes it contains

Relational database Schema

- Describes the structure of a database in a formal way
- Provides definitions for tables, fields, relationships, views, indexes, packages, procedures, functions, queues, triggers, sequences, synonyms and other objects
- Defines constraints over the database objects
- In many cases normalization is a key requirement for a database schema

Data normalization

Нормализация на базите данни

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

- What is normalizaton
- Normal forms
  - **1NF** - **Used to eliminate repeating groups**

  First normal form means the following conditions are met:
  - Eliminate repeating groups in individual tables
  - Create a separate table for each set of related repeated data
  - Identify each set of related data with Primary key

Do not use multiple fields in a single table to store similar data:

1NF

employee_id	name	job_code	job	city
21313123	Spas Belev	S03	Software Engineer	Sofia
45344353	Zhorzh Raychev	S03	Software Engineer	Plovdiv
98658699	Ivan Ivanov	T01	Technical Lead	Sofia
234656643	Georgi Georgiev	P01	Product Owner	London

- **2NF** - **Used to eliminate redundant data**

Second normal form means the following conditions are met:

- Create separate tables for sets of values that apply to multiple records
- Relate these tables with a foreign key

Records should not depend on anything other than a table's Primary key:

employee_id	name	job_code	city
21313123	Spas Belev	S03	Sofia
45344353	Zhorzh Raychev	S03	Plovdiv
98658699	Ivan Ivanov	T01	Sofia
234656643	Georgi Georgiev	P01	London

jobs

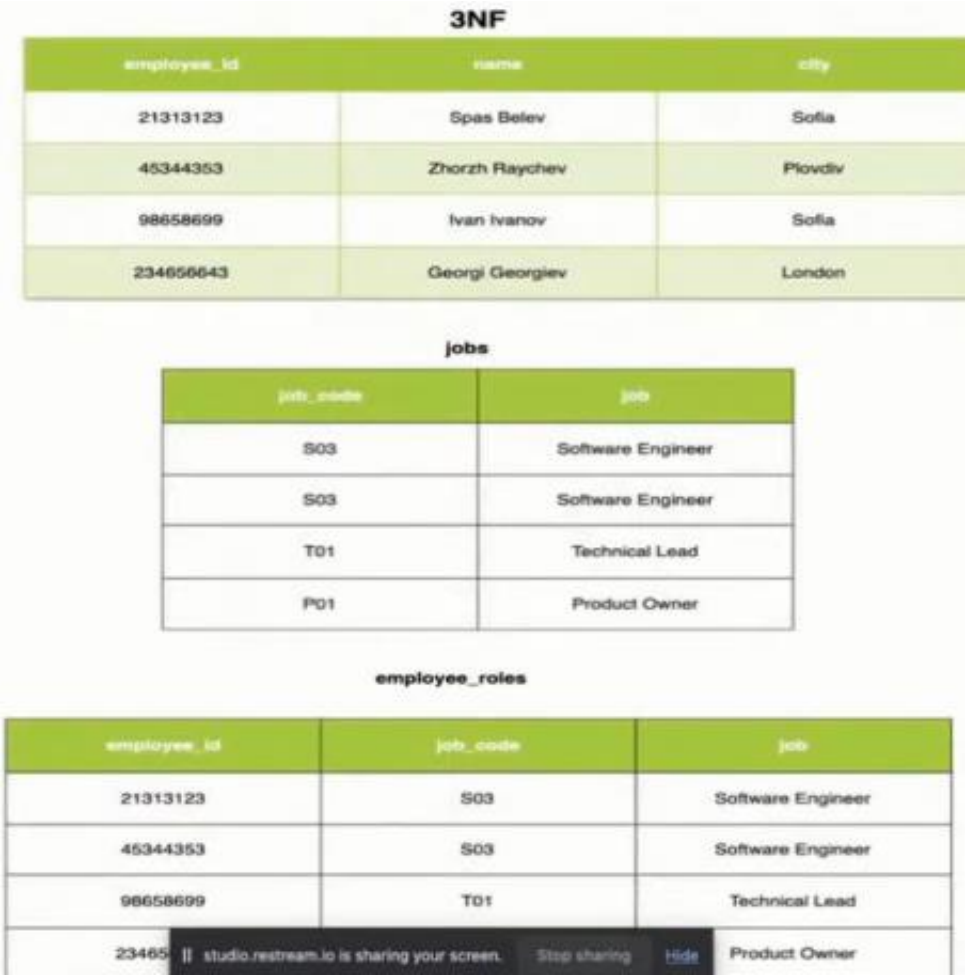
job_code	job
S03	Software Engineer
S03	Software Engineer
T01	Technical Lead

- **3NF** - **Used to eliminate columns that are not dependent on the key of the table**

Third normal form checks for **transitive dependencies**:

- Eliminate fields that do not depend on the key

Records that are not a part of the record's key do not belong in the table



3NF

employee_id	name	city
21313123	Spas Belev	Sofia
45344353	Zhorzh Raychev	Plovdiv
98658699	Ivan Ivanov	Sofia
234656643	Georgi Georgiev	London

jobs

job_code	job
S03	Software Engineer
S03	Software Engineer
T01	Technical Lead
P01	Product Owner

employee_roles

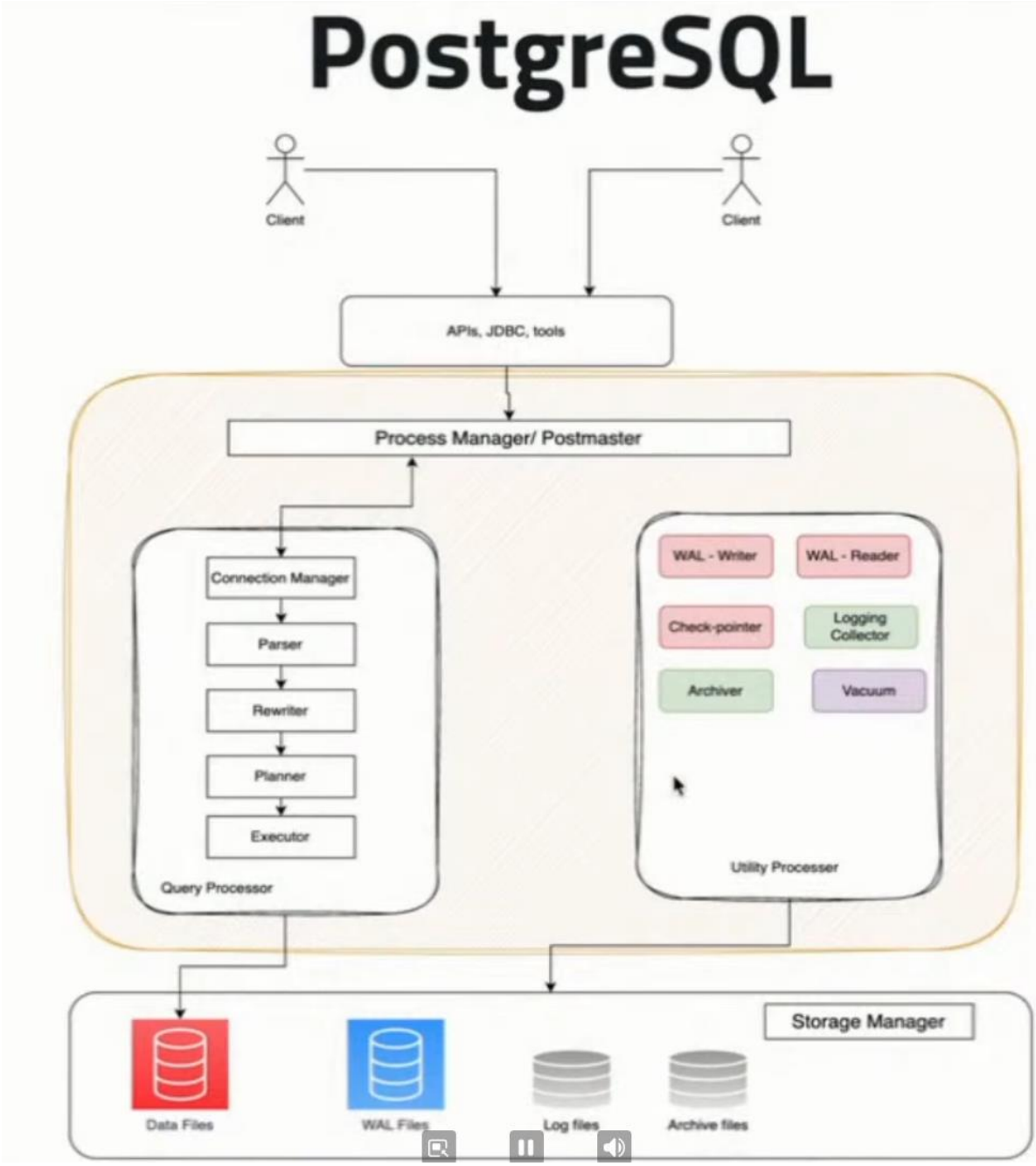
employee_id	job_code	job
21313123	S03	Software Engineer
45344353	S03	Software Engineer
98658699	T01	Technical Lead
23465	S01	Product Owner

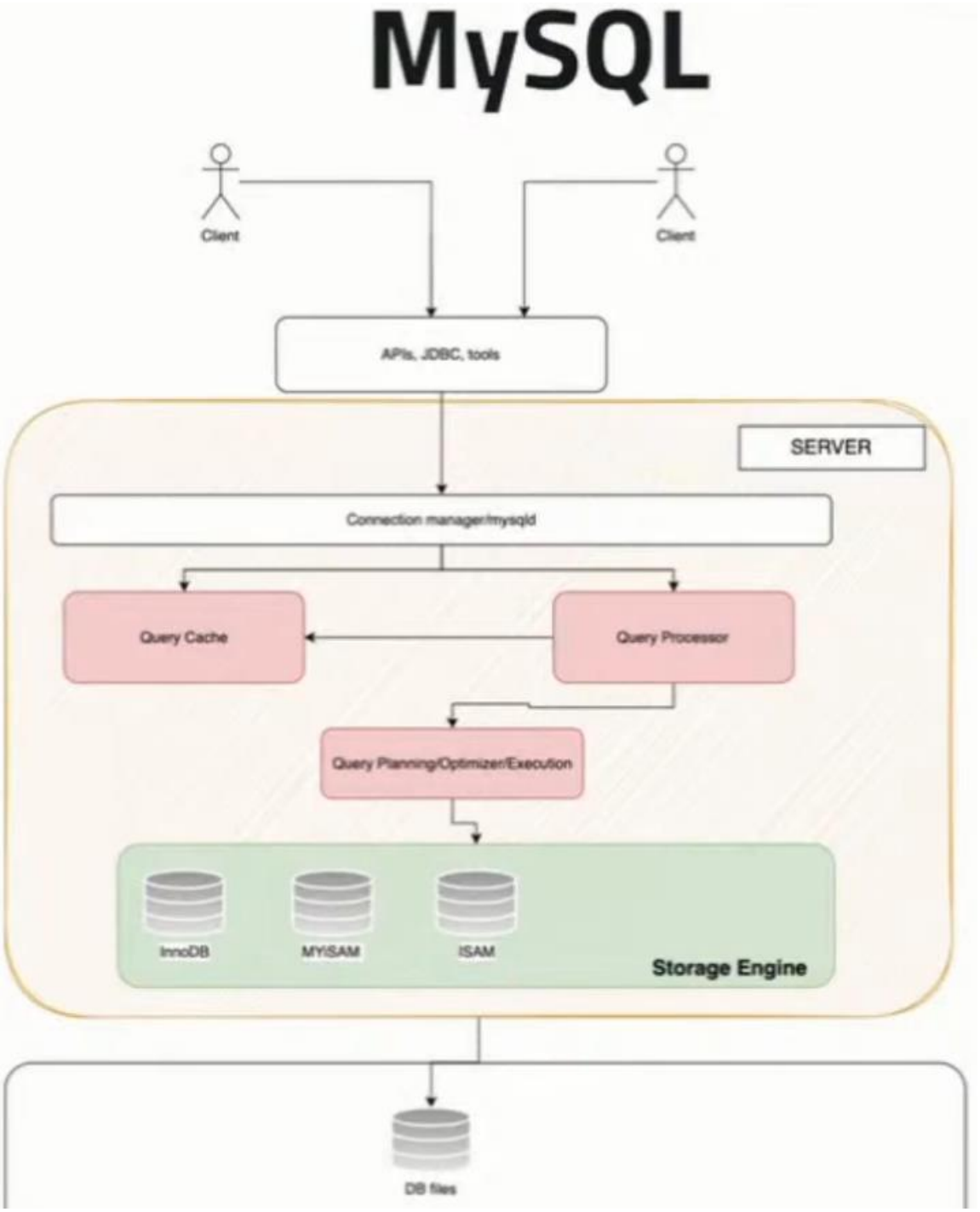
- 4NF - Isolate Independent Multiple Relationships
- 5NF - Isolate semantically related Multiple Relationships

Как работи отдолу една RDBMS

Writer
Reader
Archiver
Vacuum

PostgreSQL


Client
Client
APIs, JDBC, tools
Process Manager/ Postmaster
Connection Manager
Parser
Rewriter
Planner
Executor
Query Processor
WAL - Writer
WAL - Reader
Check-pointer
Logging Collector
Archiver
Vacuum
Utility Processer
Storage Manager
Data Files
WAL Files
Log files
Archive files



Indexes

- Improve the speed of data retrieval operations on a database table at the cost of additional writes and storage space
- Indexes are used to quickly locate data without having to search every row
- Indexes can be created using one or more columns of a database table.
- The **primary key is called also clustered index**. All other indexes are called non-clustered indexes.

Key points about indexes:

- **Index Types:** There are several types of indexes, such as B-tree, bitmap, and hash indexes, each suited to particular types of queries
- **Composite index:** An index with 2 or more columns at the same time
- **Index Key:** The column(s) on which the index is created

- **Primary key:** A type of constraint used in a table to uniquely identify each record in this table. It automatically creates an index known as a clustered index.
- **Unique index:** It ensures data present in the column is unique. It happens automatically. It allows one null value for a column!

Indexes и балансирано бинарно дърво

- Structures associated with a table or view that speeds retrieval of rows
  - Usually implemented as **B-trees (binary trees)** **– takes extra DB memory!!!**
- 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.

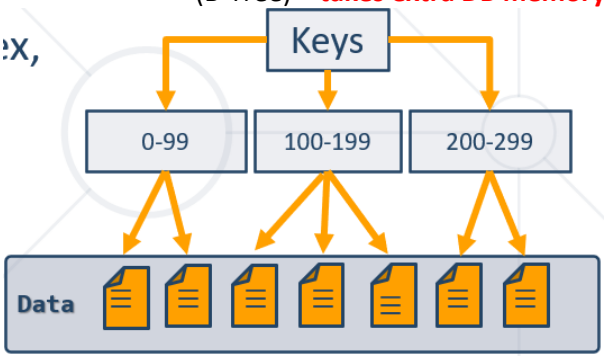
id PRIMARY KEY -> B-tree

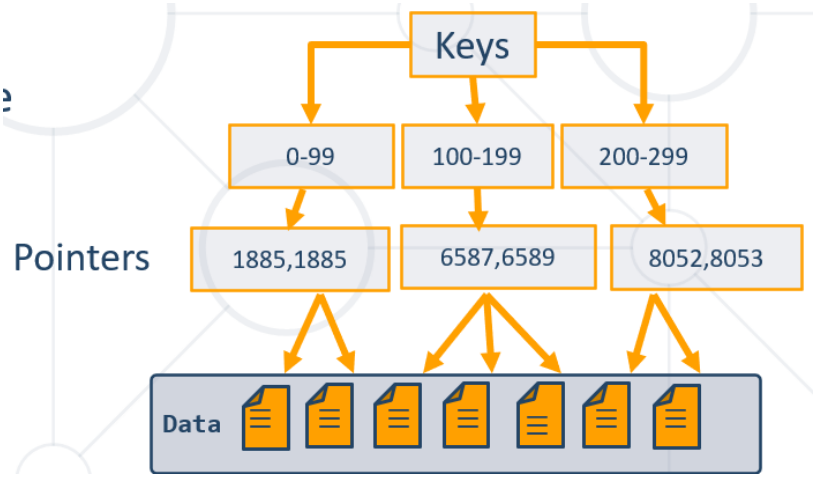
Clustered Indices

- **Clustered index(primary key) determine the order of data**
  - Very useful for fast execution of **WHERE**, **ORDER BY** and **GROUP BY** clauses
- Maximum 1 clustered index per table

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
  - If a table has no clustered index, its data rows are stored in an **unordered structure** (heap), binary tree (B-Tree) **– takes extra DB memory!!!**





Синтаксис на индекс

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

Stored procedures

- Group of pre-compiled SQL statements that are stored in the database
- Can be invoked (or called) by:
  - User application programs
  - Triggers
  - Other stores procedures from within the database
- Encapsulate logic for data transformation, data validation, and business-specific logic

Benefits of using stored procedures:

- Performance: Because stored procedures are compiled and stored in the database, they tend to be more efficient and faster.
- Network traffic: using stored procedures can reduce network traffic between clients and servers
- Security: stored procedures provide an additional layer of security. You can grant users the permission to execute a stored procedure without giving them access to the underlying tables
- Code reuse and encapsulation: Logic created in stored procedures can be reused across multiple programs

In some cases, better in the Java code to be the logic and to have more network traffic!!!

Functions

- Precompiled SQL commands that are used to perform a specific task or operation and return a value
- Built-in(aggregate, Scalar, Date, and many more) or user-defined
- Много рядко може да се наложи да си правим наша си функция. В повечето случаи вградените функции са достатъчни

Design process

- Requirement Analysis: understand what the database will be used for, by who and what data it will store - ще пишем или ще четем повече от базата, и други подобни въпроси
- Conceptual Database Design: Identify the high-level entities and relationships among them. An E/R is often created.

- Logical Database Desing: Translate the conceptual model into a logical model, which is a more detailed and precise representation of the data
- Physical Database Design
- Database Implementation

Entities

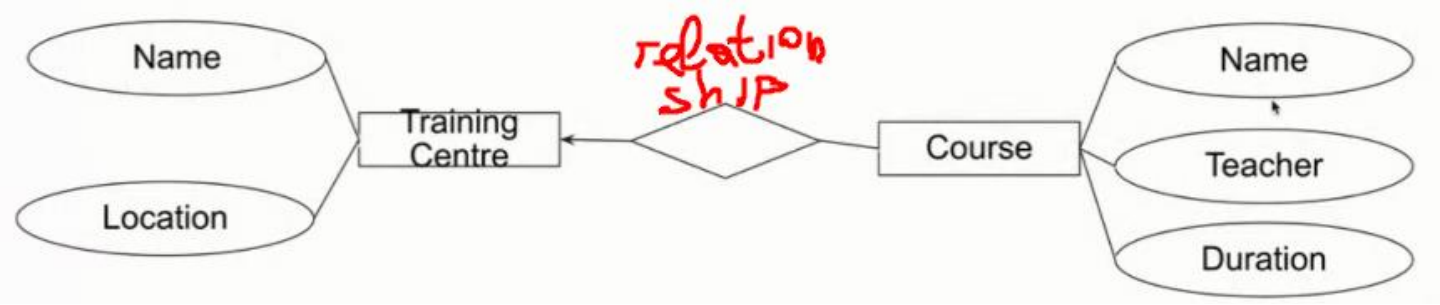
Entity - a definable thing - such as a person, object, concept or event - that can have data stored about it.

Strong entity - има си всички атрибути и е самодостатъчно/самоопределящо се. И няма нужда от foreign keys.
Weak entity - не може да се самоопредели от атрибутите си какво е точно
Associative entity - associates entities (or elements) within an entity set

Entity set: - same as entity type, but defined at a particular point in time, such as students enrolled in a class on the first day

Relationship

- How entities act upon each other or are associated with each other - One to Many, Many to One, Many to Many, One to One, etc.
- Relationships are typically shown as **diamonds** (on the ER diagrams) *or* labels directly on the **connecting lines**

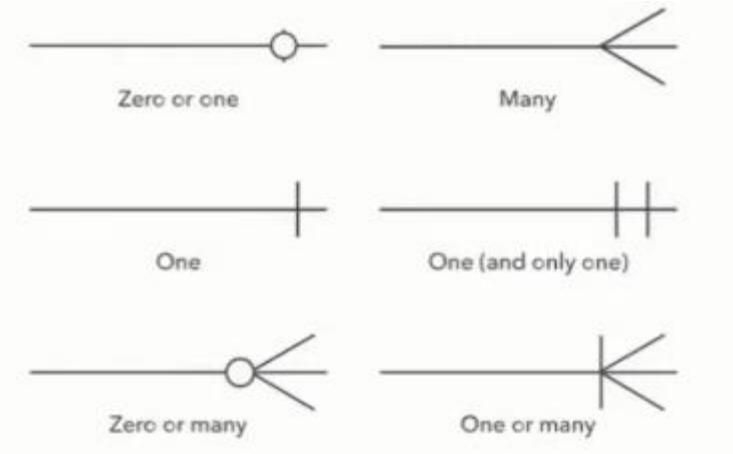


Attribute

- Attributes are categorized as simple, composite, derived, as well as single-value or multi-value:
  - Simple: attribute value is **atomic** and can't be further divided
  - Composite: Sub-attributes spring from an attribute
  - Derived: Attribute is calculated or otherwise derived from another attribute, such as age from a birthdate

Cardinality

- Defines the numerical attributes of the relationship between two entities or entity sets
- The three main cardinal relationships are one-to-one, one-to-many, and many-to-many



Tools for creating E\R or similar database diagrams (Data Modelling Tools):

- Lucidchart
- Microsoft Visio
- Draw.io
- MySQL Workbench
- ER/Studio
- https://dbdiagram.io/home
- IntelliJ Idea Ultimate
- pgAdmin
- https://dbeaver.io/

II. Installing MySQL

Installing

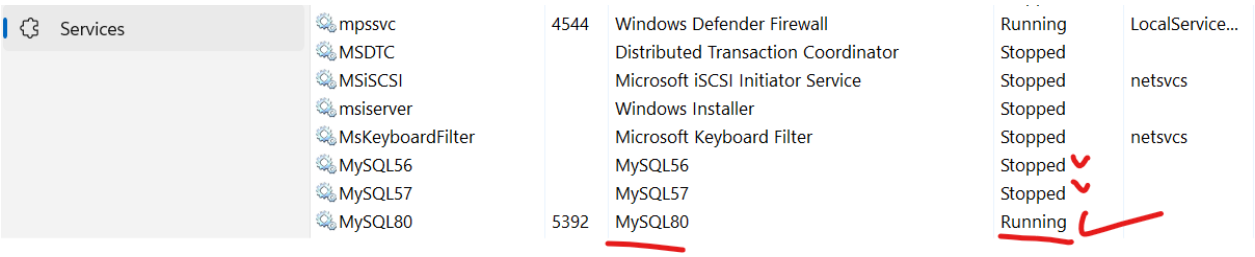
Please, see the installation manual from OneDrive, the first folder of MySQL course.

Stopping MySQL server/s on Windows

Когато сървър базата се инициализира в Докер/облачно за даден проект, то трябва да спрем ръчно съществуващия service (database server) на Windows.

При Windows за да пуснем контейнер, то трябва да използваме Docker Desktop (не знам дали има background приложение на Docker за Windows).

Докато при Линукс може и да не използваме Docker Desktop, а само с бекграунд приложение на Docker.



III. 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, PostgreSQL, 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

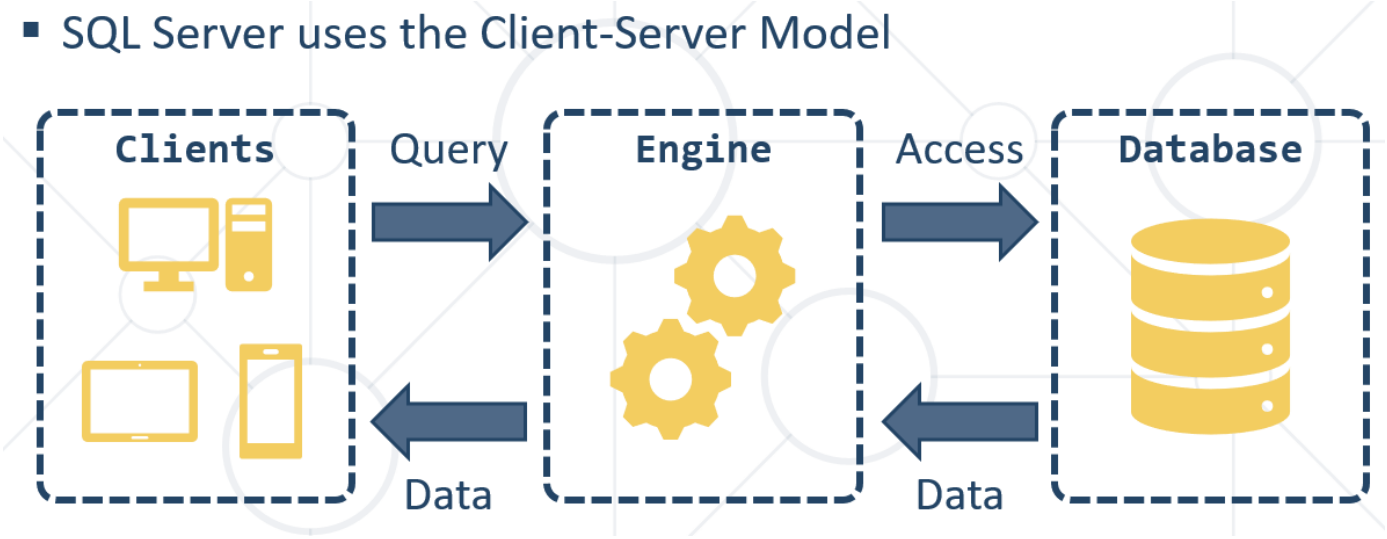
1.0. General info

- 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

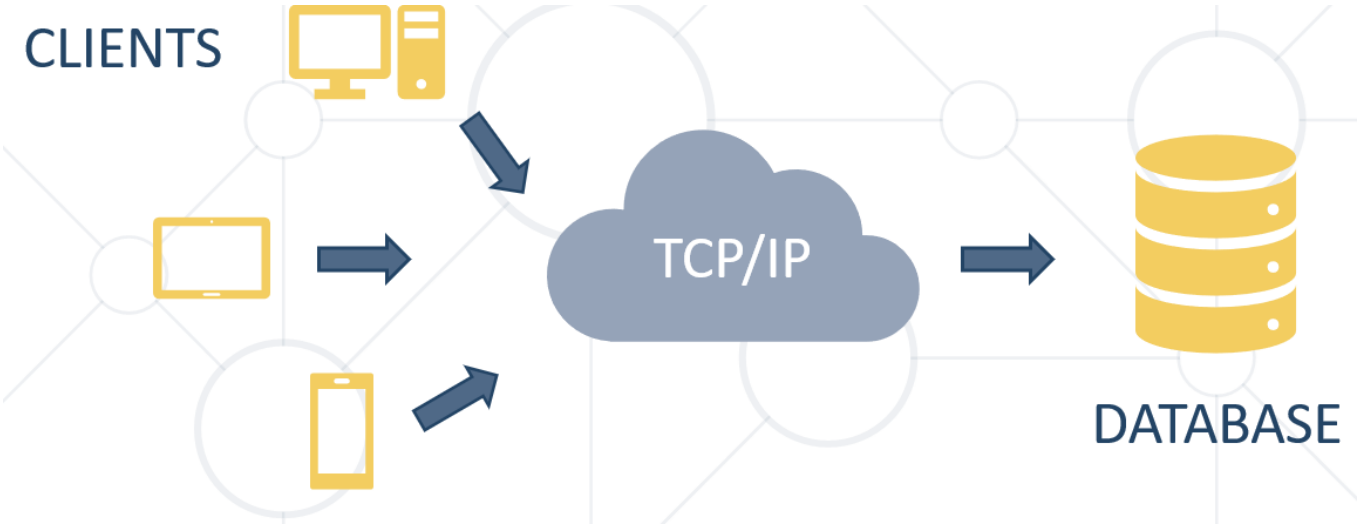
- **R**elational **D**ata **B**ase **M**anagement **S**ystem
  - Database **management**
  - It **parses requests** from the user and takes the **appropriate** action
  - The user **doesn't have direct access** to the stored data

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

1.1. Database Engine Flow



Client-Server Model



1.2. Structured Query Language = SQL

- Queries
- Clauses
- Expressions
- Predicates
- Statements

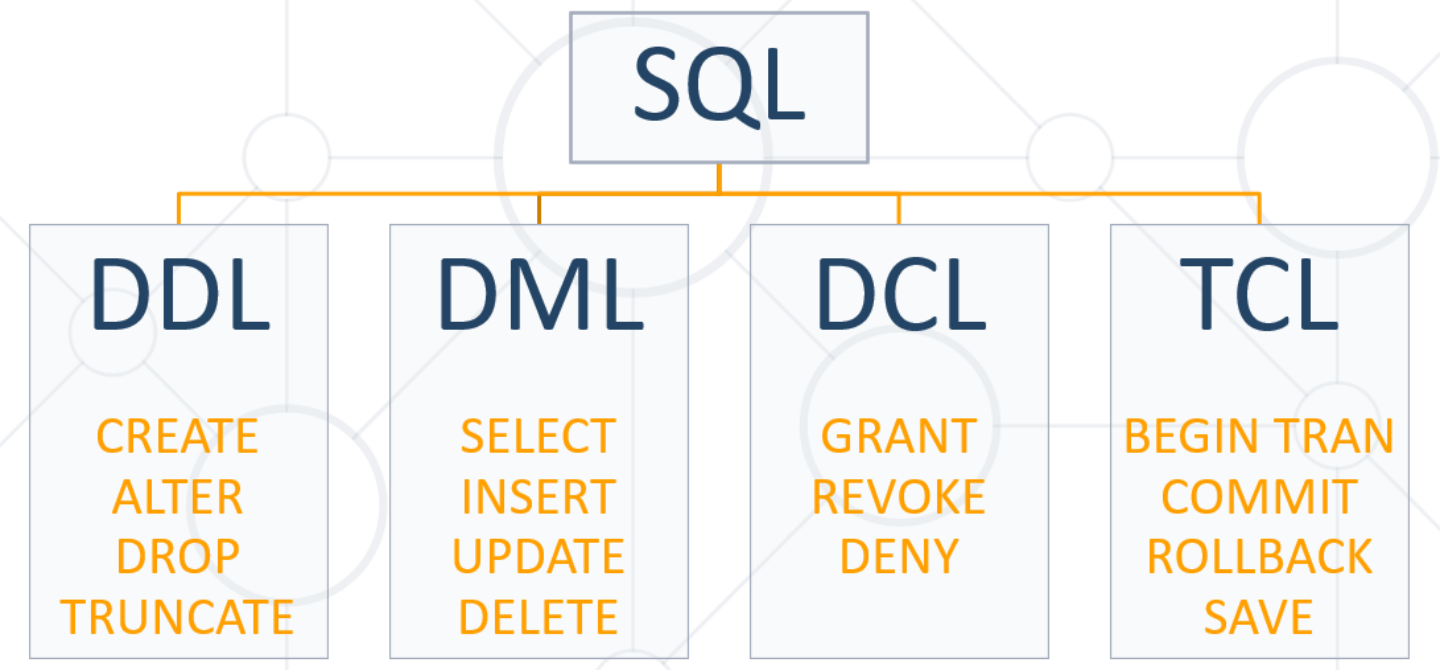
1.3. DDL, DML, DCL, TCL

Logically divided in four sections

- **Data Definition Language**– describe the structure of our data = **DDL**
- **Data Manipulation Language** – store and retrieve data = **DML**

CRUD is part of **DML** – Create, Read, Update, Delete

- **Data Control Language** – define who can access the data = **DCL**
- **Transaction Control Language** – bundle operations and allow rollback = **TCL**



1.3. DQL

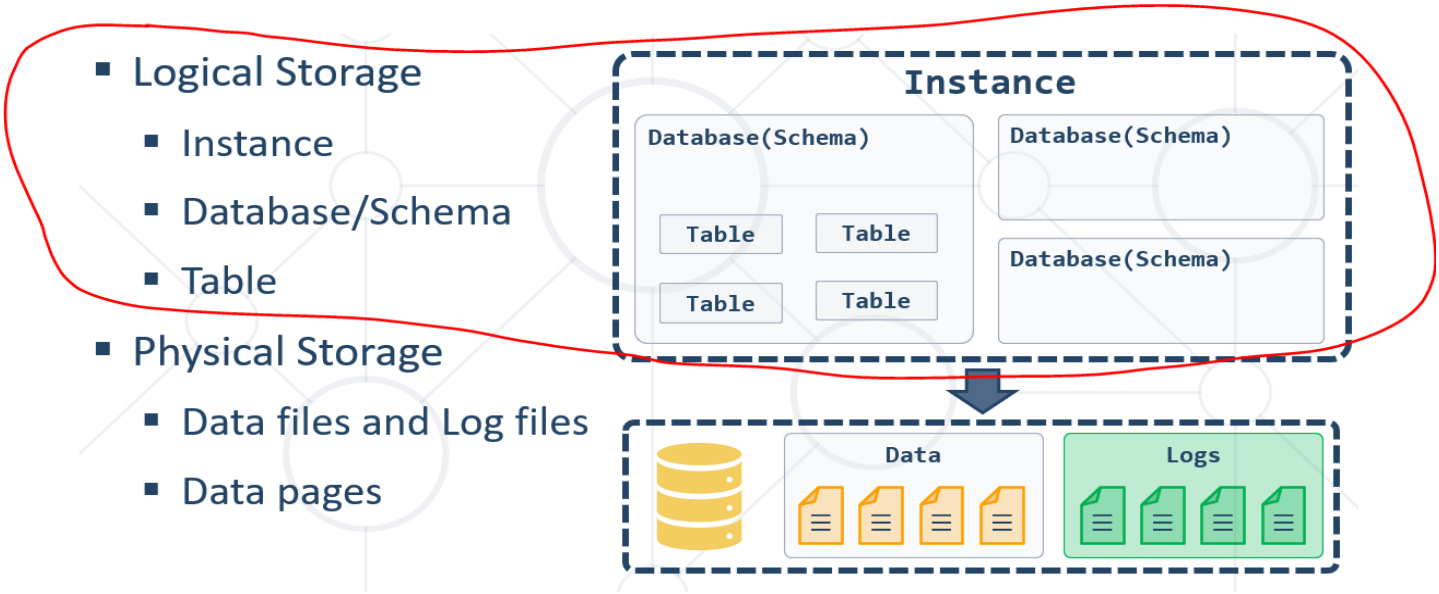
Data query language (DQL) is part of the base grouping of SQL sub-languages. These sub-languages are mainly categorized into:

- a data query language (**DQL**),
- a data definition language (DDL),
- a data manipulation language (DML).
- a data control language (DCL),

- Transaction control language (TCL)

DQL statements are used for performing queries on the data within schema objects. The purpose of DQL commands is to get the schema relation based on the query passed to it.
Although often considered part of DML, the SQL SELECT statement is strictly speaking an example of DQL. When adding FROM or WHERE data manipulators to the SELECT statement the statement is then considered part of the DML.

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/Indexes

- Indices make data lookup faster
  - Clustered – bound to the **primary key**, physically sorts data

- Non-Clustered – can be **any field**, references the primary index

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

- What are views?
  - Virtual tables that allow users to see data in the database without having to store it as a physical entity
  - Result of a SELECT query and can provide a simplified representation of a table or join
- Why use views?
  - Simplification – can hide complexity of data
  - Security – can grant access to specific part of the data
  - Consistency – consistent snapshot of the data
  - Data Abstraction – stay the same even if design changes

CREATE VIEW StudentNames **AS**
SELECT first_name, last_name **FROM** Students;

SELECT * FROM StudentNames;

Пример и с JOIN:

```
98•Create view students_info as
99 select s.student_id, s.name, c.title, e.grade
100 from students_7 s
101 join enrollments e on s.student_id = e.student_id
102 join classes_2 c on c.class_id  = e.class_id
```

- UPDATE <name> <command>
  - The view must derive from a single table
  - The view should encompass the PRIMARY KEY of the table
  - No calculated columns should exist in the view

This update reflects in the original 'Students' table
UPDATE StudentNames
SET first_name = 'John'
WHERE last_name = 'Doe';

- CREATE OR REPLACE VIEW <name> <command>

CREATE OR REPLACE VIEW StudentNames **AS**
SELECT first_name, last_name **FROM** Students

WHERE date_of_birth > '2000-01-01';

- DROP VIEW <name>

Данните не се изтриват от оригиналната таблица
DROP VIEW StudentNames;

- Views are divided into 2 types:
  - **Simple** views – can be used in almost all cases/queries
  - **Complex** views – limited to a few allowed usage cases (for example a complex view with JOIN may not be allowed to be used by a certain query – the DB will throw exception and we would understand a specific operation is not possible)

Usually, a view is a Virtual table, and as a such it corresponds to a certain SELECT query.
In OracleDB for example, there are also **materialized views** – there the view itself is stored in a separate db table!

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 characters (i.e. Japanese, Korean etc.)	Non-Unicode variable length	Unicode variable length can store both non-Unicode and 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

Info

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

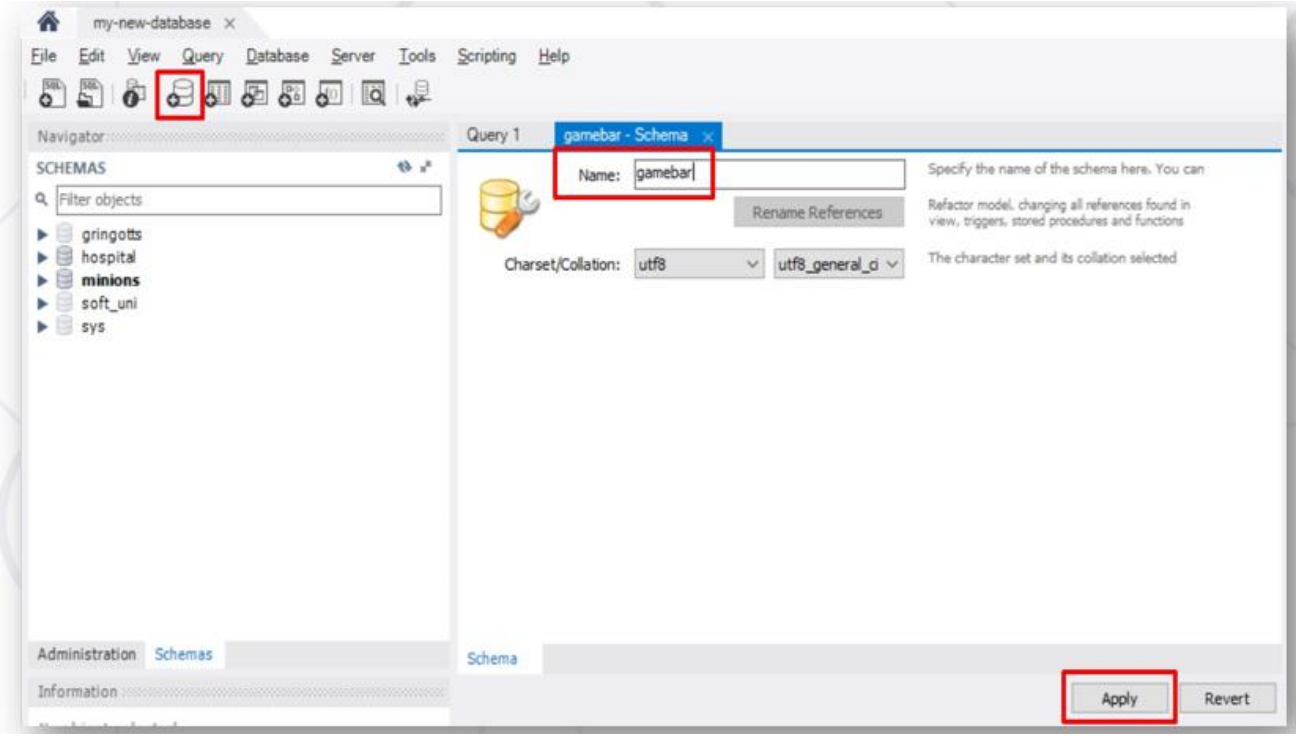
- A table is created with the CREATE TABLE statement
- Constraints may be specified:
    - As part of the column definitions
    - In the CREATE TABLE statement (outside the column definitions)
    - Outside the CREATE TABLE statement, using ALTER TABLE statements

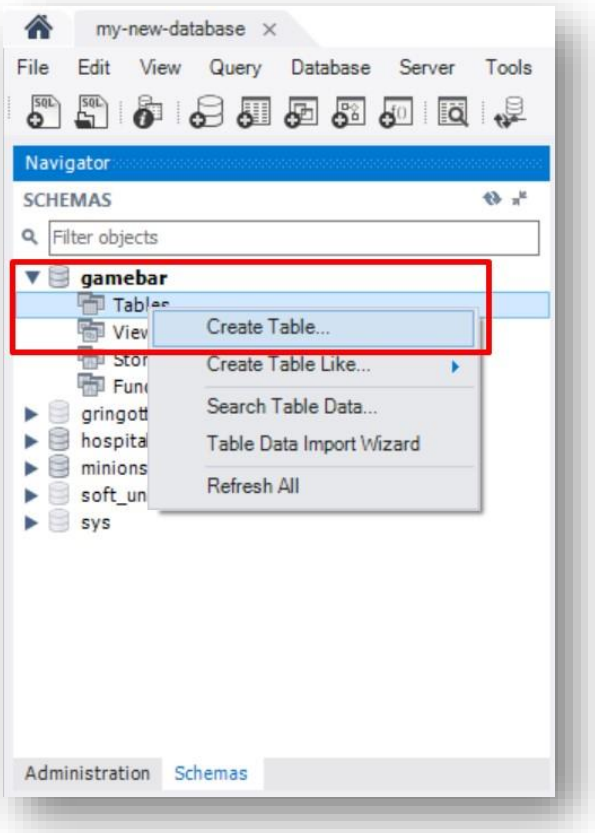
- Types of constraints:
    - NOT NULL (in MySQL it is not defined as a constraint)
    - UNIQUE
    - PRIMARY KEY
    - FOREIGN KEY
    - CHECK (not enforced by MySQL)

Using GUI

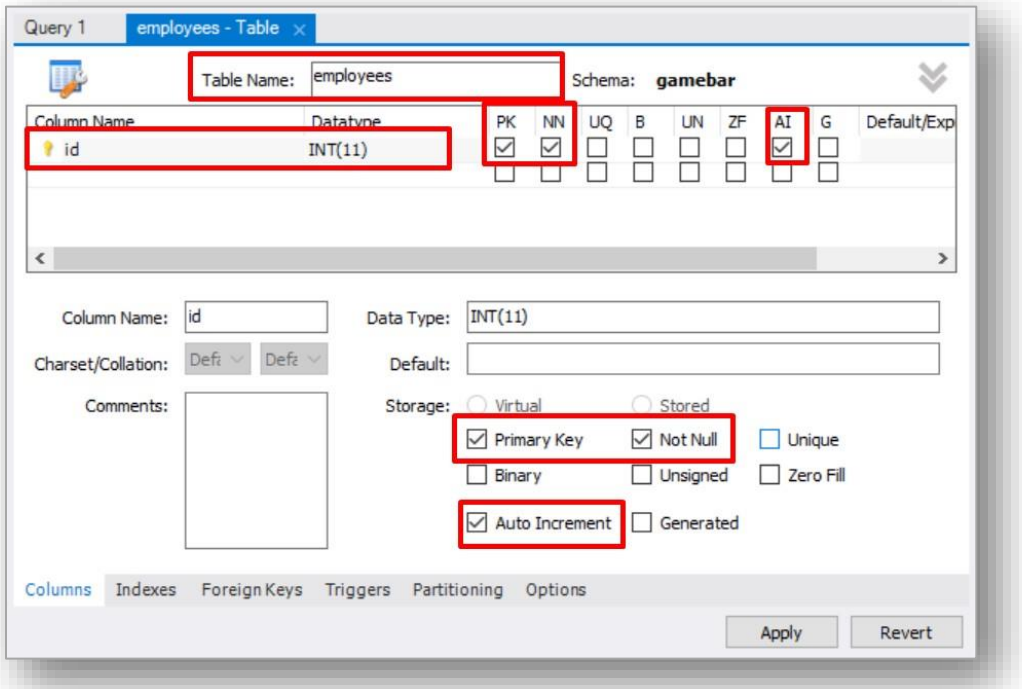
Creating a New Database

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

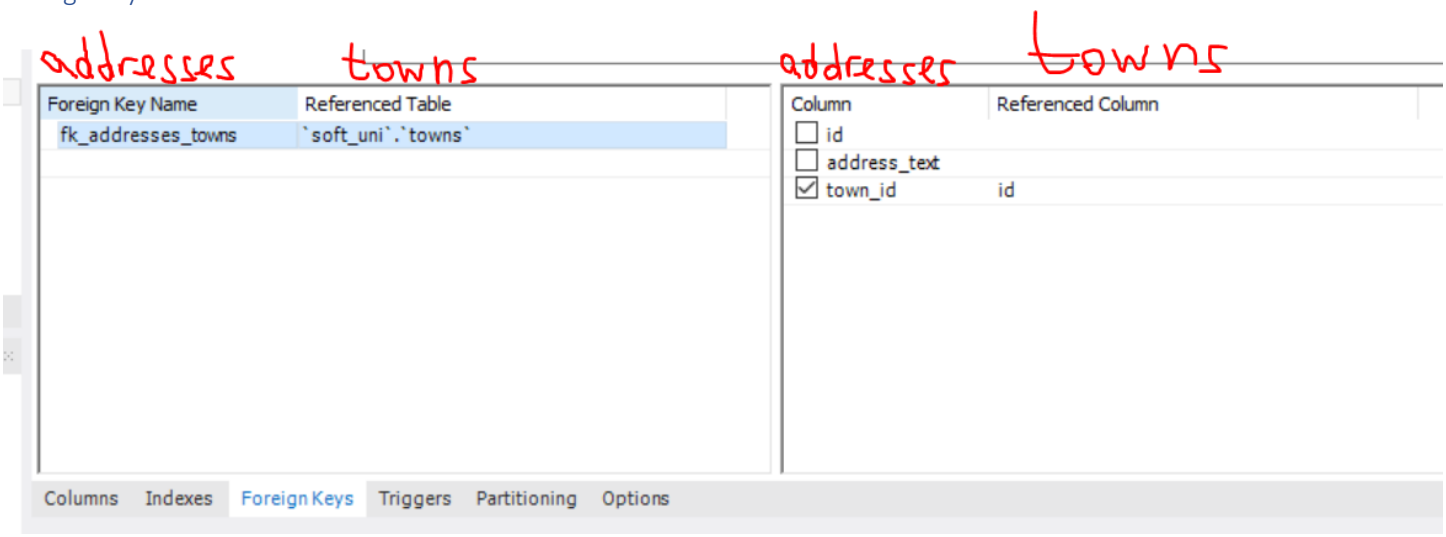




- **A Primary Key is used to uniquely identify and index records**



Foreign keys



- **Adding foreign keys**

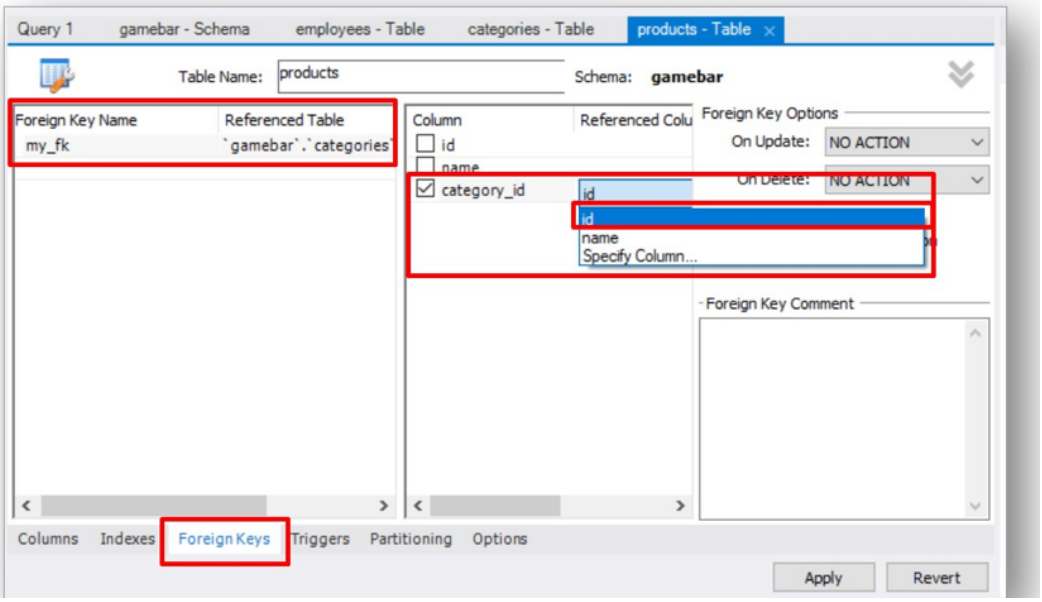
A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.
The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

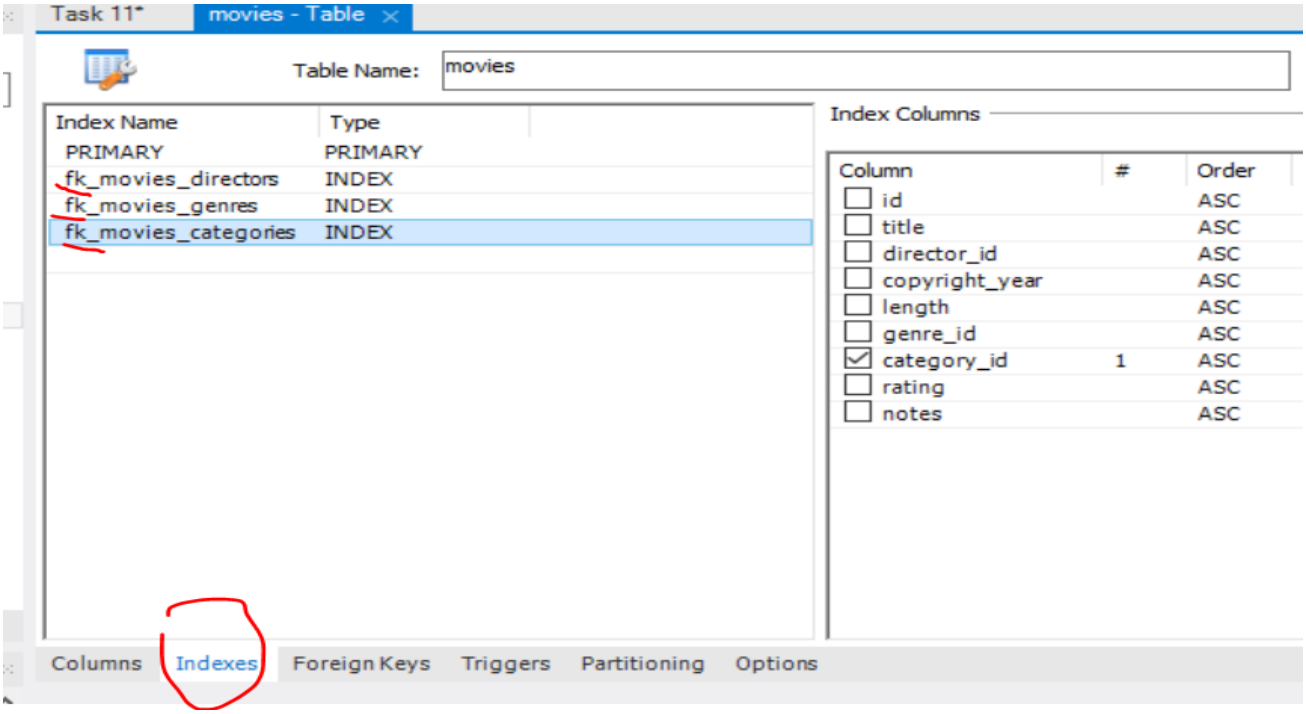
Конвенция при изписване на foreign key поле: fk_fromMinions_toTowns
В minions е чуждия ключ, а в Towns е primary ключ.

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**";



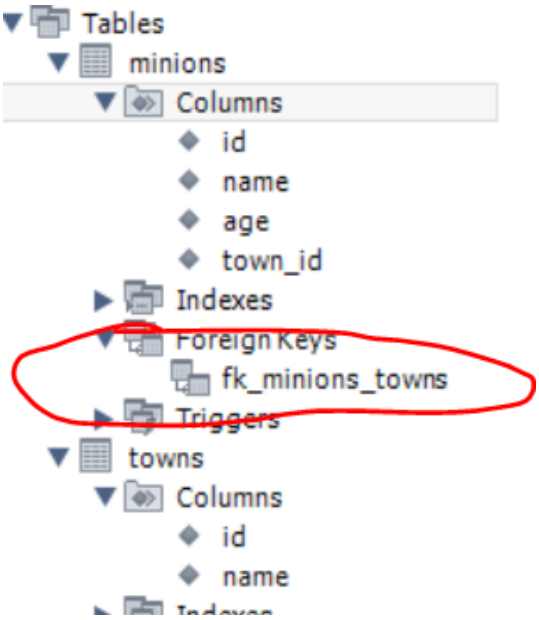



```
CREATE TABLE `Orders` (
    `OrderID` int NOT NULL,
    `OrderNumber` int NOT NULL,
    `PersonID` int,
    PRIMARY KEY (`OrderID`),

    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`);
```

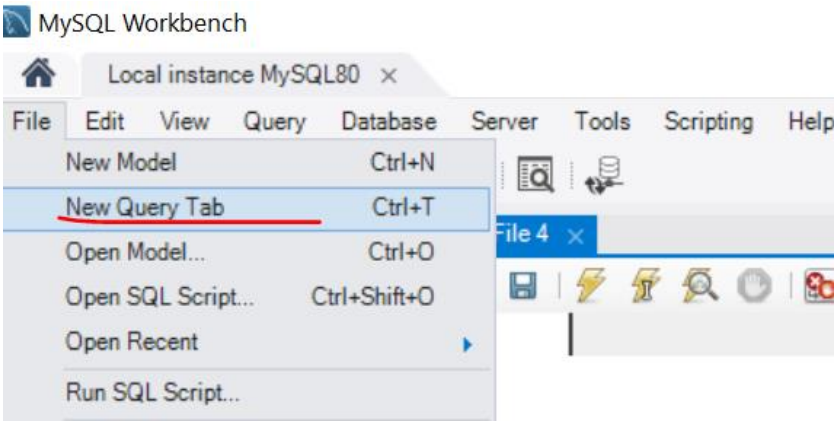


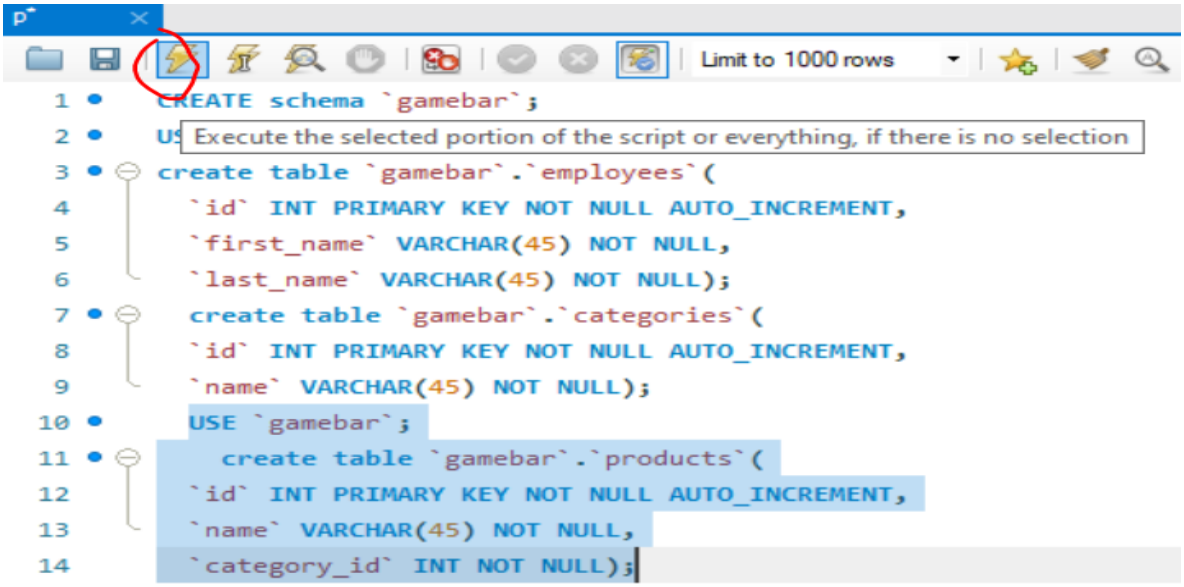
Where to run our SQL queries

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**

Using basic SQL queries

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

Erasing data in SQL:

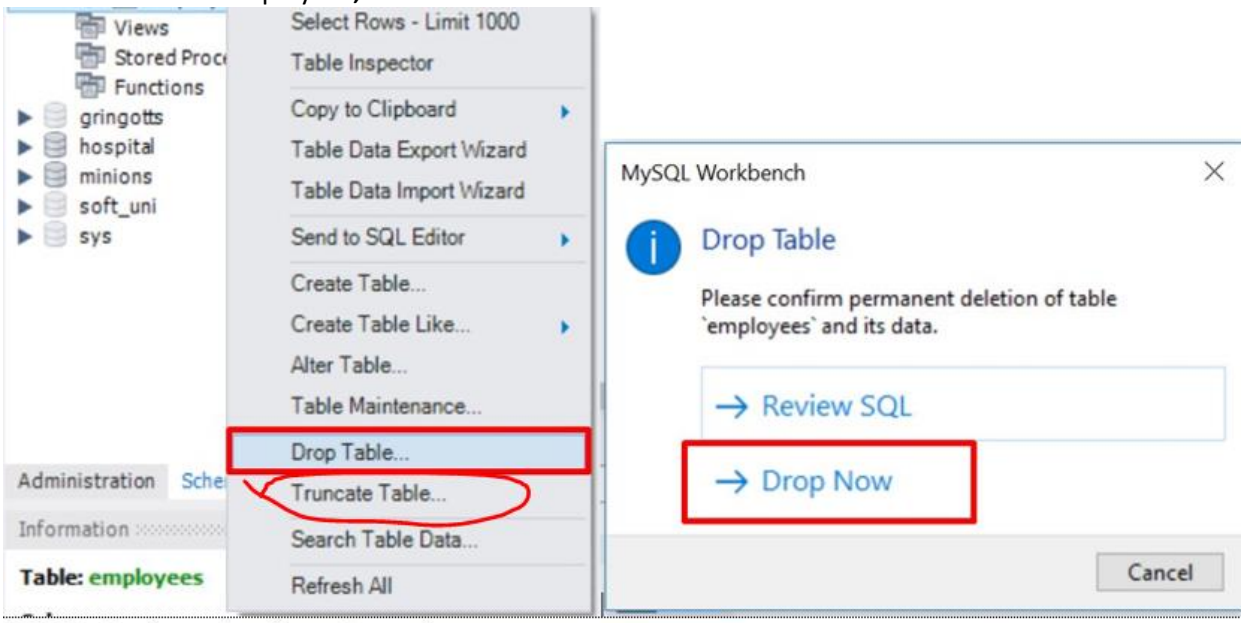
- Deleting structures is called **dropping**

- 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!

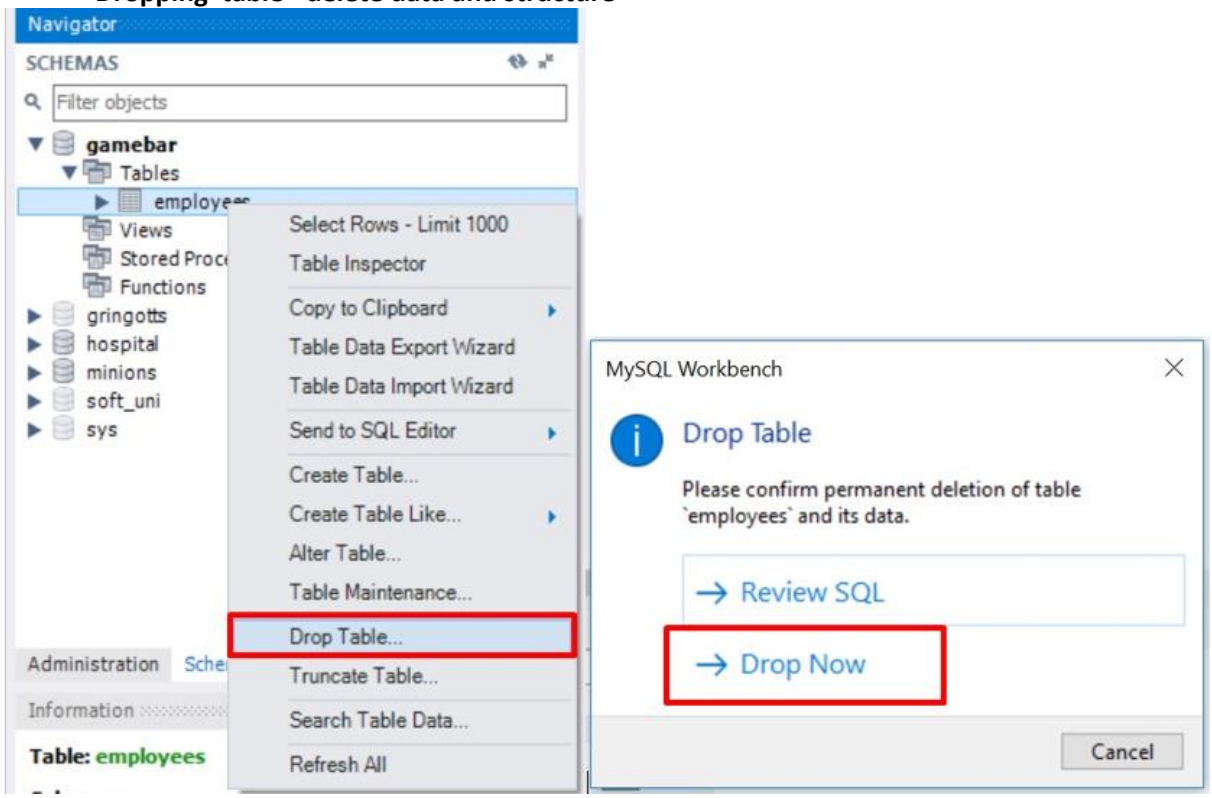
- **To delete all the entries in a table, but keep the table structure**

TRUNCATE TABLE employees**;**



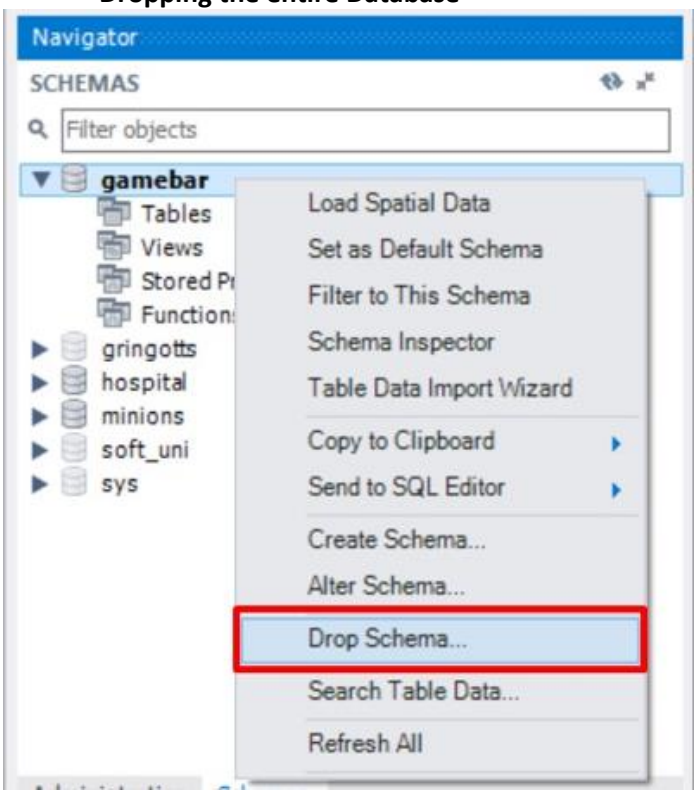
DROP Table `gamebar`.`employees`

- **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;

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;

1.10. DDL more - 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 – за MySQL
ALTER TABLE people **ALTER COLUMN** email VARCHAR(100)**; -** колоната email става от нов тип – за PostgreSQL

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() ;

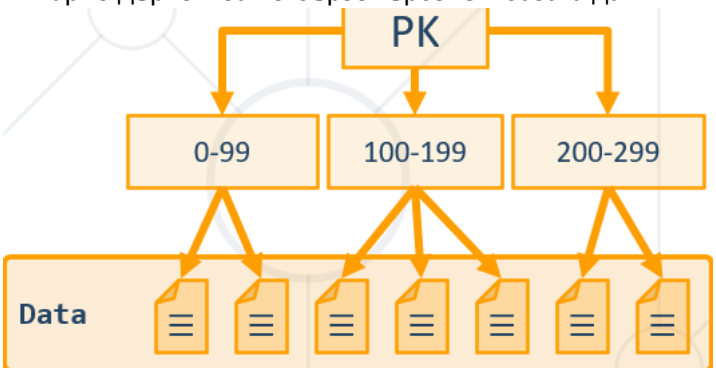
ALTER TABLE `users`
CHANGE COLUMN `username` `username` VARCHAR(30) NOT NULL DEFAULT 'Bai Peshu Starshi' ;

1.11. DDL – Indexes

Info

- Indexes are used to improve the performance of certain types of SELECT queries
- Indexes are created on one or more columns
- Indexes are structured as an **ordered tree –** implemented by means of special data structure such as **B-tree** (Binary Tree - unordered), **Binary Search Tree (ordered)** or a bitmap (*map of bits*) based on the type of index

Бинарно дърво – за по-бързо търсене в базата данни



- Indexes typically have a memory footprint when created – i.e. created in RAM memory

- **Indexes slow down DML queries** (INSERT, UPDATE and DELETE) since the index must be rebuild – в различните бази данни се случва/реализира по различен начин на практика!

- Indexes can be unique (meaning all values in the indexed column must be unique) or non-unique
- Indexes are automatically created from PRIMARY and UNIQUE key constraints

- Indexes are created typically on columns that:
  - Are primary/foreign keys that participate often in JOIN queries
  - Are used often in queries that retrieve values based on a range (e.g. values between two dates)
  - Some combined columns search used very often/needed in our application
  - Participate often in sorting operations in queries (in an ORDER BY clause)
  - Participate often in aggregation queries (in GROUP BY clause)
- Indexes are typically **not** created on columns that:
  - Have a small number of unique values
  - Are rarely used in queries

- Types of indexes in MySQL database:
  - B-tree index
  - Hash index
  - FULLTEXT indexes
- Types of indexes in Postgres
  - B-tree
  - Hash
  - GiST
  - SP-GiST
  - GIN
  - BRIN

Create B-Tree index

- B-tree index – the standard type of index in a MySQL and Postgre databases – useful when selecting values in a range and is created with the CREATE INDEX command

CREATE INDEX <name> ON table_name(<field>, <field>)

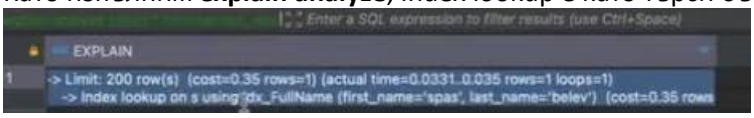
CREATE INDEX idx_StudentName
ON Students (first_name, last_name);

DROP INDEX idx_StudentName;

Когато създадем композитен индекс, то за да търси бързо и да търси по индекс изобщо, трябва да използваме и двете полета за търсене (образуващи въпросния индекс). Особено е важно това!

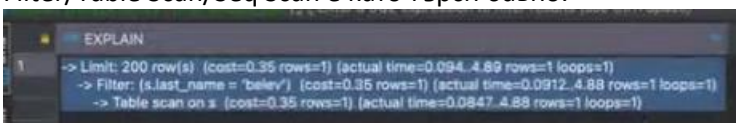
```
50•CREATE index idx_FullName
51 ON lecture_about_ddl.Students_6 (first_name, last_name);
52
53
54•explain analyze select * from lecture_about_ddl.Students_6 s
55 where s.first_name  = 'spas' and last_name = 'belev'
56
```

Като изпълним **explain analyze**, Index lookup е като търси бързо по индекс.



Index Scan е междинно между Index lookup и Table Scan/Seq Scan.

Filter/Table Scan/Seq Scan е като търси бавно.



Create Hash index

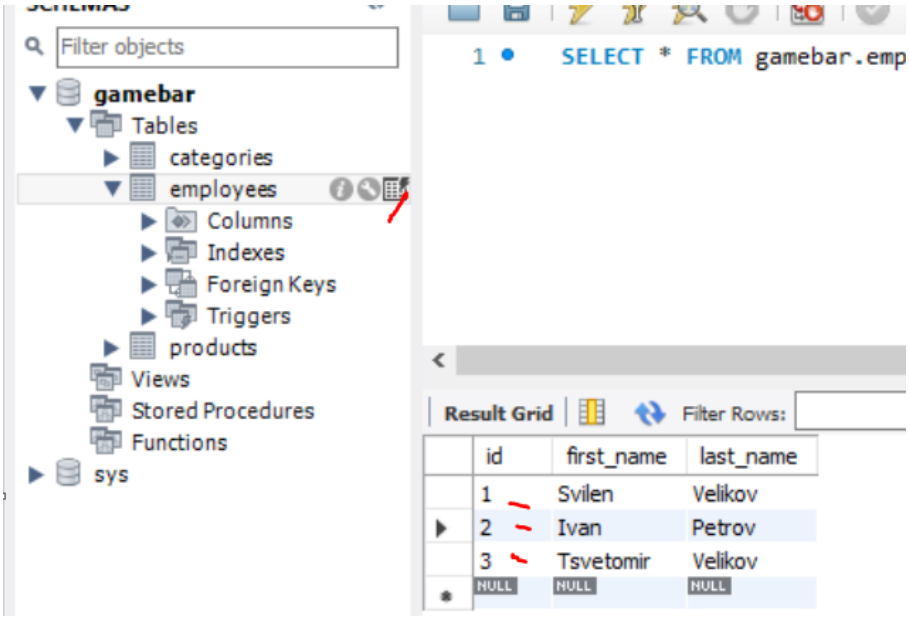
- Hash index – for columns with a small number of unique values and is typically used when data is loaded in chunks

CREATE INDEX status_ind **ON** Vacations **using hash** (Status);

1.12. DML – Data Modification Language

1.12.1. 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

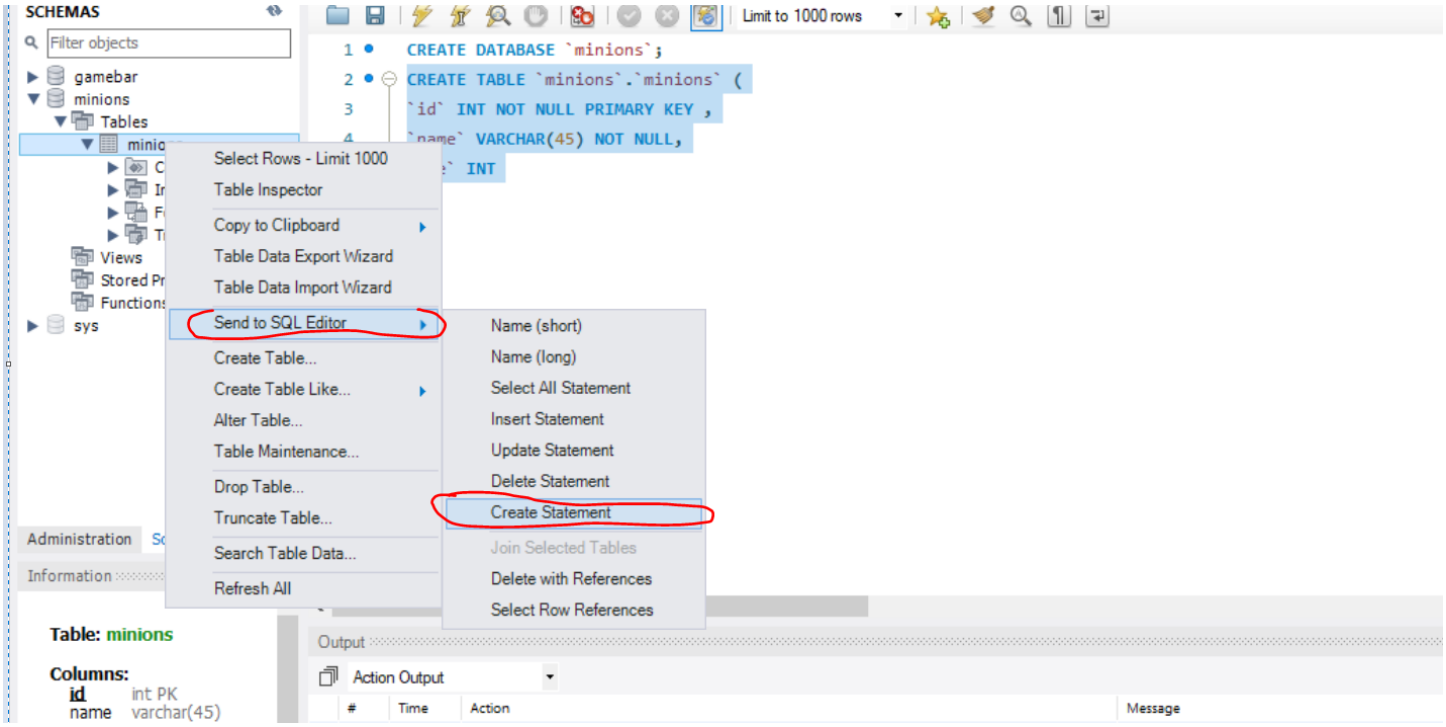
```
INSERT INTO cards(card_status, card_number, bank_account_id)
(    без тези скоби в judge
SELECT (
        CASE
                WHEN id BETWEEN 191 AND 199 THEN 'Active'
                WHEN id BETWEEN 200 AND 299 THEN 'Inactive'
                WHEN id BETWEEN 300 AND 500 THEN 'Deleted'
        END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

1.13. 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



1.14. Advanced SQL queries

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

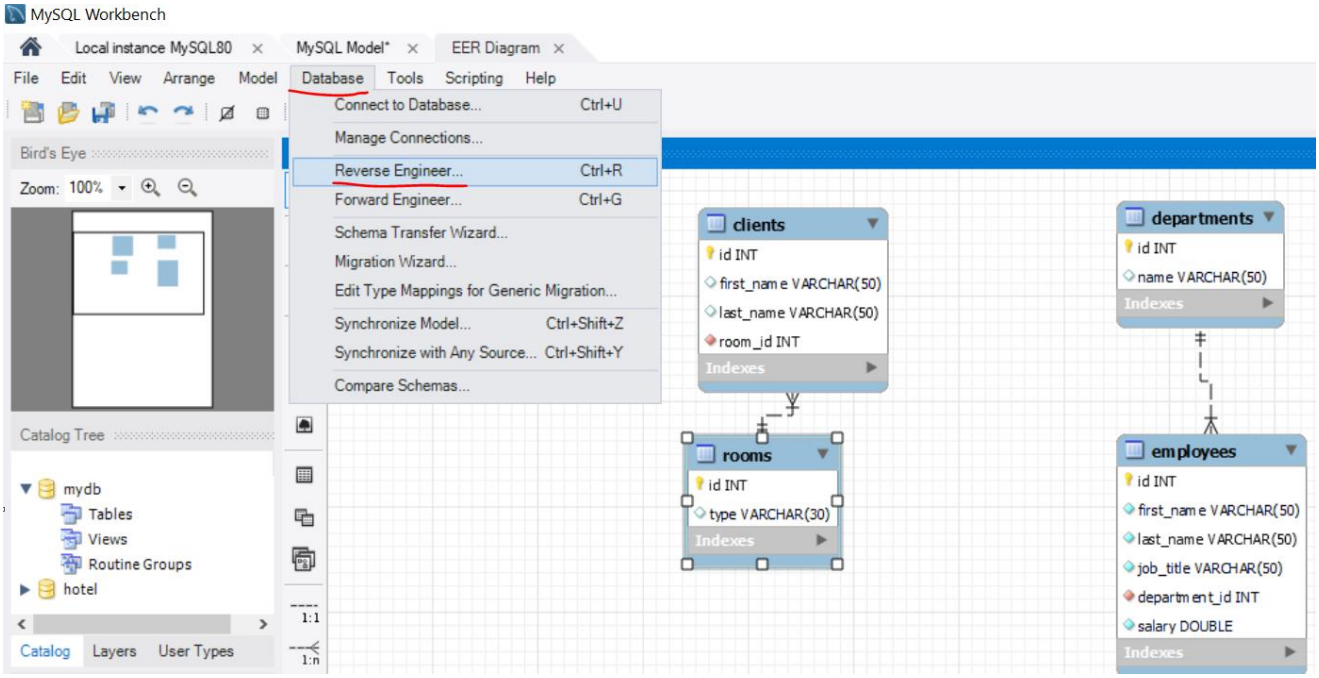
Сортира Double в низходящ ред
SELECT * **FROM** `employees`
ORDER BY `salary` **DESC;**
ORDER BY `salary` е същото като **ORDER BY** `salary` **ASC**

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**;**

1.15. E/R Diagram – диаграма на свързаността



2. DML - BASIC CRUD (Create,Read,Update, Delete) OPERATIONS

Which of the following is NOT a valid SELECT statement ?
a. SELECT (SELECT ID FROM EMPLOYEES) EMP FROM EMP
b. SELECT * FROM Employees WHERE (SELECT SALARY FROM EMPLOYEES WHERE ID == 1) == 2000
c. SELECT * FROM (SELECT SALARY FROM EMPLOYEES WHERE ID == 1)
d. SELECT * FROM EMPLOYEES WHERE ID IN (1)

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

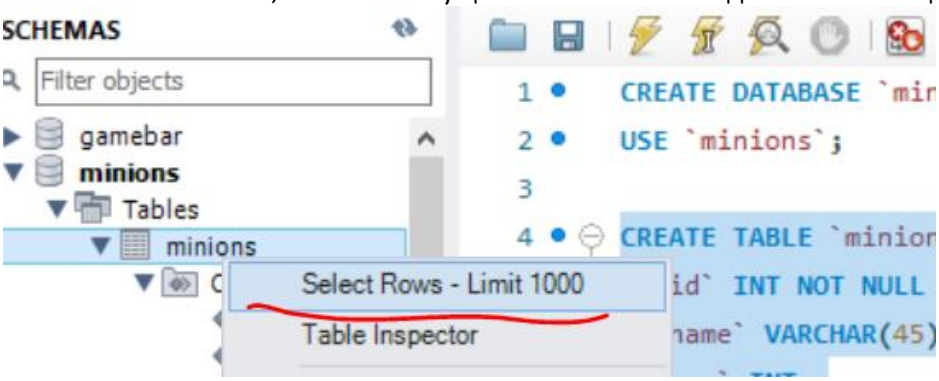
DELETE FROM `projects`
WHERE `start_date` = '2006-01-01';

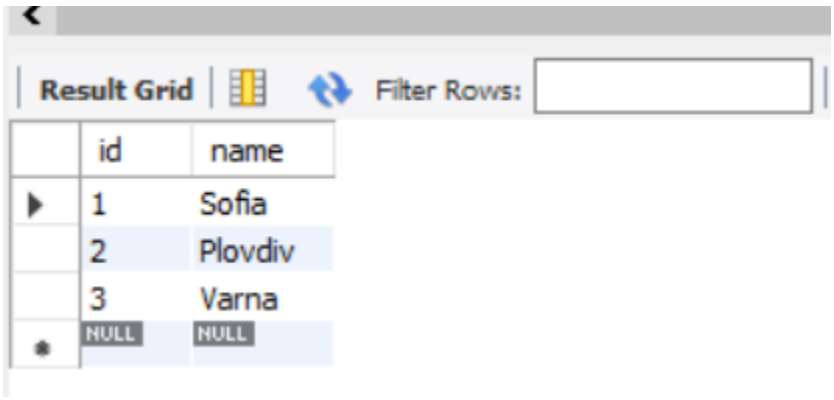
2.2. Retrieving Data

Retrieve Records in SQL - using GUI:

- Get all information from a table

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

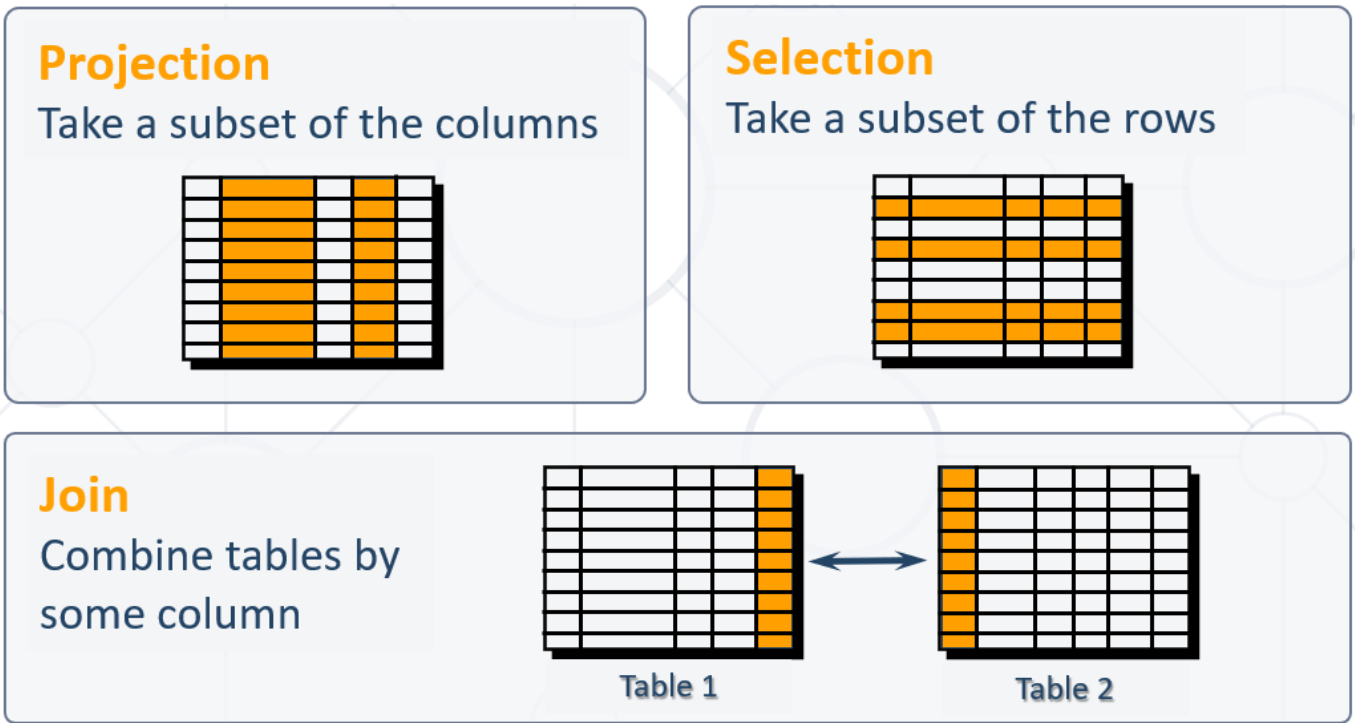




- **You can limit the columns and number of records**

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

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`;
```

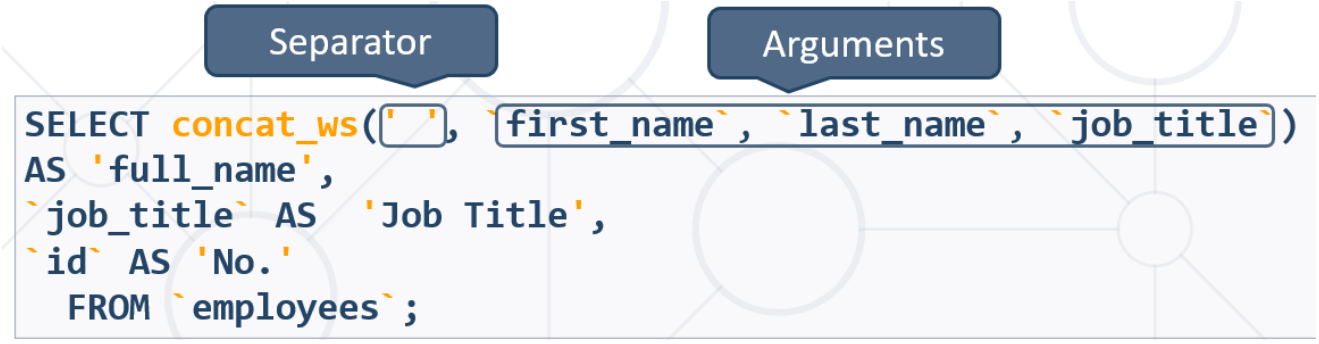
Concatenation – когато ги обединява в резултата от SELECT

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

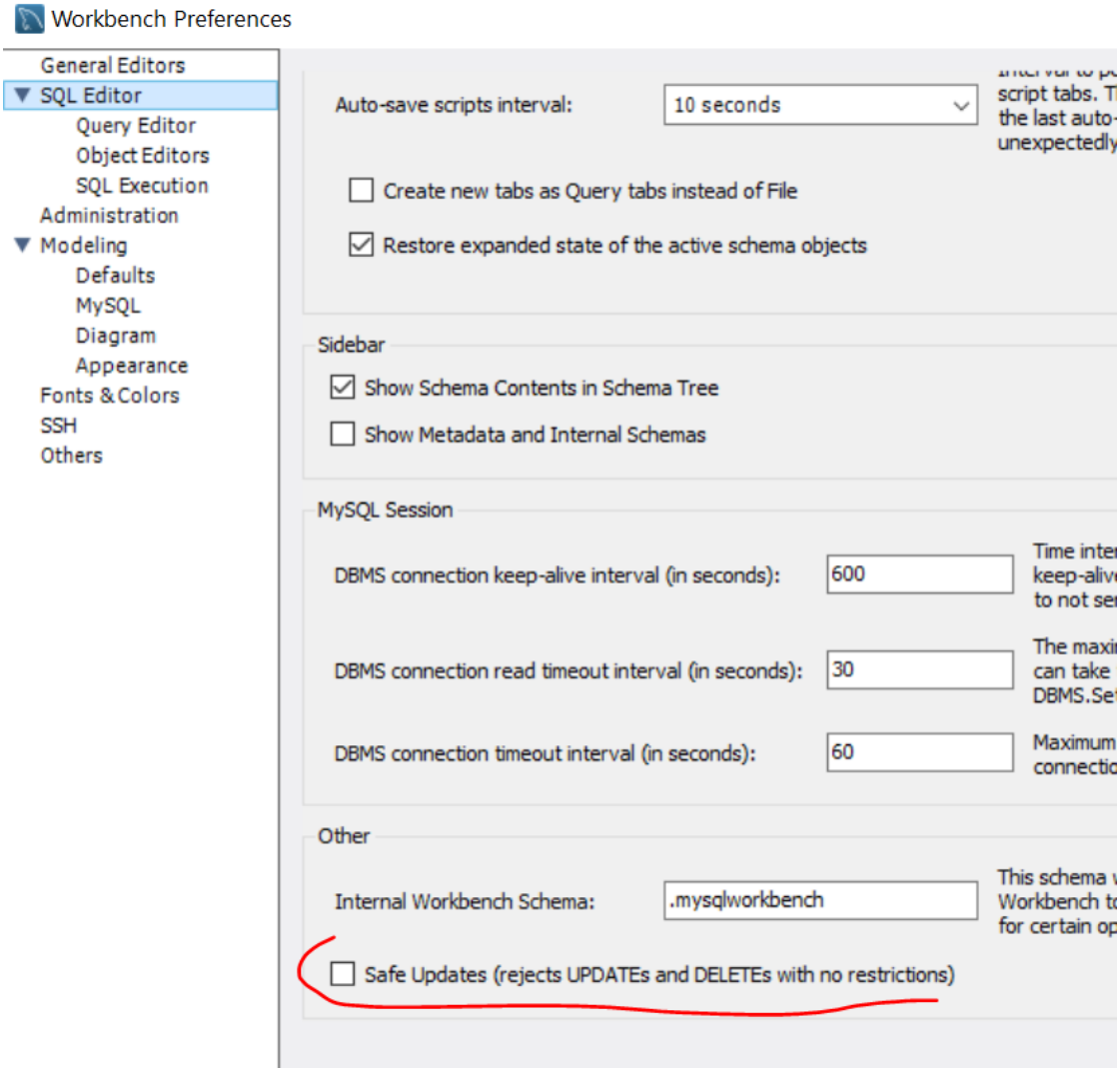
Updating data

Correcting/Updating records in SQL – using the GUI:

	id	first_name	last_name
	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



The SQL UPDATE command

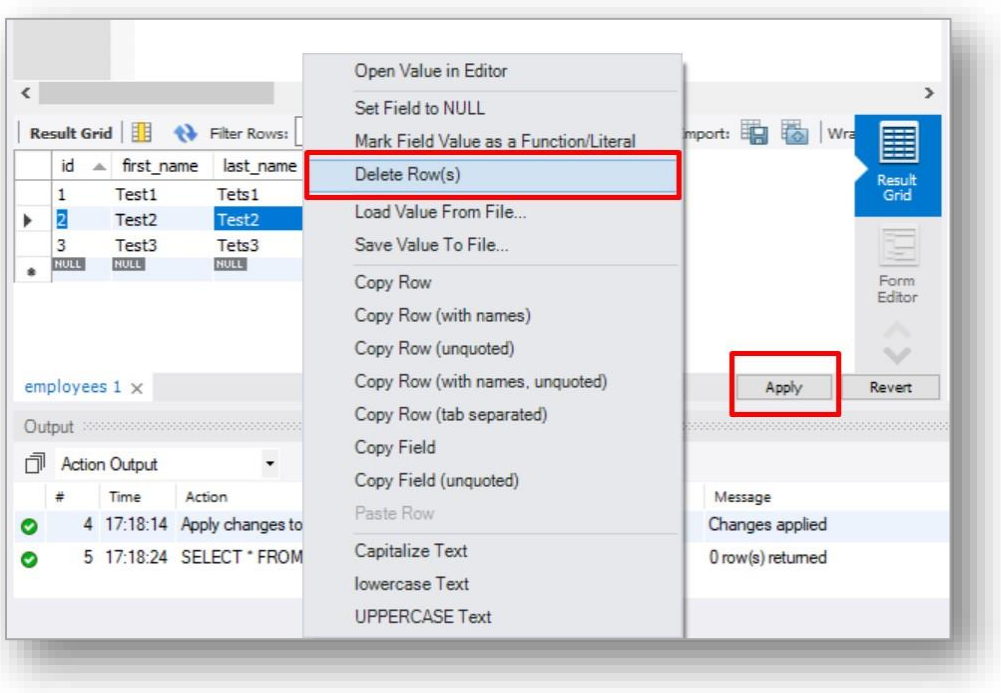
```
UPDATE `employees`
SET `last_name` = 'Brown'
WHERE `employee_id` = 1;

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

- **Deleteing row is part of DML**



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

- 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

Arithmetic expressions containing a null value are evaluated to null!!!

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

year, %Y, %y	YEAR(Date)
month, %M, %m	MONTH(Date)
day, %w, %D	DAY(Date)

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

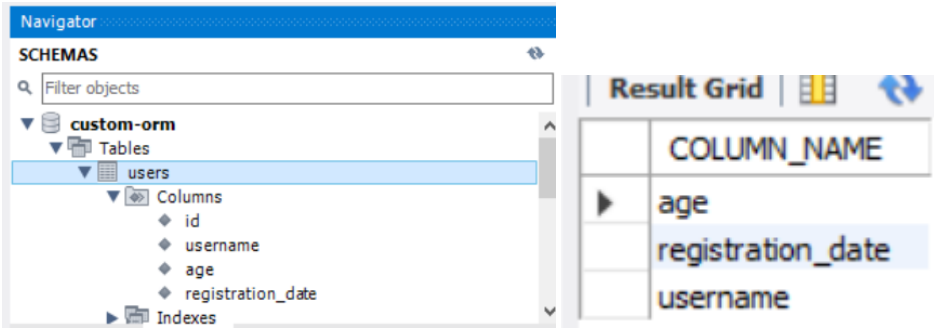
3.4. Wildcards

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
```

```
SELECT * FROM `customers`
WHERE `city` NOT REGEXP '[a-c]%';    -  да не съдържа a, b or c
```

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 IF(1 > 0, 'true', 'false'); --returns 'true'
```

```
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;

- you can exit a loop with the **LEAVE** command
- you can continue to next iteration with the **ITERATE** command

4. Data Aggregation

4.1. Grouping - Consolidating Data Based On Criteria

- Grouping allows taking data into **separate groups** based on a **common property**
- Aggregating data allows us to group rows based on certain criteria.
- SQL provides a mechanism for aggregating data using the GROUP BY clause in a SELECT statement.
- SQL provides a standard set of functions that can be used over aggregated data

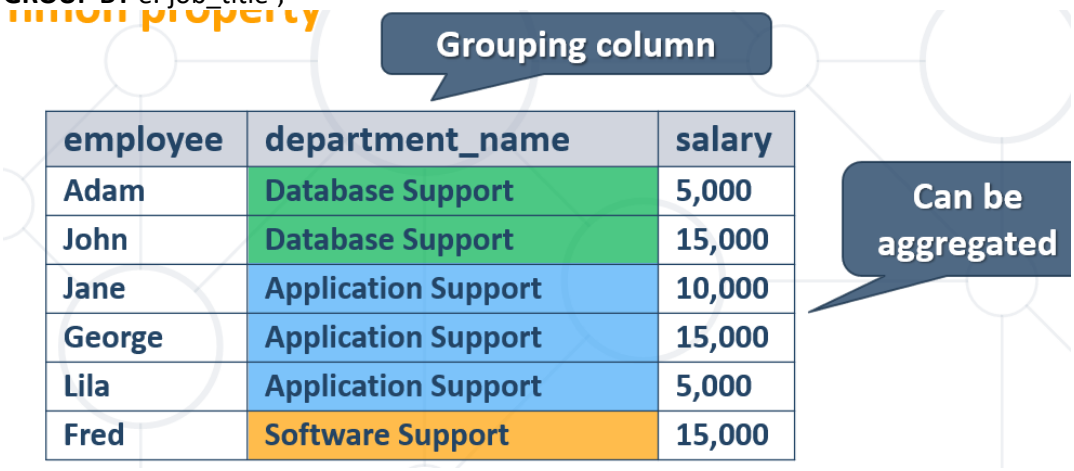
When we want to select and group by column or columns:

- Typically used with aggregate functions
- Can group by multiple columns. The result set will be grouped by the first column listed, then by the next.
- NULL values will be grouped together

Прави като представителна извадка - средната заплата на служителите в даден отдел.

```
select departmentId, avg(salary) as avgSalary
from Employees as e
group by departmentId;
```

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

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

- **Групиране по 2 критерия/колони**

```
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

Aggregating functions are:
COUNT(*) - count of the selected rows
SUM(column) - sum of the values in given column from the selected rows – in PostgreSQL does not allow on varchar type for example. On MySQL return 0 on varchar!
AVG(column) - average of the values in given column
MAX(column) - the maximal value in given column
MIN(column) - the minimal value in given column

- 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

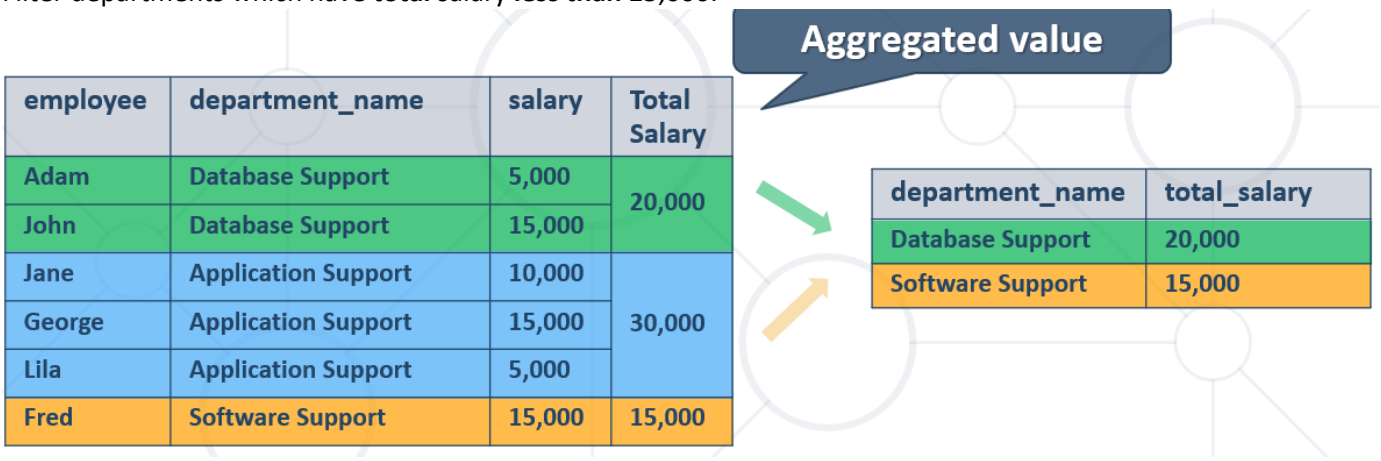
Having Clause

- 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.

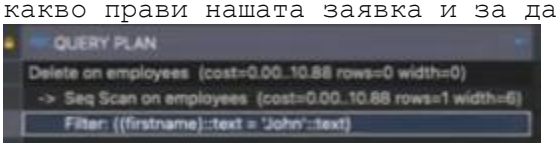
```
SELECT e.`department_id`,
e.`salary` AS `third_highest_salary`
FROM `employees` AS e
WHERE (SELECT ine.`employee_id` FROM `employees` AS ine       връща employee_id когато
 WHERE ine.`department_id` = e.`department_id`                 департамента съвпада
 GROUP BY ine.`salary`                                        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

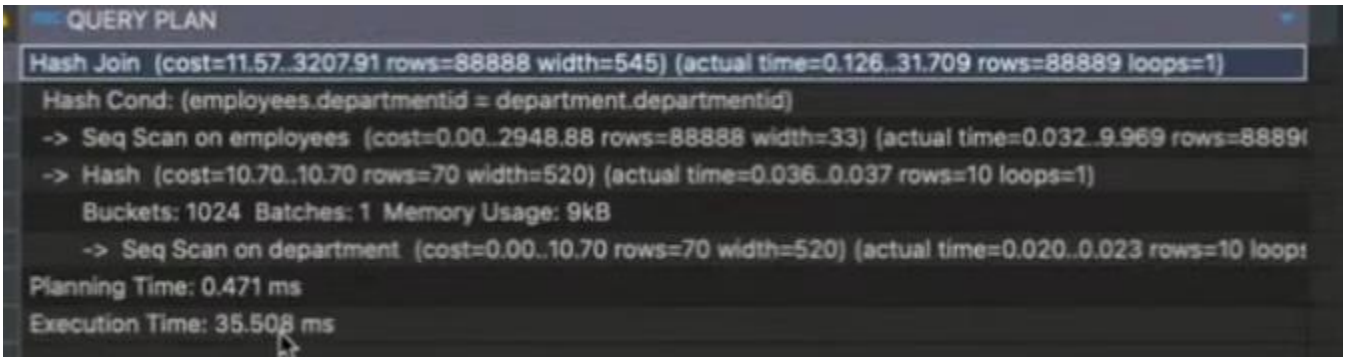
4.5. Debug – EXPLAIN ANALYZE SELECT ..

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

explain
`прави план на заявката без да я изпълнява - има го и в MySQL - като не сме сигурни какво прави нашата заявка и за да не стане някаква беля`



explain analyze



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:

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.

- Students have the following characteristics:

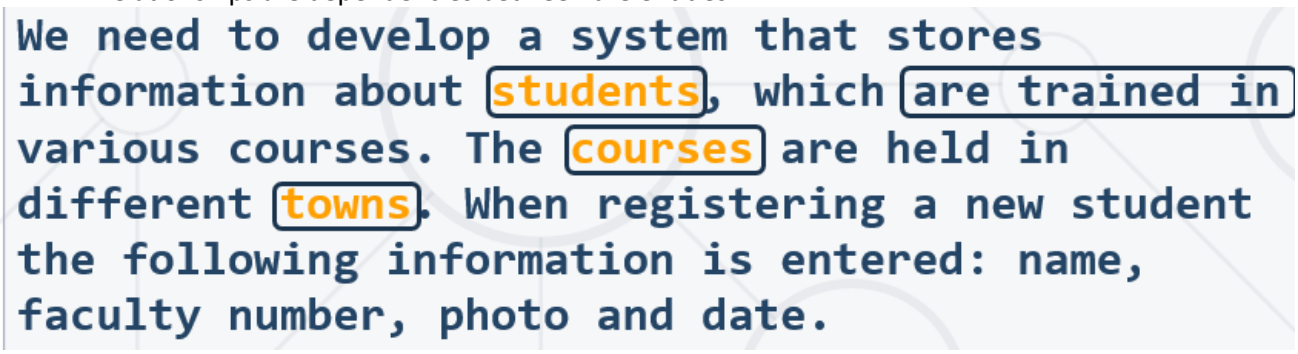
- Name, faculty number, photo, date of enlistment and a list of courses they visit

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)
);
```

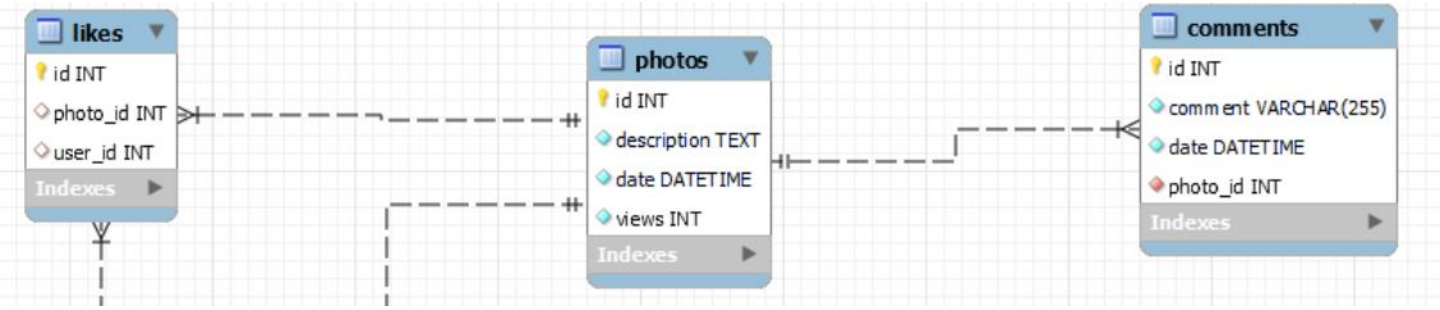
Важно: за да работи foreign key, трябва да има primary key в таблиците

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

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

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

CONSTRAINT fk_students_exams_exams
FOREIGN KEY `students_exams`(`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 (`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:

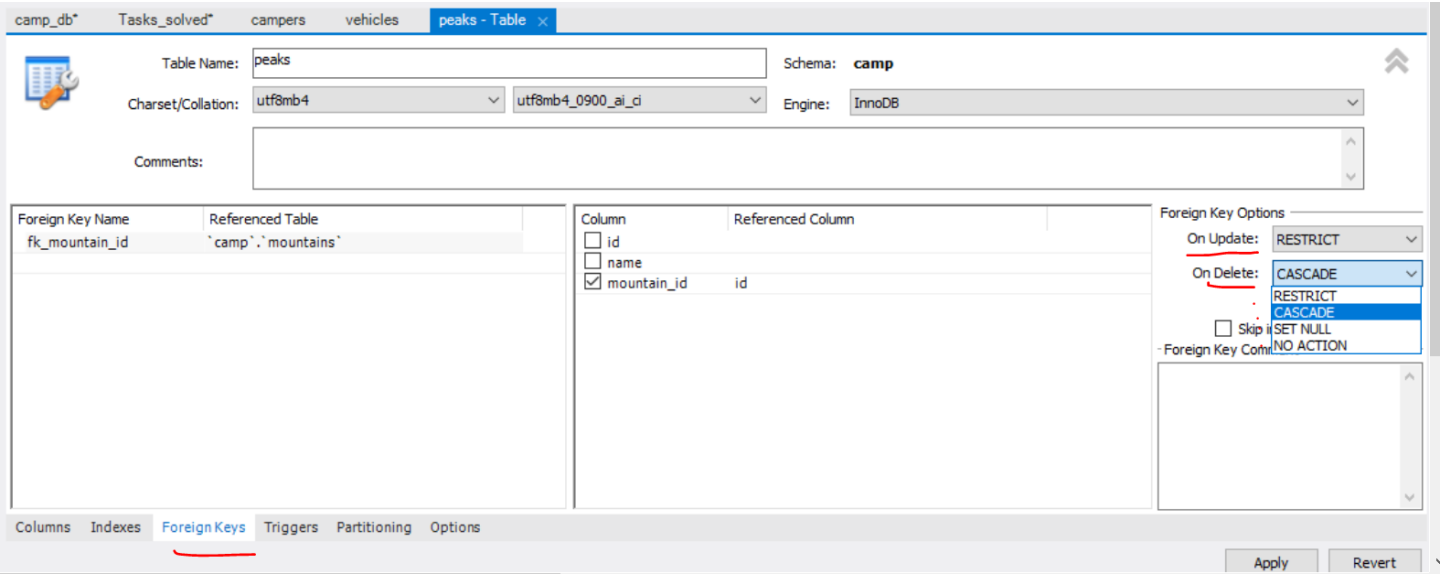
```
CREATE TABLE drivers(
  driver_id INT PRIMARY KEY,
  driver_name VARCHAR(50)
);
CREATE TABLE cars(
  car_id INT PRIMARY KEY,
  driver_id INT,
  CONSTRAINT fk_car_driver FOREIGN KEY(driver_id)
  REFERENCES drivers(driver_id) ON DELETE CASCADE
);
```

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
);
```



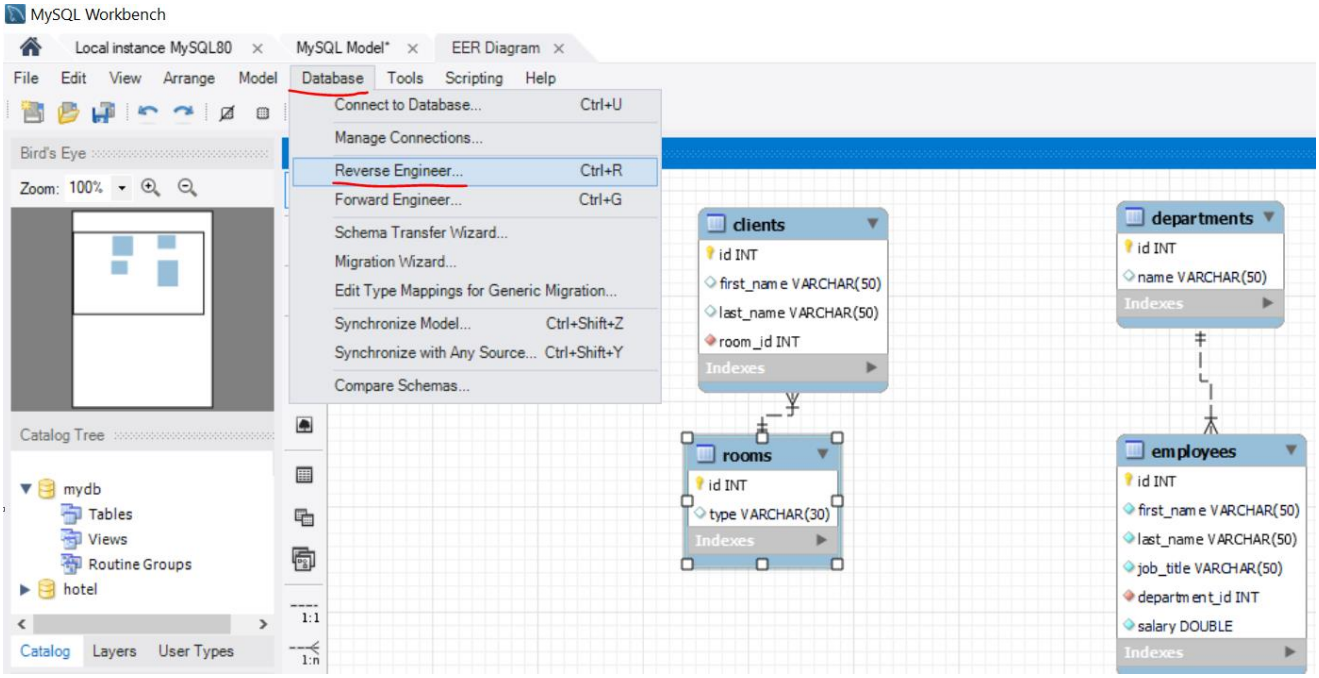
5.5. E/R Diagram

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

Relational Schema

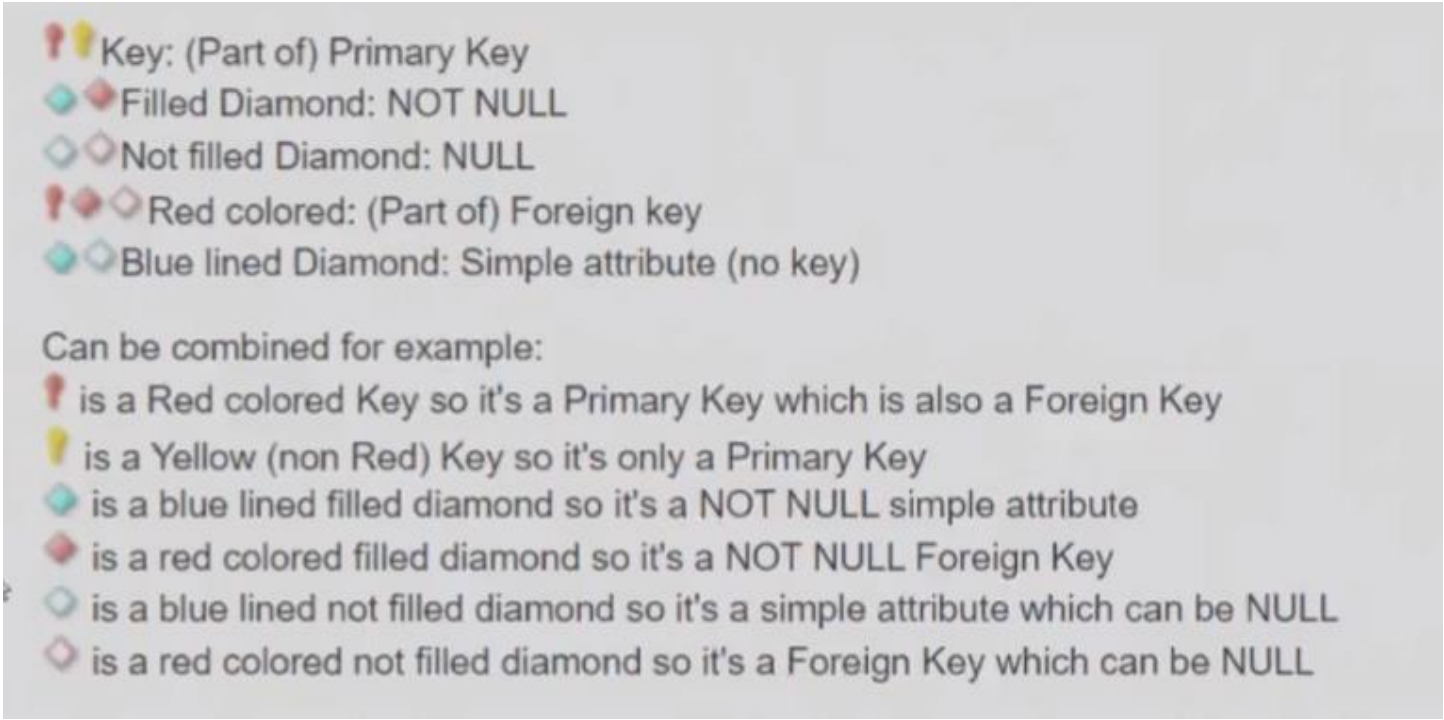
- **Relational schema** of a DB is the collection of:
  - The schemas of all tables
  - Relationships between the tables
  - Any other database objects (e.g. constraints)
- The relational schema describes the **structure** of the database
  - Doesn't contain data, but **metadata**
- Relational schemas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

Reverse engineering

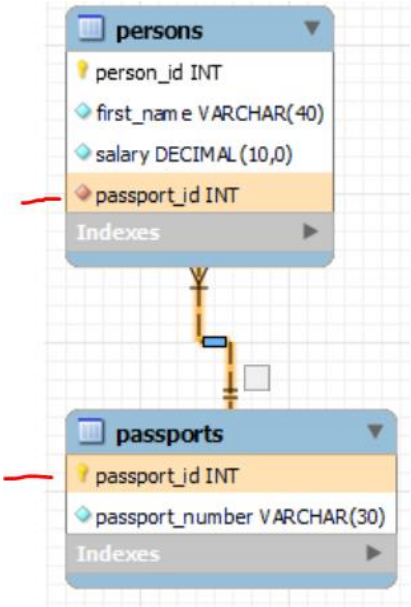


Добра практика е да слагаме префикс на името на всяка таблица, която създаваме. Например:
my_Products
my_Clients
my_personnel

или
wp_posts
wp_links
wp_commentmeta
wp_comments
wp_users



В persons таблицата пишем дефиницията за foreign key



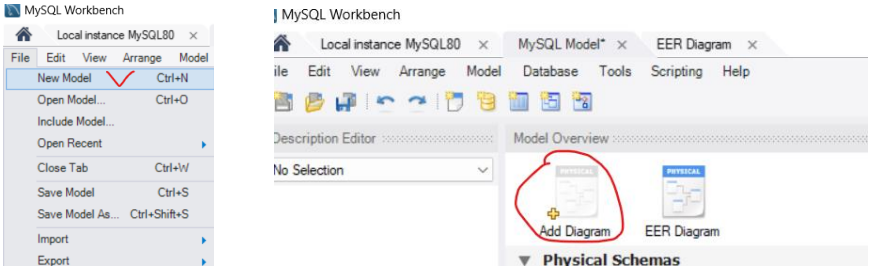
Като посочим стрелката, и ни показва кои полета са свързани.

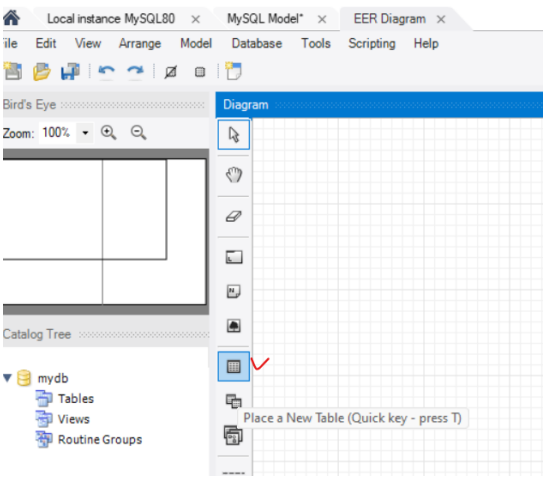
Как тълкуваме стрелката - в случая таблица persons (parent) има foreign key passport_id, който сочи към таблица passports (child дете) с ключ passport_id (или passport_id на persons сочи към passport_id на passports).

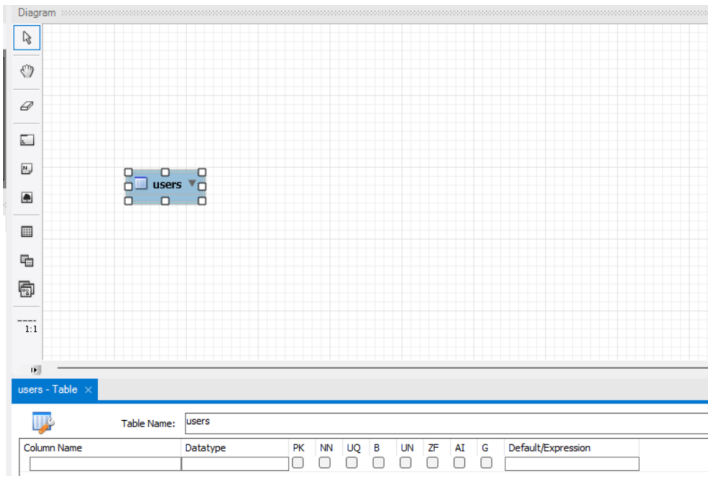
Какво означава линията:

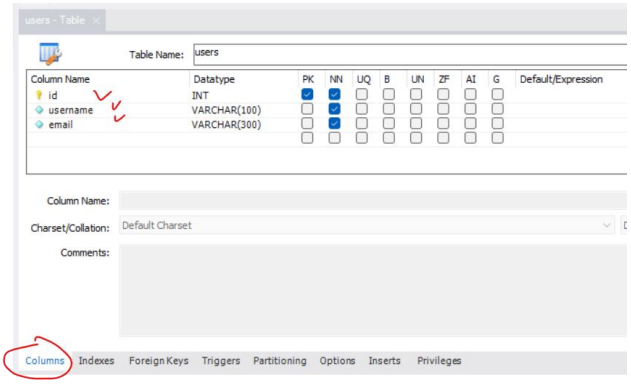
- **Плътна линия означава в двойката баща-дете, че и двете са задължителни/и двете са primary keys**
- **Прекъсната линия означава, че в двойката parent-child не са задължителни и двете**

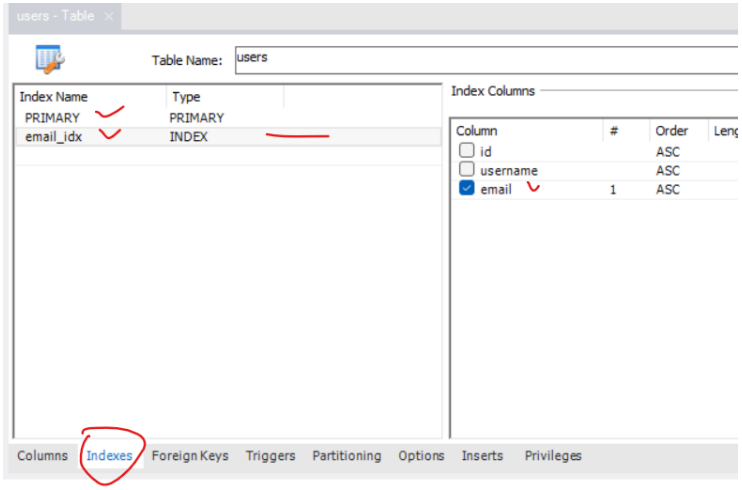
Create new model and export it

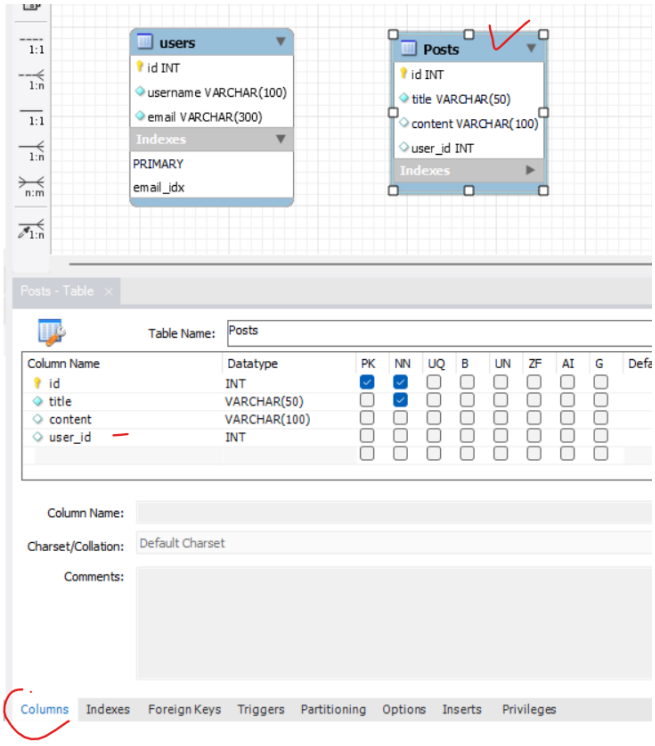


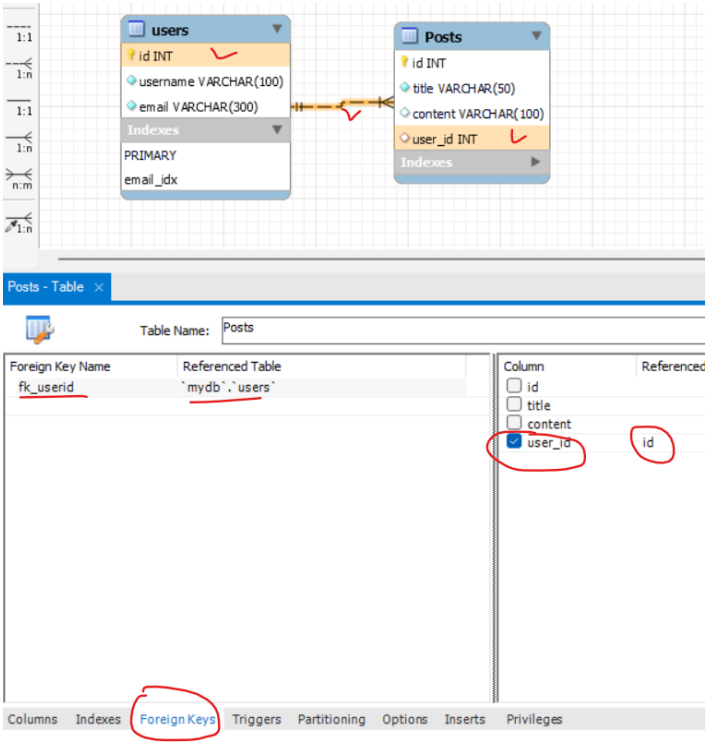






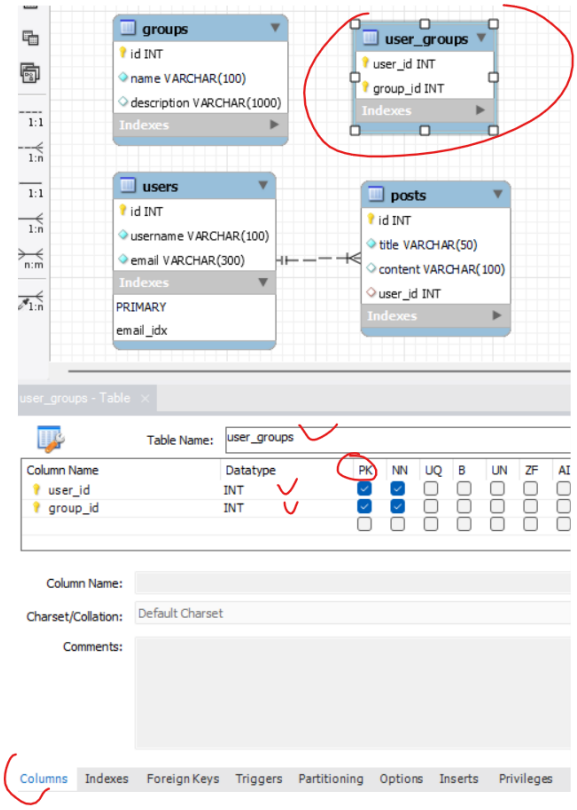


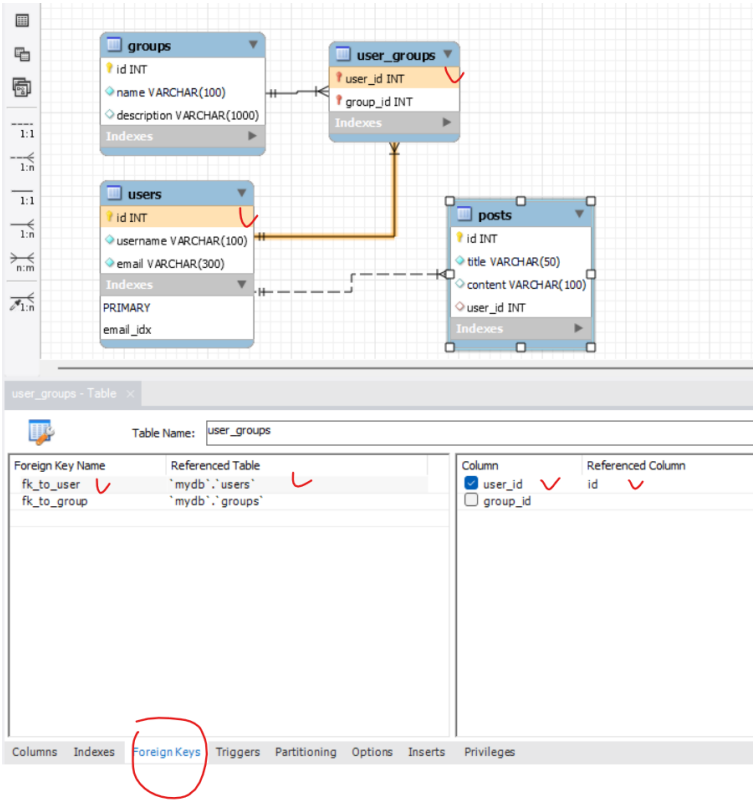




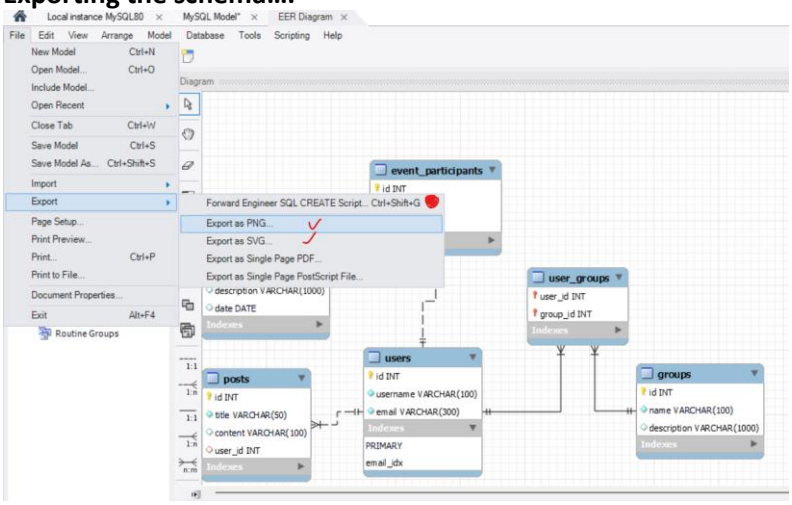
Foreign keys трябва да имат уникални имена – не може в две различни таблици да има (камо ли в една и съща таблица) две еднакви имена за foreign key!!!

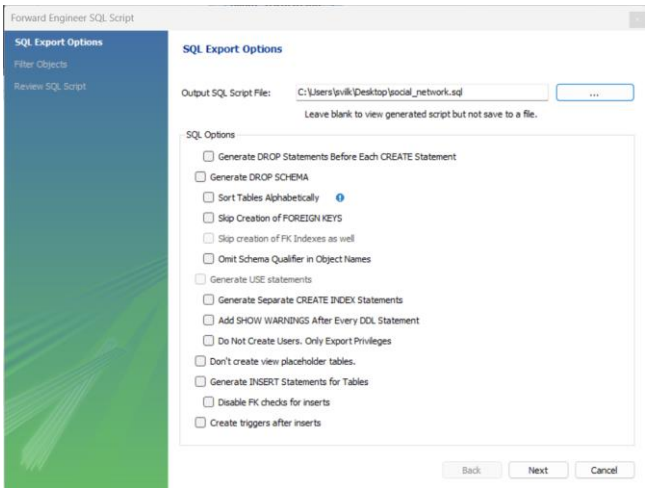
При релация Many to Many - разбиваме на две релации One To Many като правим трета помощна таблица. Третата помощна таблица реално държи ManyToOne релации към двете source таблици:

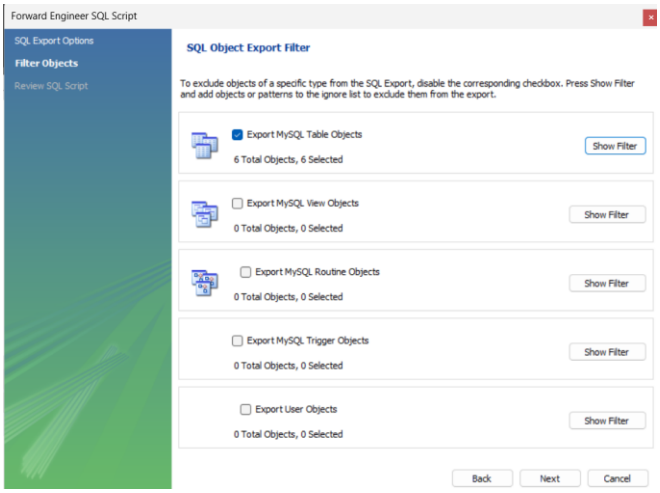




Exporting the schema….







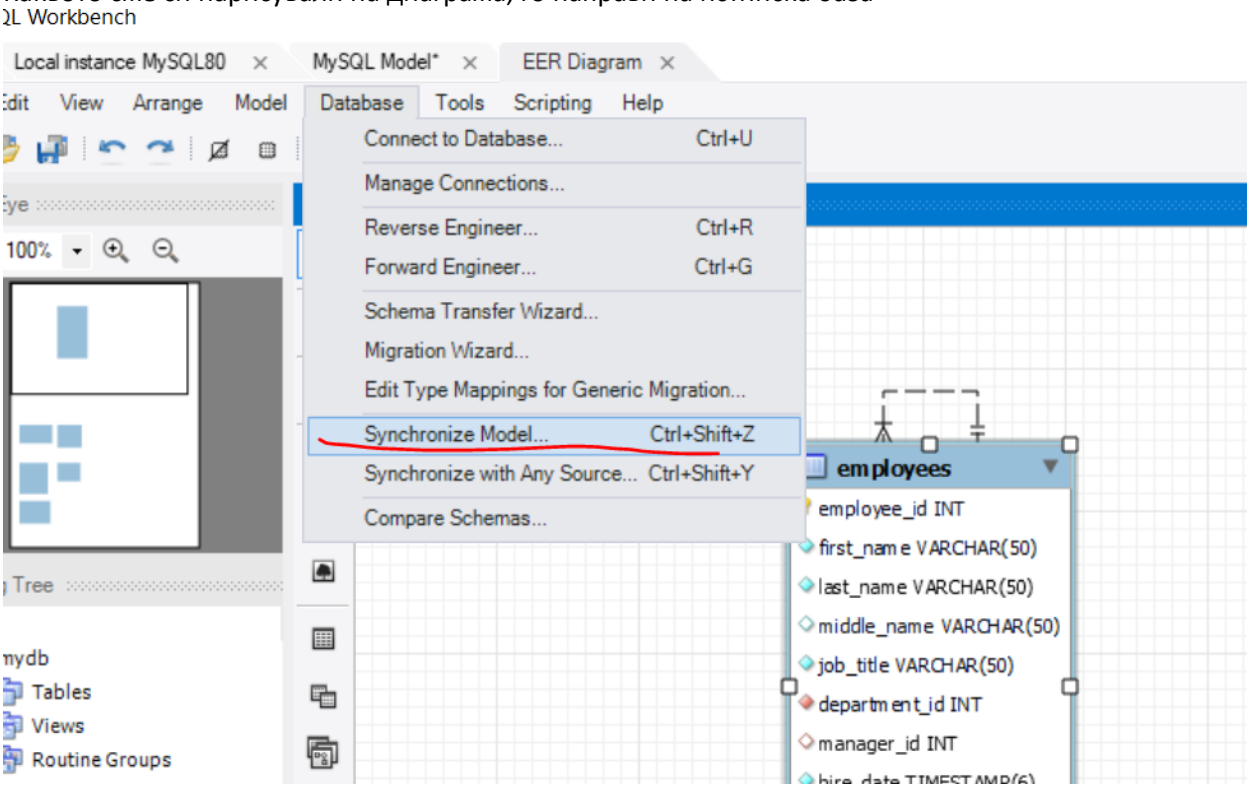
```
-- MySQL Script generated by MySQL Workbench
-- Thu Jul 18 12:25:39 2024
-- Model: New Model    Version: 1.0
-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE,
SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISIO
N_BY_ZERO,NO_ENGINE_SUBSTITUTION';

-- -----------------------------------------------------
-- Schema mydb
-- -----------------------------------------------------

-- -----------------------------------------------------
-- Schema mydb
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `mydb` DEFAULT CHARACTER SET utf8 ;
USE `mydb` ;

-- -----------------------------------------------------
-- Table `mydb`.`users`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`users` (
  `id` INT NOT NULL,
  `username` VARCHAR(100) NOT NULL,
  `email` VARCHAR(300) NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `email_idx` (`email` ASC) INVISIBLE)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mydb`.`posts`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`posts` (
  `id` INT NOT NULL,
  `title` VARCHAR(50) NOT NULL,
  `content` VARCHAR(100) NULL,
  `user_id` INT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_userid_idx` (`user_id` ASC) VISIBLE,
  CONSTRAINT `fk_userid`
    FOREIGN KEY (`user_id`)
    REFERENCES `mydb`.`users` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mydb`.`groups`
```

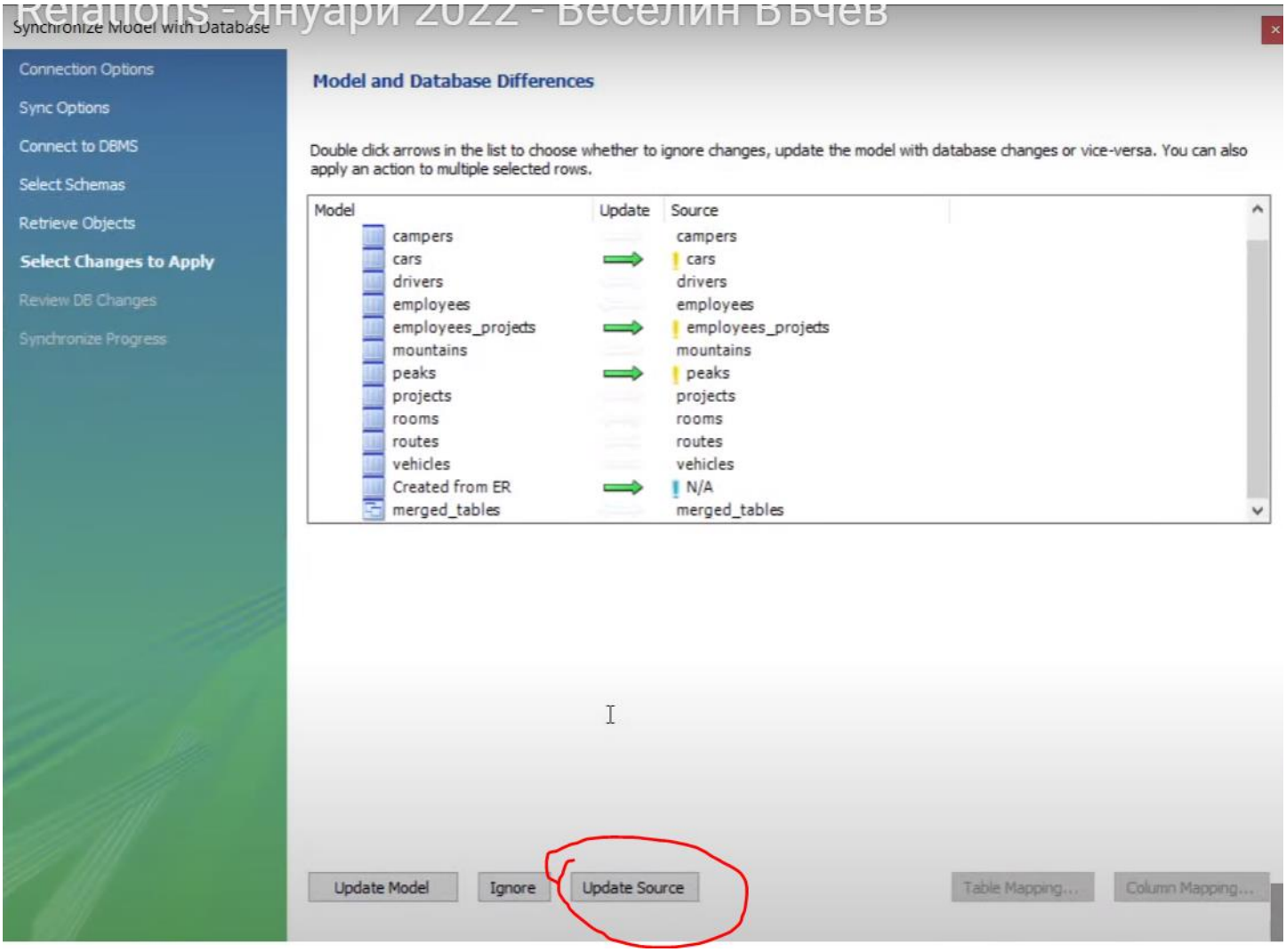
```
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`groups` (
  `id` INT NOT NULL,
  `name` VARCHAR(100) NOT NULL,
  `description` VARCHAR(1000) NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mydb`.`user_groups`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`user_groups` (
  `user_id` INT NOT NULL,
  `group_id` INT NOT NULL,
  PRIMARY KEY (`user_id`, `group_id`),
  INDEX `fk_to_group_idx` (`group_id` ASC) VISIBLE,
  CONSTRAINT `fk_to_user`
    FOREIGN KEY (`user_id`)
    REFERENCES `mydb`.`users` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_to_group`
    FOREIGN KEY (`group_id`)
    REFERENCES `mydb`.`groups` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mydb`.`events`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`events` (
  `id` INT NOT NULL,
  `creator_id` INT NULL,
  `name` VARCHAR(45) NULL,
  `description` VARCHAR(1000) NULL,
  `date` DATE NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mydb`.`event_participants`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`event_participants` (
  `id` INT NOT NULL,
  `user_id` INT NOT NULL,
  `event_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_to_users_idx` (`user_id` ASC) VISIBLE,
  INDEX `fk_to_events_idx` (`event_id` ASC) VISIBLE,
```

```
  CONSTRAINT `fk_to_users`
    FOREIGN KEY (`user_id`)
    REFERENCES `mydb`.`users` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_to_events`
    FOREIGN KEY (`event_id`)
    REFERENCES `mydb`.`events` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

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

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





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

A Cartesian product is formed when:

- A join condition is omitted - когато нямаме никакъв condition
- A join condition is invalid
- All rows in the first table are joined to all rows in the second table

To avoid a Cartesian product, always include a valid join condition!

6.2. Union

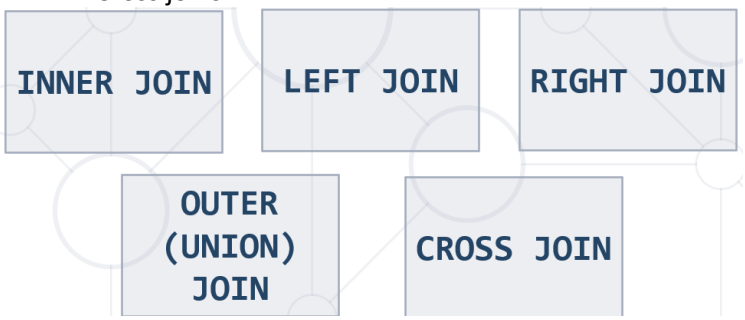
- UNION combines the results from several SELECT statements
- The columns **count**, **types** and **order(поредност)** should match

```
SELECT FIRST_NAME AS NAME
FROM EMPLOYEES

UNION

SELECT LAST_NAME AS NAME
FROM EMPLOYEES
```

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

Different types of joins include:

- Natural joins
- Join with USING clause
- Inner joins with ON clause
- Left, right and full outer joins
- Self joins
- Cross joins



NATURAL JOIN

The NATURAL JOIN combines the rows from two tables that have equal values in all matched by name columns.

Natural Join in SQL refers to **joining two or more tables based on common columns, which have the same name and data type**. We do not need to specify the column used for joining two tables in natural join. Natural join is used to retrieve data from more than one table in a single place.

```
SELECT * FROM ORDERS NATURAL JOIN CUSTOMERS;
```

USING clause

If several columns have the same names, we can limit the NATURAL JOIN to only one of them by the USING clause:

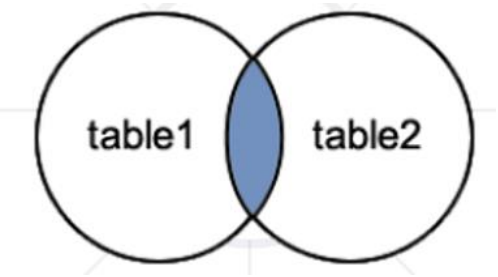
The USING clause **specifies which columns to test for equality when two tables are joined**. It can be used instead of an ON clause in the JOIN operations that have an explicit join clause.

```
SELECT E.EMPLOYEE_ID, E.LAST_NAME, D.LOCATION_ID, D.DEPARTMENT_NAME
FROM EMPLOYEES E
JOIN DEPARTMENTS D
USING (DEPARTMENT_ID);
```

EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_NAME
102	De Haan	1700	Executive
103	Hunold	1400	IT
104	Ernst	1400	IT
...	...	...	...

JOIN(== INNER JOIN) with ON clause

- To specify arbitrary conditions or specify columns to join, the ON clause is used
- Used very often
- взема сечението
- Produces a set of records which **match in both tables**



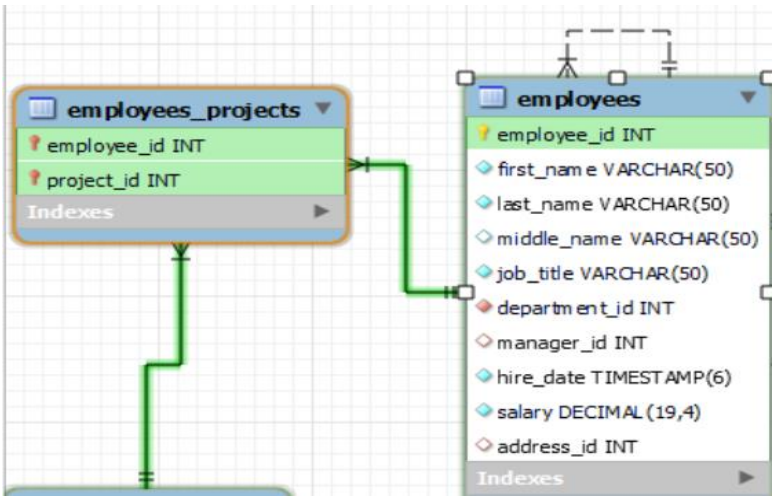

```
SELECT e.`first_name`, d.`name` AS 'dept_name'
FROM `employees` AS e                    - таблица 1 = e
INNER JOIN `departments` AS d            - таблица 2 = d
ON e.`department_id` = d.`department_id`;
```

Пример 1:

```
//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;    //НО само когато за employee_id от employees ИМА ep.`project_id`
```



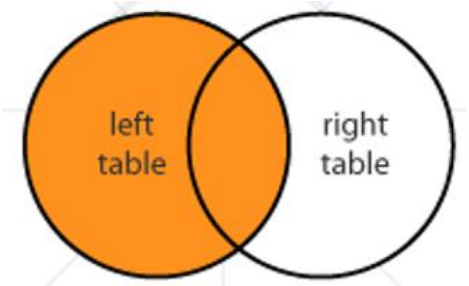
Пример 2:

```
SELECT E.EMPLOYEE_ID, E.LAST_NAME, E.DEPARTMENT_ID, D.DEPARTMENT_ID,
D.DEPARTMENT_NAME
FROM EMPLOYEES E
JOIN DEPARTMENTS D
ON (E.DEPARTMENT_ID = D.DEPARTMENT_ID)
WHERE D.DEPARTMENT_NAME != 'Department 2';
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	Whalen	10	10	Executive
201	Hartstein	20	20	IT
202	Fay	20	20	IT

LEFT JOIN (== LEFT OUTER JOIN)

- A join that returns all the rows from the left table and the matched ones from the right
- If no match, the result is NULL on the right
- Matches every entry in **left** table regardless of match in the **right**



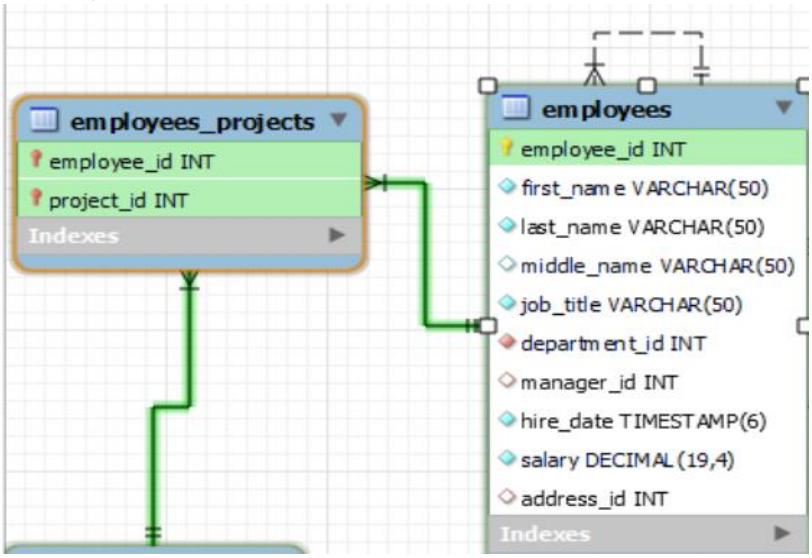
Example, see powerpoint presentation

Пример 1:

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;



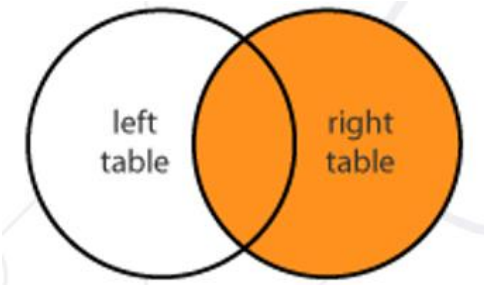
Пример 2:

```
SELECT E.EMPLOYEE_ID, E.LAST_NAME, E.DEPARTMENT_ID, D.DEPARTMENT_ID,
D.DEPARTMENT_NAME
FROM EMPLOYEES E
LEFT JOIN DEPARTMENTS D
ON (E.DEPARTMENT_ID = D.DEPARTMENT_ID);
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	Whalen	10	10	Executive
201	Hartstein	20	20	IT
202	Fay	20	NULL	NULL

RIGHT JOIN (== RIGHT OUTER JOIN)

- A join that returns all the rows from the right table and the matched ones from the left
- If no match, the result is NULL on the left
- Matches every entry in **right** table regardless of match in the **left**



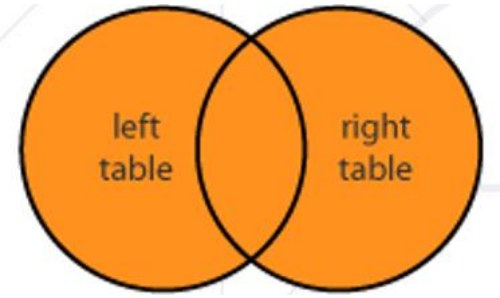
Example, see powerpoint presentation

```
SELECT E.EMPLOYEE_NAME, D.DEPARTMENT_ID, D.DEPARTMENT_NAME
FROM EMPLOYEES E
RIGHT JOIN DEPARTMENTS D
ON (E.DEPARTMENT_ID = D.DEPARTMENT_ID);
```

EMPLOYEE_NAME	DEPARTMENT_ID	DEPARTMENT_ NAME
Jennifer Whalen	10	Administration
Michael Hartstein	20	Marketing
Den Raphaely	30	Purchasing
(null)	120	Treasury
...	...	...

FULL JOIN (==FULL OUTER JOIN)

- Not implemented in MySQL
- A join that returns all the rows where there is a match (left or right)
- If no match, the result is NULL on the respective side - left or right
- Разликата с Cartesion product е че в долния пример няма дуплициране на данни
- Returns all records in both tables regardless of **any** match
    - Less useful than **INNER**, **LEFT** or **RIGHT JOINs**
    - We can use **UNION** of a **LEFT** and **RIGHT JOIN**



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

```
SELECT E.EMPLOYEE_NAME, D.DEPARTMENT_ID, D.DEPARTMENT_NAME
FROM EMPLOYEES E
FULL OUTER JOIN DEPARTMENTS D
ON (E.DEPARTMENT_ID = D.DEPARTMENT_ID);
```

EMPLOYEE_NAME	DEPARTMENT_ID	DEPARTMENT_ NAME
Jennifer Whalen	10	Administration
Michael Hartstein	20	Marketing
Clara Vishney	(null)	(null)
Jason Mallin	(null)	(null)
(null)	150	Shareholder Services

FULL JOIN == FULL OUTER JOIN – взема всичко **включително и сечението**, но не е имплементирано в MySQL

В MySQL нямаме нито FULL OUTER JOIN нито само FULL JOIN, и в MySQL тази операция се постига чрез UNION of LEFT and RIGHT JOIN.

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

Self joining

Self join means to join a table to itself.
Always used with table aliases.

```
SELECT CONCAT (E.FIRST_NAME, ' ', E.LAST_NAME) AS EMPLOYEE_NAME,
SELECT CONCAT (M.FIRST_NAME, ' ', M.LAST_NAME) AS MANAGER_NAME
FROM EMPLOYEES E
JOIN EMPLOYEES M ON (E.MANAGER_ID = M.EMPLOYEE_ID);
```

EMPLOYEE_NAME	MANAGER_NAME
Neena Kochhar	King
Lex De Haan	King
Alexander Hunold	De Haan
Bruce Ernst	Hunold

Three way join

A three-way join is a join of three tables.

```
SELECT E.EMPLOYEE_ID, L.CITY, D.DEPARTMENT_NAME
FROM EMPLOYEES E
JOIN DEPARTMENTS D ON D.DEPARTMENT_ID = E.DEPARTMENT_ID
JOIN LOCATIONS L ON D.LOCATION_ID = L.LOCATION_ID;
```

EMPLOYEE_ID	CITY	DEPARTMENT_ NAME
103	Southlake	IT
104	Southlake	IT
124	San Francisco	Administration
...	...	...

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
  - Not used often as **a lot of duplication**

```
SELECT E.LAST_NAME, D.DEPARTMENT_NAME
FROM EMPLOYEES E
CROSS JOIN DEPARTMENTS D;
```

6.4. Set operations

Set operations in SQL is a type of operations which allows **the results of multiple queries to be combined into a single result set**.
Set operators in SQL include UNION, INTERSECT, and EXCEPT, which mathematically correspond to the concepts of union, intersection and set difference.

UNION operator

sales2005

person	amount
Joe	1000
Alex	2000
Bob	5000

sales2006

person	amount
Joe	2000
Alex	2000
Zach	35000

```
SELECT * FROM sales2005
UNION
SELECT * FROM sales2006;
```

Note that there are two rows for Joe because those rows are distinct across their columns. There is only one row for Alex because those rows are not distinct for both columns.

person	amount
Joe	1000
Alex	2000
Bob	5000
Joe	2000
Zach	35000

`UNION ALL` gives different results, because it will not eliminate duplicates. Executing this statement:

```
SELECT * FROM sales2005
UNION ALL
SELECT * FROM sales2006;
```

person	amount
Joe	1000
Joe	2000
Alex	2000
Alex	2000
Bob	5000
Zach	35000

INTERSECT operator

The following example INTERSECT query returns all rows from the Orders table where Quantity is between **50 and 100.**

```
SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 1 AND 100

INTERSECT

SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 50 AND 200;
```

EXCEPT operator

The following example EXCEPT query returns all rows from the Orders table where Quantity is between 1 and 49, and those with a Quantity between 76 and 100.
Worded another way; the query returns all rows where the Quantity is between 1 and 100, apart from rows where the quantity is between 50 and 75.

```
SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 1 AND 100

EXCEPT

SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 50 AND 75;
```

6.5. 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;
```

6.5. Subqueries example - more

Given tables Employees with columns Id, Name, Email and Salary, EmployeeDeparments with columns Id, EmployeeId, DepartmentId and Departments with columns Id, Name write an SQL query to return the names of Deparments along with the number of employees in each department as Name and Count columns
Note: Use LEFT joins

```
SELECT D.Name,
(SELECT count(*) FROM EmployeeDeparments EEDD
WHERE EEDD.DepartmentId = ED.DepartmentId AND EEDD.EmployeeId IS NOT NULL) AS `COUNT`
FROM Departments D

LEFT JOIN EmployeeDeparments ED
ON ED.DepartmentId = D.Id

LEFT JOIN Employees E
ON ED.EmployeeId = E.Id

GROUP BY D.Name;
```

Групира се таблица/резултат от JOIN-овете, и там реда с Administrative си стои и е 1 на брой. Във вложената селект заявка **ED.DepartmentId** представлява тази съща резултатна таблица като в случая на Administrative записа **ED.DepartmentId** е **null**. Съществува и друга опция даден служител да не се води в никой отдел - **ED.DepartmentId** може да не null, а **EEDD.EmployeeId** да е null.

```
Name            COUNT
--------------  ----------
Administrative       0
Business Develo  1
HR                   2
IT                   1
```

—----------

6.6. 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;
```

7. DCL - Data Control Language

```
Grant SELECT,INSERT,UPDATE,DELETE on Employee To User1;

Revoke INSERT On Employee To user1;

Deny Update On Employee to user1;
```

GRANT in first case we gave privileges to user User1 to do SELECT, INSERT, UPDATE and DELETE on the table called employees.

REVOKE with this command we can take back privilege to default one, in this case, we take back command INSERT on the table employees for user User1.

DENY is a specific command. We can conclude that every user has a list of privilege which is denied or granted so command DENY is there to explicitly ban you some privileges on the database objects.:

8. Глобални настройки / стойности

```
SELECT DATABASE(); --Returns the name of the current database
SELECT USER(); --Returns the current MySQL user and host
SELECT VERSION(); --Returns the version of MySQL you are using
```

За смяна на root паролата без парола да е

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;

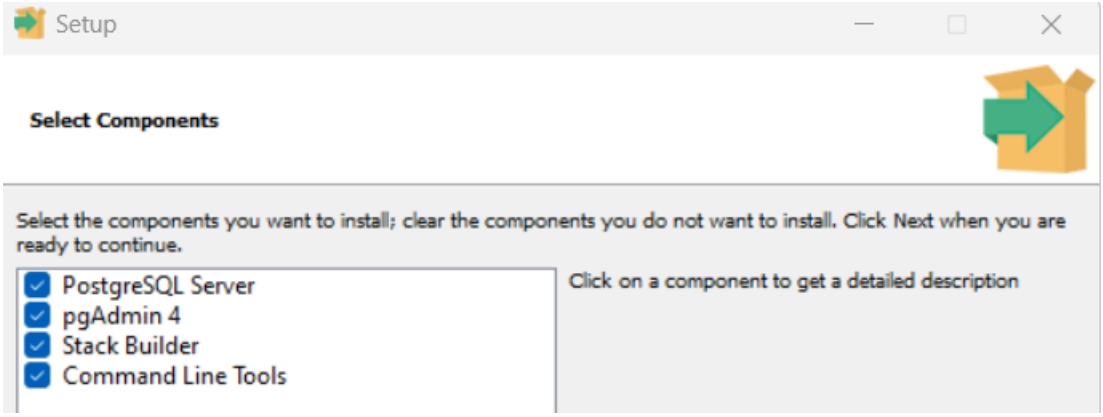
SET sql_mode = 'ONLY_FULL_GROUP_BY'; **- активен този mode**
SET sql_mode = '';- премахни only full group by – да го използваме в Judge, но няма ефект ☹

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!!!

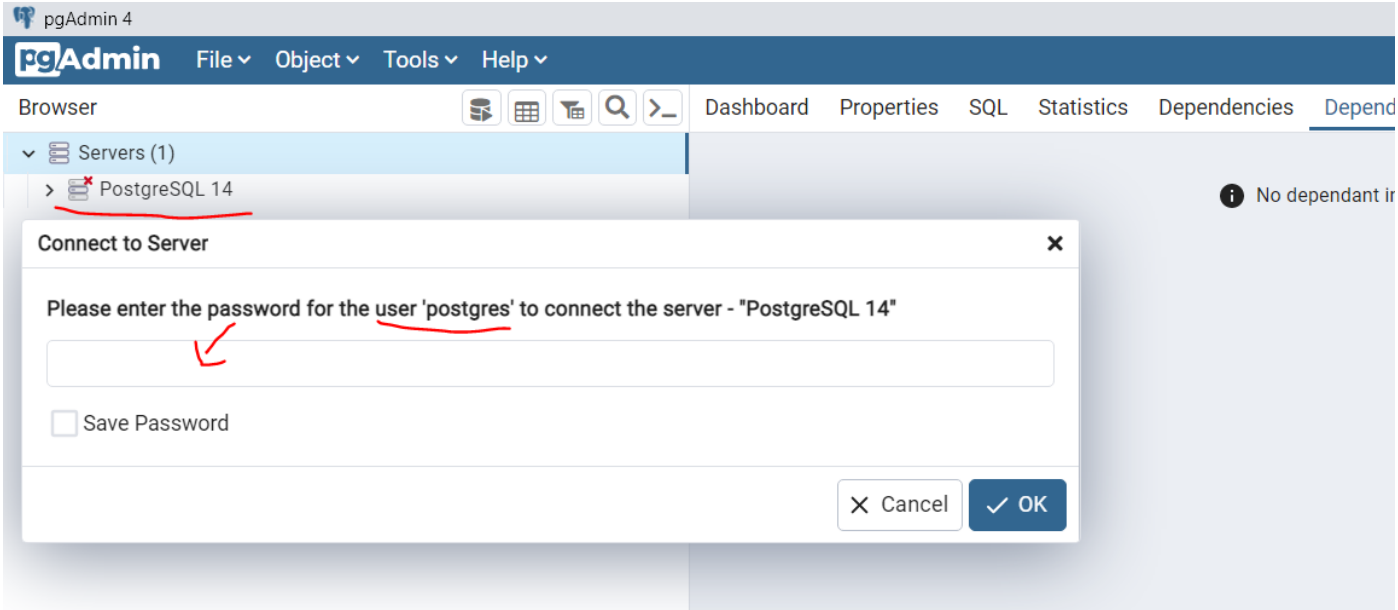
GROUP_CONCAT
ANY_VALUE(muhaha) **AS** `bla_bla_bla`;

IV. Installing PostgreSQL

https://www.postgresqltutorial.com/postgresql-getting-started/install-postgresql/

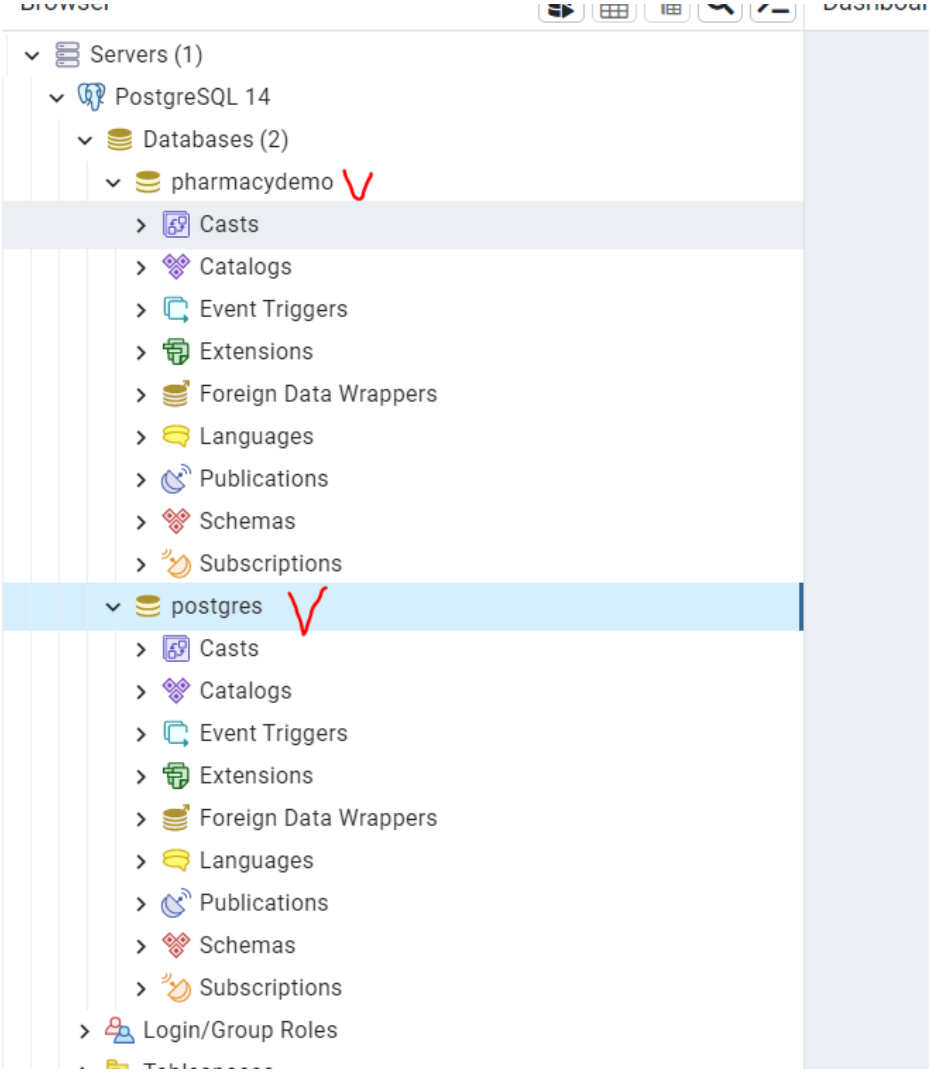


Creating new database

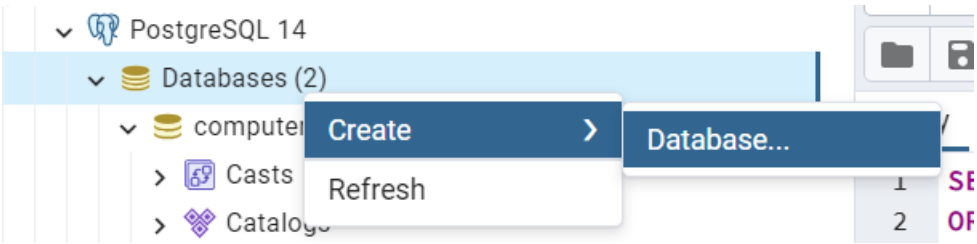


Мога да си създавам new database на server PostgreSQL 14 (user: postgres pass: 1), и да ползвам schema public винаги, и така няма проблем.

```
quarkus.datasource.db-kind=postgresql
quarkus.hibernate-orm.database.generation=update
quarkus.datasource.username = postgres
quarkus.datasource.password = 1
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/pharmacydemo
```



Вратката е да си създадеш **ръчно и предварително!!! (за разлика от WorkBench където става автоматично)**
Database -> Create -> Database … със съответното име!



И след това няма проблем

```
driverClassName: org.postgresql.Driver
url:
jdbc:postgresql://localhost:5432/computerstore?allowPublicKeyRetrieval=true&useSSL=false&createDatabaseIfNotExis
t=true&serverTimezone=UTC
username: postgres
password: 1111
```

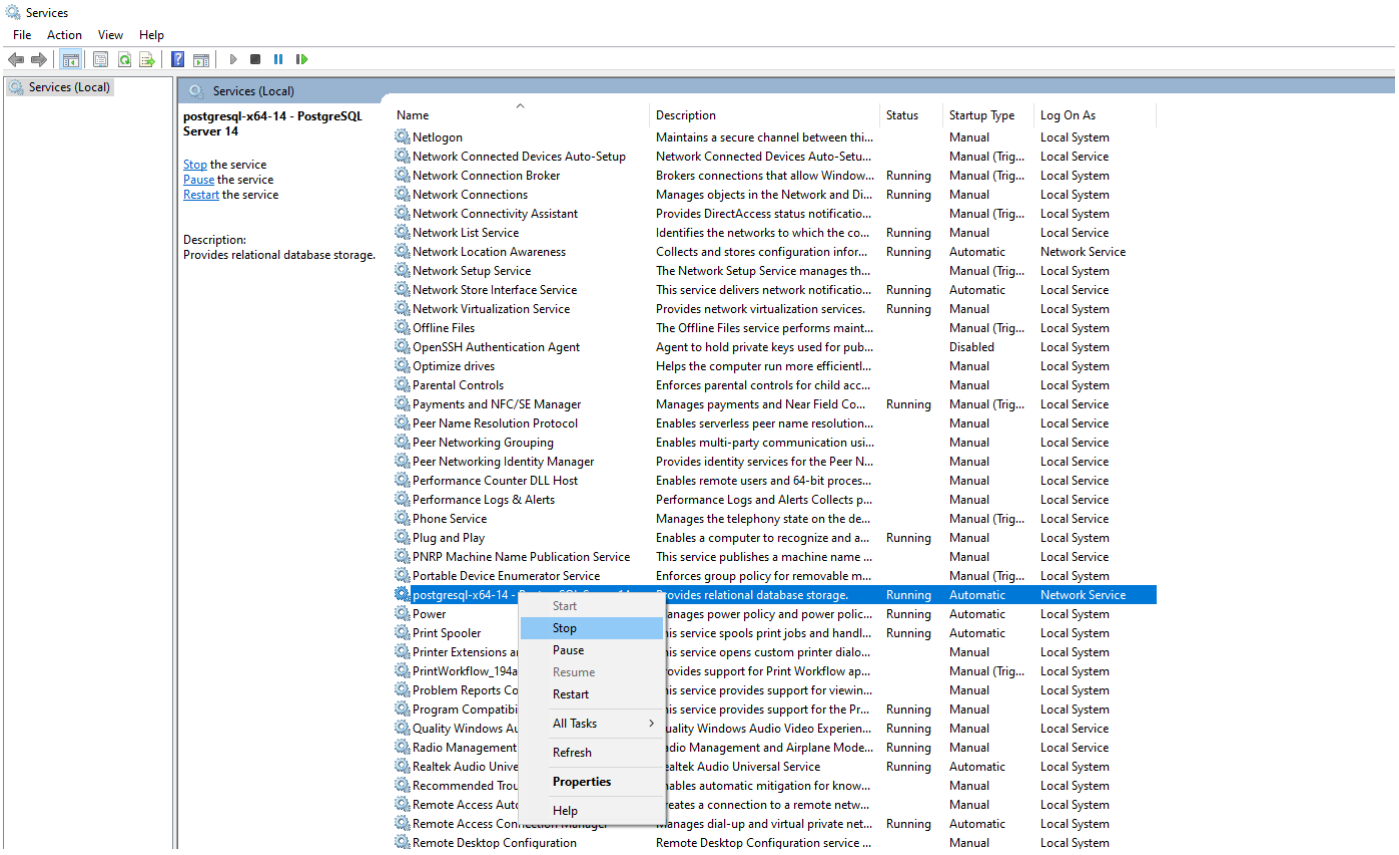
Stopping postgres server on Windows

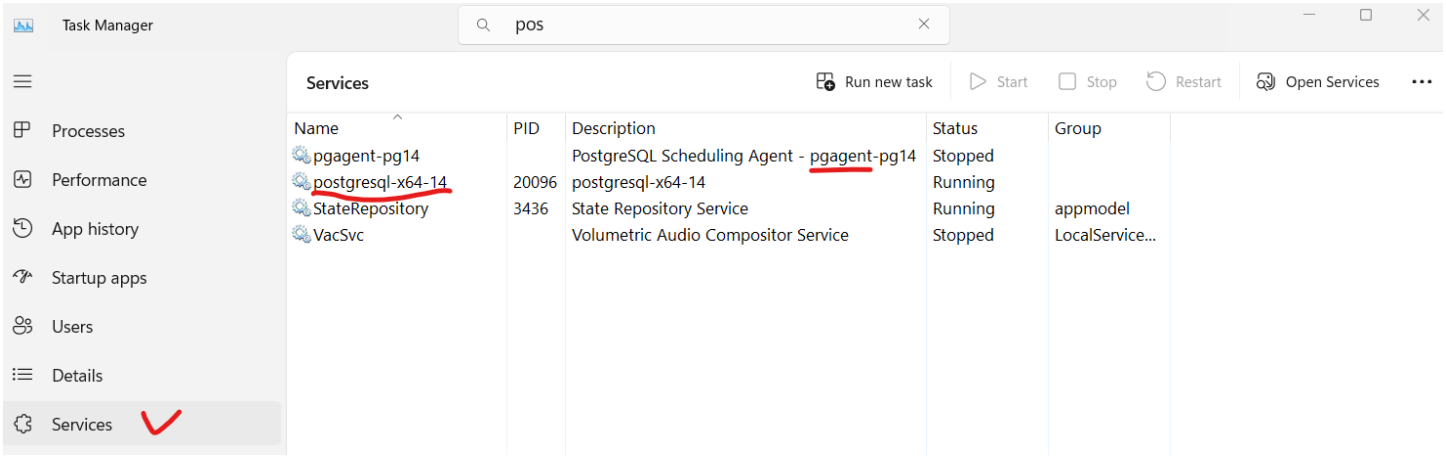
Когато сървър базата се инициализира в Докер/облачно за даден проект, то трябва да спрем ръчно съществуващия service (database server) на Windows.

При Windows за да пуснем контейнер, то трябва да използваме Docker Desktop (не знам дали има background приложение на Docker за Windows).

Докато при Линукс може и да не използваме Docker Desktop, а само с бекграунд приложение на Docker.

1. Open Run Window by Winkey + R.
2. Type **services.msc**
3. Search Postgres service based on version installed.
4. Click stop, start or restart the service option.





V. PostgreSQL vs MySQL

В общи линии, всичко каквото има MySQL, го има и в PostgreSQL.
Но не всичко, което е в PostgreSQL го има в MySQL.

Връщане на резултата от заявката

При MySQL ще се наложи да направим втора заявка за гетване на обновения запис.

```
insert into USERS(username, email, password)
values('spas2', 'spas2@gmail.com', 'password')
returning *;
```

Nested SELECT subqueries

SELECT statements can be nested in the where clause.

- Correlated and Uncorrelated subqueries
- Note: If a subquery and a JOIN can accomplish the same task, it is often **more efficient to use a JOIN**! - понеже базата прави оптимизации при JOIN, а не прави оптимизации при **nested subqueries**
- **When two or more different tables, subquery possible only in PostgreSQL!**

Subquery in the SELECT clause

```
SELECT firstName, lastName,
(SELECT departmentName FROM Department WHERE Department.departmentId =
Employees.departmentId) AS departmentName
FROM Employees;
```

Subquery in the FROM clause

```
SELECT AVG(emloyeeCount)
FROM (
   SELECT COUNT(*) AS employeeCount
   FROM Employees
   GROUP BY departmentId
) AS subQuery;
```

Subquery in the WHERE clause

Correlated subquery

```
SELECT E.firstName, E.lastName, E.departmentId
FROM Employees E
WHERE EXISTS ( using the EXISTS function
   SELECT *
   FROM Department D
   WHERE D.departmentId = E.departmentId AND D.departmentName = 'Department 1'
);

The argument of EXISTS is an arbitrary SELECT statement, or subquery. The subquery is
evaluated to determine whether it returns any rows. If it returns at least one row, the
result of EXISTS is "true";
```

UPDATE JOIN

Updating data - using data from other tables

PostgreSQL can use also subqueries:

```
UPDATE table_name
SET column1 = (SELECT column2 from other_table WHERE condition)
WHERE condition;
```

MySQL cannot use subqueries, but join:

```
UPDATE table_name
JOIN other_table
ON table_name.column1 = other_table.column2
SET table_name.column1 = new_value
WHERE condition;
```

DELETE JOIN

Deleting data using Joins

PostgreSQL can use also subqueries:

```
DELETE FROM Emloyees
WHERE id IN (SELECT id FROM Departments WHERE condition);
```

MySQL cannot use subqueries, but join:

```
DELETE Emloyees, Departments FROM Emloyees
JOIN Departments ON Employees.id = Departments.id
WHERE condition;

DELETE empl FROM Emloyees AS empl
JOIN Departments ON empl.id = Departments.id
WHERE condition;
```

ON CONFLICT (like a trigger)

PostgreSQL

```
explain INSERT INTO Employees(id, firstname, lastname, departmentid, dateofbirth,
employeecardid, status)
VALUES(34, 'Jo', 'Doe', 1111, '1990-01-01', 1234, 'Available')
ON CONFLICT (id) DO UPDATE SET
status='Blocked',
timestamp=now();
```

MySQL

```
explain INSERT INTO Employees(id, firstname, lastname, departmentid, dateofbirth,
employeecardid, status)
VALUES(34, 'Jo', 'Doe', 1111, '1990-01-01', 1234, 'Available')
ON DUPLICATE KEY UPDATE
status='Blocked',
timestamp=now();
```

```
A row to be inserted would cause a duplicate value in a UNIQUE index or PRIMARY KEY
//An INSERT ... ON DUPLICATE KEY UPDATE statement against a table having more than one
unique or primary key is also marked as unsafe. (Bug #11765650, Bug #58637)
```

Numbers in PostgreSQL vs numbers in Java

32 bit signed integer - positive and negative: +(2^31-1) and -(2^31):

В java това е int
В PostgreSQL това е int4

64 bit signed integer - positive and negative: +(2^63-1) and -(2^63):
В java това е long
В PostgreSQL това е int8

BigDecimal
В java - BigDecimal
В PostgreSQL - numeric или decimal

Name	Storage Size	Description	Range
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point

В postgresql няма разлика между decimal(19, 2) и numeric(19 ,2), но в други SQL има разлика.

numeric(19,2)

nextval() setval()

nextval() взема следващ номер в поредицата на postgresql

```
create sequence hibernate_sequence start 1 increment 1;

create table if not exists loyalty_level (
                                    id int8 not null,
                                    level int4 not null,
                                    points int8 not null,
                                    primary key (id)
);

insert into loyalty_level(id, level, points)
values
(nextval('hibernate_sequence'), 1, 0),
(nextval('hibernate_sequence'), 2, 20),
(nextval('hibernate_sequence'), 3, 35),
(nextval('hibernate_sequence'), 4, 55),
```

Друг начин за вкарване на данни

```
create sequence if not exists Tag_SEQ increment by 50;
select setval('Tag_SEQ',  (SELECT MAX(id) FROM Tag));

insert into Tag (id, name)
select nextval('Tag_SEQ'), 'KYC_GAMEPLAY_BLOCK'
where not exists(select id from tag where name = 'KYC_GAMEPLAY_BLOCK');
```

temporary sequence

```
-- First you create temporary sequence. It will get dropped when the session is ended.
create temporary sequence s_milestone;

-- Next you assign the milestones with a simple update query
update loyalty_level
set milestone = nextval('s_milestone')
where level in (1,6,11,16,21)
order by level asc;
```

Cursor (in MySQL)

Еквивалента на **курсор** е **итератор** когато говорим за колекции!

```
CALL `hrm`.`print_employee_details`();

CALL `hrm`.`print_employee_info`('test@test.com');

DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `print_employee_info`(email varchar(200))
BEGIN
    DECLARE v_name varchar(100);
    DECLARE v_email varchar(254);
    DECLARE emp_cursor CURSOR FOR
    select Name, Email from employees where email = email;
    OPEN emp_cursor;
        FETCH emp_cursor into v_name, v_email;
        SELECT CONCAT(v_name, ' ', v_email);
    close emp_cursor;
END$$
DELIMITER ;
```

Procedure

POSTGRESQL syntaxis

```
alter table if exists loyalty_level
   add column milestone int4 not null default 0;

DO
$$
   DECLARE
       next_milestone int4;
       current_level int4;
       level_config_size int4;
   BEGIN
       current_level := (SELECT ll.level FROM loyalty_level AS ll ORDER BY ll.level ASC
LIMIT 1);--BO can update levels by deleting level number 1, so we take the lowest present level
number
       next_milestone := 1;
       level_config_size := (SELECT COUNT(*) FROM loyalty_level);

       WHILE (current_level <= level_config_size) LOOP
           IF current_level IN (1,6,11,16,21)
           THEN
               --levels between milestone levels also have milestones
               update loyalty_level
               set milestone = next_milestone
               where level >= current_level;
```

```
            next_milestone := next_milestone + 1;--next milestone always increase with 1
        END IF;

        current_level := current_level + 1;--levels always increase with 1
    END LOOP;
END
$$;
```

Using cast()

In PostgreSQL we can use cast

```
SELECT cast(p.id as text) FROM Profile p WHERE p.status IN (?1)
```

COALESCE function syntax

PostgreSQL

The COALESCE function accepts an unlimited number of arguments. It returns the first argument that is not null. If all arguments are null, the COALESCE function will return null.

```
update wallet w
set lifetime_deposit = coalesce((select sum(d.amount)
                    from deposit d
                    where d.wallet_id = w.id
                    and d.balance_change_reason in ('PLAYER_DEPOSIT',
'PLATFORM_MIGRATION')
                         and d.status = 'ACCEPTED'), 0);
```

SELECT in WITH clause - only in PostgreSQL

Common Table Expression (CTE) subquery

The basic value of `SELECT` in `WITH` is to break down complicated queries into simpler parts. An example is:

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
),
top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

which displays per-product sales totals in only the top sales regions. The `WITH` clause defines two auxiliary statements named `regional_sales` and `top_regions`, where the output of `regional_sales` is used in `top_regions` and the output of `top_regions` is used in the primary `SELECT` query. This example could have

been written without `WITH`, but we'd have needed two levels of nested sub-`SELECT`s. It's a bit easier to follow this way.

Common functions

String functions:

MySQL: CONCAT(), SUBSTRING(), LENGTH(), UPPER(), LOWER(), TRIM(), REPLACE()
PostgreSQL: CONCAT(), SUBSTRING(), LENGTH(), UPPER(), LOWER(), TRIM(), REPLACE()

Mathematical functions:

MySQL: ABS(), CEIL(), FLOOR(), ROUND(), SQRT(), LOG(), EPX(), POW(), RAND()
PostgreSQL: ABS(), **CEILING()**, FLOOR(), ROUND(), SQRT(), **LN()**, EXP(), **POWER()**, **RANDOM()**

Date and time functions:

MySQL: NOW(), CURDATE(), CURTIME(), DATE(), TIME(), YEAR(), MONTH(), DAY(), HOUR(), MINUTE(), SECOND(), EXTRACT(YEAR from date), EXTRACT(MONTH from date), EXTRACT(DAY from date),

PostgreSQL: NOW(), CURRENT_DATE, CURRENT_TIME, DATE(), TIME(),
EXTRACT(YEAR from date), EXTRACT(MONTH from date), EXTRACT(DAY from date),
EXTRACT(HOUR FROM time), EXTRACT(MINUTE FROM time), EXTRACT(SECOND FROM time)

Conditional expressions:

MySQL: IF(), CASE
PostgreSQL: CASE, COALESCE(), NULLIF()

Regular expressions:

MySQL: REGEXP, NOT REGEXP

PostgreSQL:
~
!~
~*
!~*

```
SELECT firstname FROM employees WHERE firstname ~ 'a'; --Returns names containing 'a'
SELECT firstname FROM employees WHERE firstname !~ 'a'; --Returns names not containing 'a'
```

Array to String

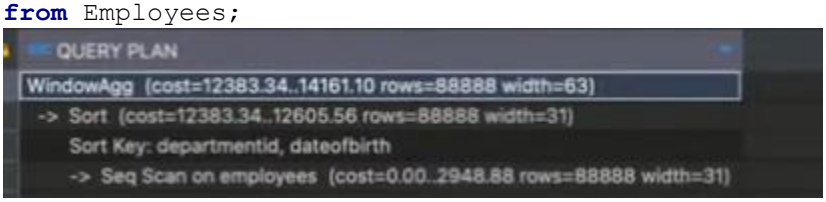
In PostgreSQL

```
SELECT ARRAY_TO_STRING(ARRAY[1, 2, 3, 4], ',');
SELECT STRING_TO_ARRAY('1,2,3,4', ',');
SELECT ARRAY_LENGTH(ARRAY[1, 2, 3, 4], 1);
```

Window aggregation

Both for MySQL and PostgreSQL

```
select FirstName, Salary,
avg(Salary) over (PARTITION by DepartmentId order by DateOfBirth)
from Employees;
```



Глобални команди

```
SELECT CURRENT_DATABASE(); --Returns the name of the current database
SELECT CURRENT_USER; --Returns the name of the current user
SELECT VERSION(); --Returns the version of PostgreSQL you are using

SYSTEM information functions;
```

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.

Stored routines

- Set of SQL statements that perform complex operations
- Stored in the database
- Invoked with the CALL statement
- **Stored routines include the typical programming language features like variables, conditional statements, loops, exceptions, procedures, functions**

- Stored procedures are named procedures that can be executed repeatedly on the database server:
  - Combine SQL statements and programming logic
  - Can take parameters
  - Usually implement the database logic (data-related business rules)
- Functions are stored routines that return a value

- Stored routines allow for improved performance
  - Data should not be moved out of the server in order to be processed
- Stored routines allows for easier maintenance, data integrity and security:
  - When the database structure is changed, these changes can reflect only the stored procedure

- Applications should access the DB only through the stored routines

User-Defined Functions in MySQL

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;   - декларирание 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`
```

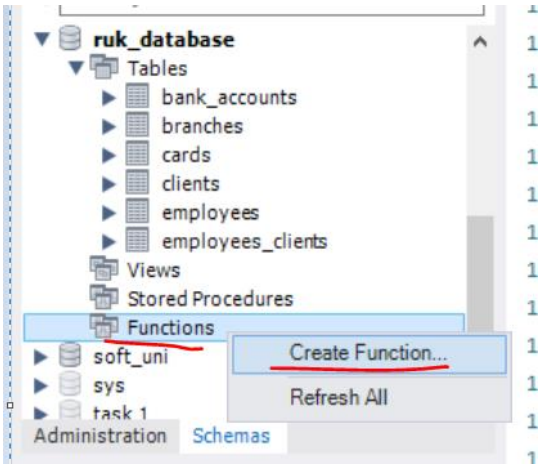
```
        INNER JOIN `towns` AS t ON t.`town_id` = a.`town_id`
        WHERE t.`name` = `town_name`));

END;
$$$$
```

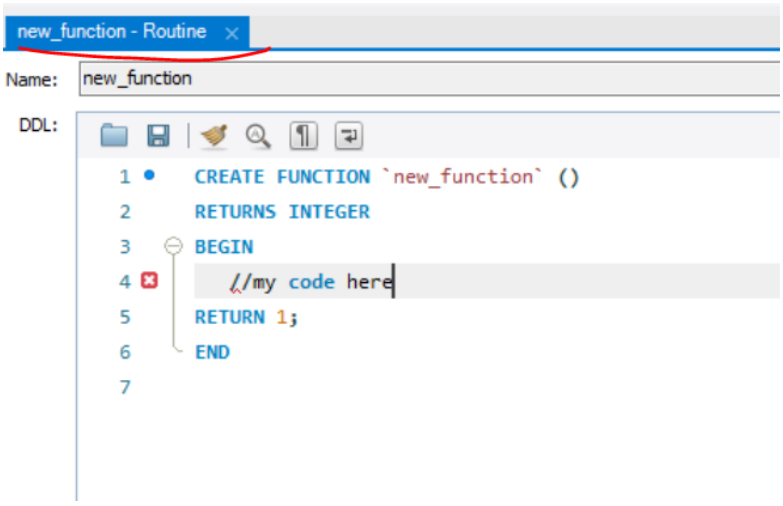
Deterministic vs. non-deterministic functions

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;
```





Stored Procedures in MySQL

Info

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**

- Parameters of the stored procedure can additionally be defined as:
  - **IN** – parameter has initial value when passed to the procedure, but is not modified after procedure finishes
  - **OUT** – parameter might be modified by the procedure and used with modified value after procedure finishes
  - **INOUT** – parameter has an initial value and might be modified by the procedure

- There are two ways to define variables in a stored routine
  - Using the **SET** command – defines a user variable accessible even outside the stored procedure/function
  - Using the DECLARE command which may define:
    - Local variables
    - Error conditions and handlers
    - cursors

- The format for declaring a variable with DECLARE is:

DECLARE <variable_name> <datatype> [DEFAULT <value>];

DECLARE class_size INT DEFAULT 30;

- SET variables are visible outside the routine procedure while DECLARE variables are visible only in the body of the procedure

BEGIN
    SET **@class_size** = 30; в самата процедура
END;

CALL RegisterStudentForClass(); извикваме процедурата
SELECT **@class_size**; извън процедурата е достъпно

Example:

```
CREATE PROCEDURE AssignGrade(student_id INT, class_id INT, grade VARCHAR(2))
BEGIN
    DECLARE enrolled INT;
    SELECT COUNT(*) INTO enrolled
    FROM Students s
    WHERE id = student_id AND class_id = class_id;

    IF enrolled > 0 THEN
        UPDATE Students
        SET grade = grade
        WHERE id = student_id AND class_id = class_id;
    ELSE
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Student is not enrolled in the class';
    END IF;
END

CALL AssignGrade(1234, 1, 'A');
```

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

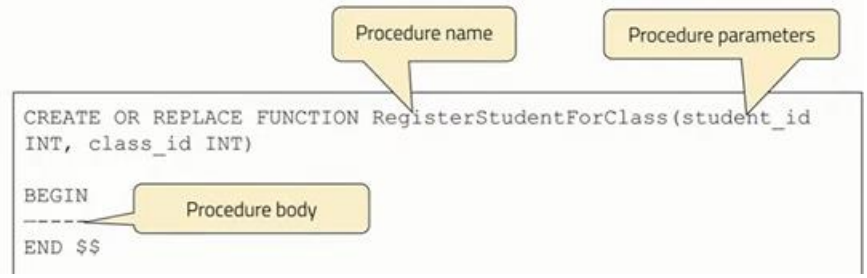
Все едно процедурата работи като функция:

```
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.

Stored Procedures in PostgreSQL

В PostgreSQL сторната процедура е в кавички, тъй като там работими само с функции



Prepared statements

- Offer performance and security benefits
- Prepared statements can be used to create dynamic SQL queries and execute them
- Dynamic queries are created as plain text and can contain parameters that are passed when the dynamic query is executed

MySQL statement syntax

```
PREPARE stmt_name FROM ‘statement’;
EXECUTE stmt_name USING  @variable;
```

DEALLOCATE PREPARE stmt_name;

Пример:
SET @student_id = 1234;
SET @name = 'John Doe';
SET @email = 'john.doe@example.com'

PREPARE stmt FROM 'INSERT INTO students (student_id, name, email) VALUES (?, ?, ?)';
EXECUTE stmt USING @student_id, @name, @email;

PostgreSQL statement syntax

PREPARE stmt_name (datatype, …) AS
SELECT ….;
EXECUTE stmt_name(value1, ….);

Пример:
PREPARE stmt (INT, VARCHAR(100), VARCHAR(100)) AS
INSERT INTO students (student_id, name, email) VALUES ($1, $2, $3);

EXECUTE stmt(1234, 'John Doe', 'john.doe@example.com');

- Prepared statements can be deleted with the DEALLOCATE PREPARE command;

DEALLOCATE PREPARE stmt_name;

TCL - Transactions – a kind of a stored procedure

Info

Modern DBMS servers have built-in transaction support

- Implement "ACID" transaction
- **ACID** means:
  - **A**tomicity – the whole transaction is treated as a single unit. Either all the operations are executed, or none are.
  - **C**onsistency – the database should transition from one consistent state to another
  - **I**solation – concurrent transactions should be executed in such a way that they appear to be serialized
  - **D**urability – once a transaction is committed, its changes are permanent

- 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
  - All changes in a transaction are temporary, but changes become **final** when COMMIT is executed
  - At any time all changes can be cancelled by ROLLBACK
- Transactions can be cancelled by issuing the command ROLLBACK
  - State of the data after ROLLBACK:
    - Data changes are undone
    - Previous state of the data is restored
    - **Locks** on the affected rows are released
- 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.

Specific case

В MySQL при първа операция валидна и втора операция НЕвалидна, все пак се осъществява partial update след като COMMIT-нем. Т.е. за да не се изпълни нищо, то ни трябва ИЗРИЧЕН rollback като работим с MySQL.

При PostgreSQL същият сценарий ролбаква всичко / нищо не се изпълнява ако някъде има грешка – без да се налага изричен запис ROLLBACK!

MySQL

```
START TRANSACTION;
UPDATE Students
SET date_of_birth  =
'1997-10-10' WHERE first_name  =
'John';
COMMIT;
```

PostgreSQL

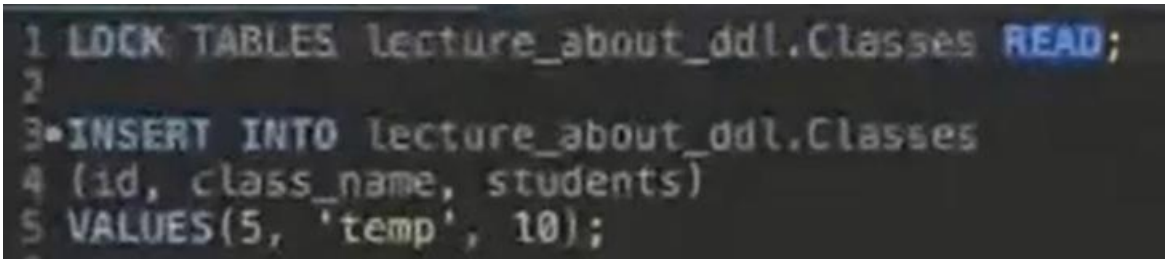
```
BEGIN;
UPDATE Students
SET date_of_birth  =
'1996-10-10' WHERE first_name  =
'John';
COMMIT;
```

Table locks

- Table locks
    - Prevent destructive interaction between concurrent transactions
    - Require no user action
    - Automatically use the lowest level of restrictiveness
    - Are held for the duration of the transaction
- When a user needs a locked data, it waits until it is unlocked

- Most RDBMS support the so called AUTOCOMMIT mode that runs each query in a separate transaction – this is also the default behavior in MySQL (so far we didn't have to issue COMMIT after each INSERT/UPDATE or DELETE query)
- In order to disable AUTOCOMMIT mode in MySQL, you have to issue
  SET AUTOCOMMIT = 0;

Пример:
Задаваме му да е READ, и като се опитваме да вкарваме/ъпдейтваме данни, и гърми.



select * from pg_stat_activity; показва статистика за трансакции и други неща

select * from pg_catalog.pg_locks; показва статистика за локове

Pessimistic locking vs. Optimistic locking

In one corner, we've got pessimistic locking — the cautious guardian that assumes the worst and secures the resource upfront, ensuring exclusive access. Picture it as someone reserving an entire row of seats at the movies just in case the theater gets crowded, leading to potential waiting times but guaranteeing no seat-stealing shenanigans.

On the flip side, optimistic locking takes a more relaxed approach. It's like attending a festival without assigned seating — multiple folks can enjoy the show simultaneously, but before the final act, organizers make sure nobody's accidentally grabbed someone else's popcorn. If conflicts arise, it's back to the drawing board for a retry or rollback.

Isolation levels

Info

- Define how transactions interact with each other
- Ensure data consistency in a multi-transactional world
- Four isolation levels with different trade-offs
- Each RDBMS provides a mechanism for specifying an isolation level for a transaction

- There are three phenomena that can occur during concurrently executing transactions:
  - **Dirty reads** – A dirty read occurs when one transaction reads data that has been modified (but not committed) by another transaction.
  - **Non-repeatable read** – a transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data
  - **Phantom reads** – a transaction re-runs a query and finds that another committed transaction has added/deleted or updated some rows

- The stronger isolation ensures better consistency but works slower and the data is locked for a longer period of time – 4 standartized levels:

Level of isolation	Dirty reads	Repeatable reads	Phantom reads
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

Read uncommited – искаме да имаме Dirty reads, Repeatable reads и Phantom reads

….

….

Serializable – всички са на опашка, и чакат една по една да приключат – най-високото ниво на изолация решаващ и трите проблема

Обикновено се използват **Read committed** или **Repeatable read**

Isolation levels – table locks

- Table level
  - **Locks the whole table** – примерно ак искаме да вкараме нова колона с данни в дадената таблица
  - Better for bulk operations (INSERT, UPDATE, DELETE) or loading data
  - Other transactions cannot modify data

- Column level (няма по подразбиране в MySQL/PostgreSQL) – custom изолация е това/не се използва почти!
  - Restrict access to individual column
  - Used rarely when we need to update only one column
  - Most RDBMS do not support true column level lock

- Row level – искаме да си заключим някои редове само, а останалите да са на разположение
  - Fine-grained and allowing high concurrency
  - Used to edit specific records in a transactional way

- Селектирана колона за update операция

- An isolation level can be set for all transactions in the current or all sessions with the SET TRANSACTION statement – must be the first statement in a transaction

SET LOCAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;

- Tables can be explicitly locked with LOCK TABLES command

LOCK TABLES <table> <lock_type (e.g. READ or WRITE)>, [… <table> <lock_type>]

LOCK [TABLE] [ONLY] name [*] [, …] [IN lockmode MODE] [NOWAIT]

- Rows can be explicitly locked with the SELECT FOR UPDATE STATEMENT

SELECT * FROM EMPLOYEES FOR UPDATE;

```
begin;
select * from lecture_about_ddl.Students_3 where  last_name  = 'Joe' for update ;

UPDATE lecture_about_ddl.Students_3  SET date_of_birth  = '2001-10-10' WHERE last_name  = 'Joe';


commit'
```

- Table level locks are useful for preventing excessive locking on rows or the occurrence of a deadlock

Examples

Само намаля в случая:

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

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;
```

Transactions in Stored Procedures

- Transactions are not bound to a BEGIN … END block:
  - one block may have multiple transactions
  - one transaction can have multiple blocks
- Some operations are non-transactional (such as writing to a file) – these operations are not reversed when transaction fails to commit

#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);
```

#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);
```

Transactions with triggers

- Transactional operations performed in a trigger way are atomic in regard to the statement that invoked the trigger:
    - If the statement fails all transactional operations in the trigger are reversed
    - If it commits – all transactional operations in the trigger also commit

Triggers – като Event Listener

Info

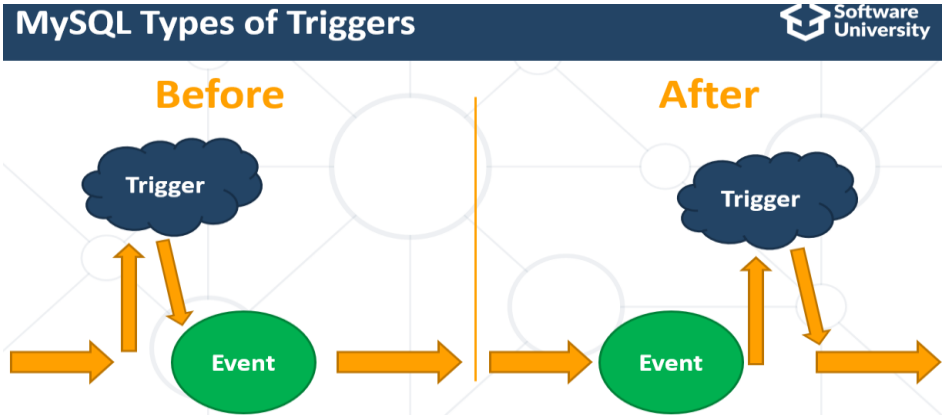
Event-listener – чакаме нещо да се случи, и тогава прави нещо

- 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

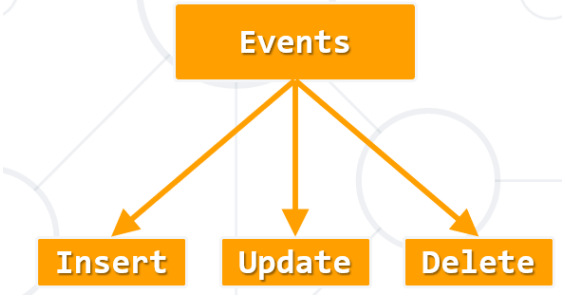
- Triggers may be executed at the following points of time:

- **before** a row is added/deleted/modified
- **after** a row is added/deleted/modified

- Triggers are typically used for:
  - Logging information about data changes to the tables
  - Archiving data
  - Rejecting table manipulations if some criteria is not met
  - Checking data before/after manipulations
  - Showing users a message when a command is executed



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!**

Creating a trigger

- Triggers are created with the CREATE TRIGGER command

CREATE [**DEFINER** = {user | CURRENT_USER}] **TRIGGER** <trigger_name>
<trigger_time> <trigger_event> да се изпълнява на определено време и/или вида на ивента
ON <tbl_name> **FOR EACH ROW**
<trigger_body>

Пример MySQL:

```
CREATE TRIGGER after_enrollment_insert
AFTER INSERT
ON enrollments
FOR EACH ROW
```

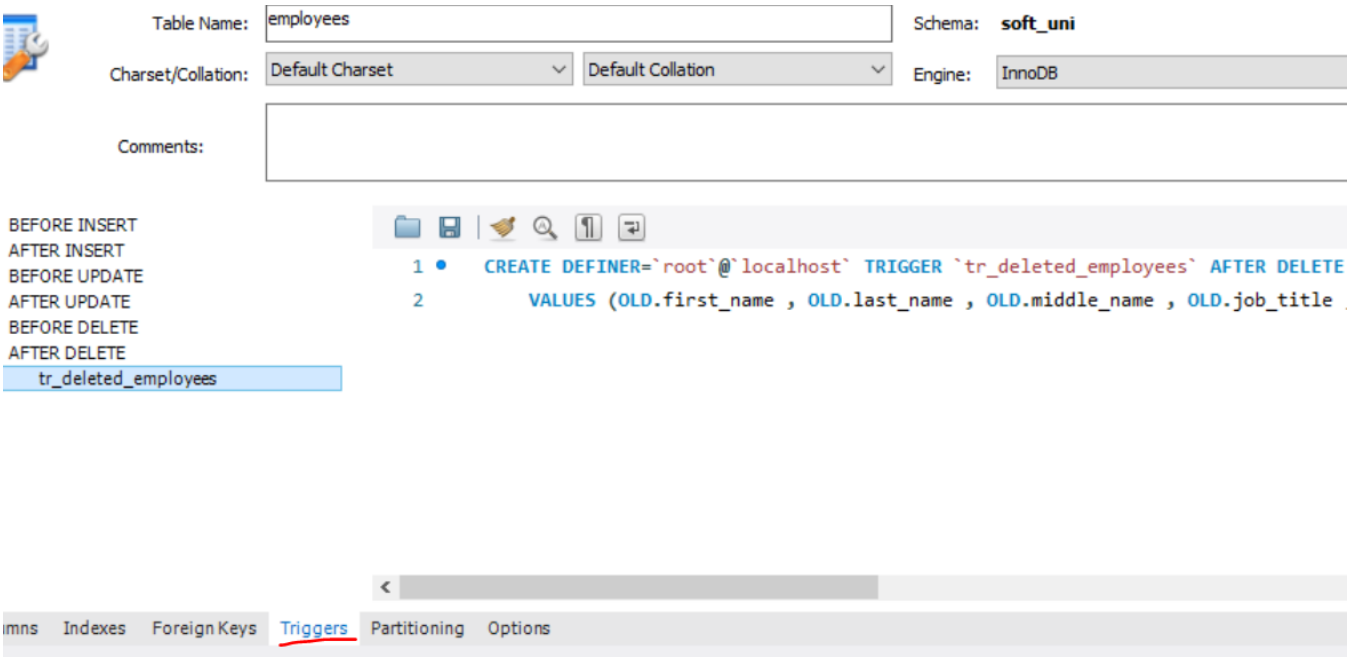
```
BEGIN
        INSERT INTO audit_log(operation_type, log_details)
        VALUES('INSERT', CONCAT("New student has been enrolled with studentId:",
        NEW.student_id,
        "in classId:", NEW.class_id));
END;

INSERT INTO enrollments (student_id, class_id, grade)
VALUES (99, 1, 'A');
```

Пример PostgreSQL:

```
CREATE TRIGGER after_enrollment_insert
AFTER INSERT
ON enrollments
FOR EACH ROW execute function log_enrollment();  We need a function in PostgreSQL!!!


CREATE OR REPLACE FUNCTION log_enrollment()
RETURNS TRIGGER AS $$
BEGIN
        INSERT INTO audit_log(operation_type, log_details)
        VALUES('INSERT', CONCAT("New student has been enrolled with studentId:",
                NEW.student_id, "in classId:", NEW.class_id));
        RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

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;

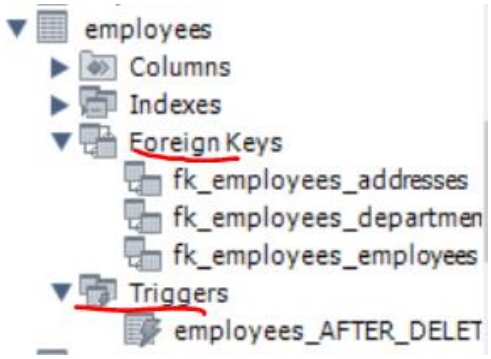
After Update

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
```

Аналози в JPA Hibernate

https://www.baeldung.com/jpa-entity-lifecycle-events

Аналог на trigger в JPA са entity lifecycle events @PrePersist, @PostPersist, @PreRemove, @PostRemove, *@PreUpdate, @PostUpdate, @PostLoad*
Въпросът е под капака Hibernate тригери ли създава?
Отговорът е НЕ: тригери се създават като обекти в базата данни, а въпросните анотации следят в самото Java приложение.

Error handling

- Errors can occur during the execution of stored routines – most often returned from an SQL statement executed by the stored routine
- Errors can be handled by error handlers that are basically another stored routines that provide error-handling logic

- There are basic three types of errors in a stored procedure:
  - Named system errors
  - Unnamed user-defined errors
  - Named user-defined errors

- Name of errors can be attached to error codes using the **DECLARE … CONDITION** command

DECLARE **no_such_table** CONDITION FOR 45000;

DECLARE CONTINUE HANDLER FOR **no_such_table**
BEGIN
-- Body of handler
END;

```
39    IF enrolled > 0 THEN
40        UPDATE Students
41        SET grade = grade
42        WHERE id = student_id AND class_id = class_id;
43    ELSE
44        SIGNAL SQLSTATE '45000'
45        SET MESSAGE_TEXT = 'Student is not enrolled in the class';
46    END IF;
47 END
```

In PostgreSQL

```
24 CREATE OR REPLACE FUNCTION AssignGrade(studentId INT, classId INT, studentGrade VARCHAR(2
25 RETURNS VOID AS $$
26 DECLARE
27     enrolled INT;
28 BEGIN
29     SELECT COUNT(*) INTO enrolled
30     FROM lecture_about_ddl.enrollments
31     WHERE enrollments.student_id = studentId AND enrollments.class_id = classId;
32
33     IF enrolled > 0 THEN
34         UPDATE lecture_about_ddl.enrollments
35         SET lecture_about_ddl.enrollments.grade = studentGrade
36         WHERE lecture_about_ddl.enrollments.student_id = studentId AND enrollments.class_
37     ELSE
38         RAISE EXCEPTION 'Student is not enrolled in the class';
39     END IF;
40 END; $$
41 LANGUAGE plpgsql;
```

Events

- Relational database systems typically provide mechanism for scheduled execution of tasks
- MySQL events are tasks that run according to a schedule
- Oracle database provides the DBMS_JOB PL/SQL package for creating scheduled jobs
- PostgreSQL does not provide a built in scheduler and user has to rely on **pg_cron** plugin

MySQL way

- In order to create an event (scheduled job) in MySQL the event_scheduler thread must be enabled:

SET GLOBAL **event_scheduler=on;**

- In order to check that the event_scheduler thread is running, you can display all current MySQL processes using:

SHOW PROCESSLIST;

- After the MySQL scheduler process is enabled, you can schedule jobs by creating events (an event is created in the current database)
- There are two types of events:
  - One time events – executed only once
  - Repeating events – executed multiple times

- General syntax for creating an event:

CREATE [DEFINER = {user | CURRENT_USER}] EVENT [IF NOT EXISTS] event_name
ON SCHEDULE schedule

```
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'comment']
DO event_body;
```

```
CREATE EVENT clear_old_logs
ON SCHEDULE EVERY 1 MONTH
DO
DELETE FROM audit_log WHERE log_date > NOW() – INTERVAL 1 MONTH;
```

Database Metadata

- Database metadata refers to **information about the database objects**
- MySQL and PostgreSQL provide various utilities to retrieve database metadata

MySQL

- MySQL list all databases:

```
SHOW DATABASES;
```

- MySQL list all tables in a database:

```
SHOW TABLES FROM database_name;
```

- MySQL describe a table:

```
DESCRIBE table_name;
```

- MySQL get column information for a table:

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'table_name';
```

- MySQL get currently selected database:

```
SELECT DATABASE();
```

- MySQL list all indexes in a table:

```
SHOW INDEX FROM table_name;
```

- MySQL how many selects/inserts/updates are already executed so far:

```
SHOW status WHERE variable_name = 'Com_select';
SHOW status WHERE variable_name = 'Com_insert';
SHOW status WHERE variable_name = 'Com_update';
```

PostgreSQL

- PostgreSQL list all databases:

```
SELECT datname FROM pg_database;
```

- PostgreSQL list all tables in a database:

```
SELECT tablename FROM pg_tables WHERE schemaname == 'public';
```

- PostgreSQL list all schemas:

```
SELECT schema_name FROM information_schema.schemata;
```

- PostgreSQL describe a table:

```
SELECT * FROM information_schema.columns
WHERE table_name = 'table_name';
```

- PostgreSQL get column information for a table:

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'table_name ';
```

- PostgreSQL describe indexes:

```
SELECT * FROM pg_indexes
WHERE tablename = 'table_name';
```

- PostgreSQL access control: listing roles and their priviliges:

```
SELECT grantee, privilege_type
FROM information_schema.role_table_grants
WHERE table_name = 'table_name';
```

- PostgreSQL check table constraints:

```
SELECT * FROM information_schema.table_constraints
WHERE table_name = 'table_name';
```

- PostgreSQL list all views in a database:

```
SELECT table_name FROM information_schema.views
WHERE table_schema = 'public';
```

- PostgreSQL get database information:

За проверка на умрели връзки в базата примерно.

```
SELECT * FROM pg_stat_activity;
SELECT * FROM pg_stat_database;
```

- PostgreSQL view function definitions:

```
SELECT proname, prosrc FROM pg_proc
WHERE pronamespace = (SELECT oid FROM pg_namespace WHERE nspname = 'table_name');
```

- PostgreSQL how many selects/inserts/updates are already executed so far:

```
SELECT sum(calls) AS total_select_queries
FROM pg_stat_statements
WHERE query LIKE 'SELECT%' and userid = pg_backend_pid();
```

- PostgreSQL see role information

```
SELECT rolname, rolsuper, rolcreaterole, rolecreatedb, rolcanlogin
FROM pg_roles;

WHERE rolname = 'teaching_assistant';  за определена роля
```

Users and Permissions

Users

- Both MySQL and PostgreSQL manage permissions through a system of user accounts and privileges.
- MySQL uses a privilege system that includes static global privileges, dynamic global privileges, database privileges, table privileges, column privileges, and routine privileges.
- PostgreSQL uses a role-based access control system. In PostgreSQL, a role can be a user or a group, and privileges can be assigned to roles.

- Multiple users can be created within the database with the CREATE USER command in MySQL

CREATE USER ‘username’@’host’ IDENTIFIED BY ‘password’;

CREATE USER ‘newUser’@’localhost’ IDENTIFIED BY ‘password’

- When we need to modify user data we use ALTER USER in MySQL

ALTER USER ‘newUser’@’localhost’ IDENTIFIED BY ‘newPassword’

- Respectively multiple roles can be created within the database with CREATE ROLE command in PostgreSQL

CREATE ROLE username LOGIN PASSWORD ‘password’;

CREATE ROLE testLogin LOGIN password ‘testlogin’;

SELECT * FROM pg_catalog.pg_roles
WHERE rolname = ‘testlogin’;

- Respectively we can use ALTER ROLE in PostgreSQL

ALTER ROLE username WITH PASSWORD ‘newpassword’;

- When we need to rename a user we use RENAME USER in MySQL

RENAME USER ‘username’@’host’ TO ‘new_user_name’@’host’;

- Respectively we can use ALTER ROLE in PostgreSQL

ALTER ROLE username TO ‘new_name’;

- When we need to remove a user we use DROP USER in MySQL

DROP USER ‘username’@’host’;

- Respectively we can use DROP ROLE in PostgreSQL

DROP ROLE username;

Permissions

- Permissions are certain privileges that can be assigned to users – ако няма права, то даден дб потребител не може да изпълни определени операции.
- Permissions are given with the GRANT command
- Permissions are removed with the REVOKE command

- Granting privileges in MySQL

GRANT **SELECT, INSERT, DELETE** ON database.table TO ‘username’@’host’;

GRANT SELECT, INSERT ON classes TO ‘username’@’host’;

- Respectively in PostgreSQL

GRANT SELECT, INSERT ON TABLE table TO username;

- Listing priviliges in MySQL

SHOW GRANTS FOR ‘newUser’@’localhost’;

- PostgreSQL покажи права/listing:

\du username това е ако се свържем през CLI към нашата база

Покажи права за user testlogin
SELECT * FROM pg_catalog.pg_roles
WHERE rolname = ‘testlogin’;

- Revoking privileges in MySQL

REVOKE DELETE on classes FROM ‘username’@’host’;

- Revoking privileges in PostgreSQL

REVOKE DELETE ON TABLE table FROM username;

- Granting permission with resource limits in MySQL – fine grained in comparison with PostgreSQL

CREATE USER ‘student’@’localhost’ IDENTIFIED BY ‘password’;
GRANT ALL PRIVILEGES ON school_db.* TO ‘student’@’localhost’;

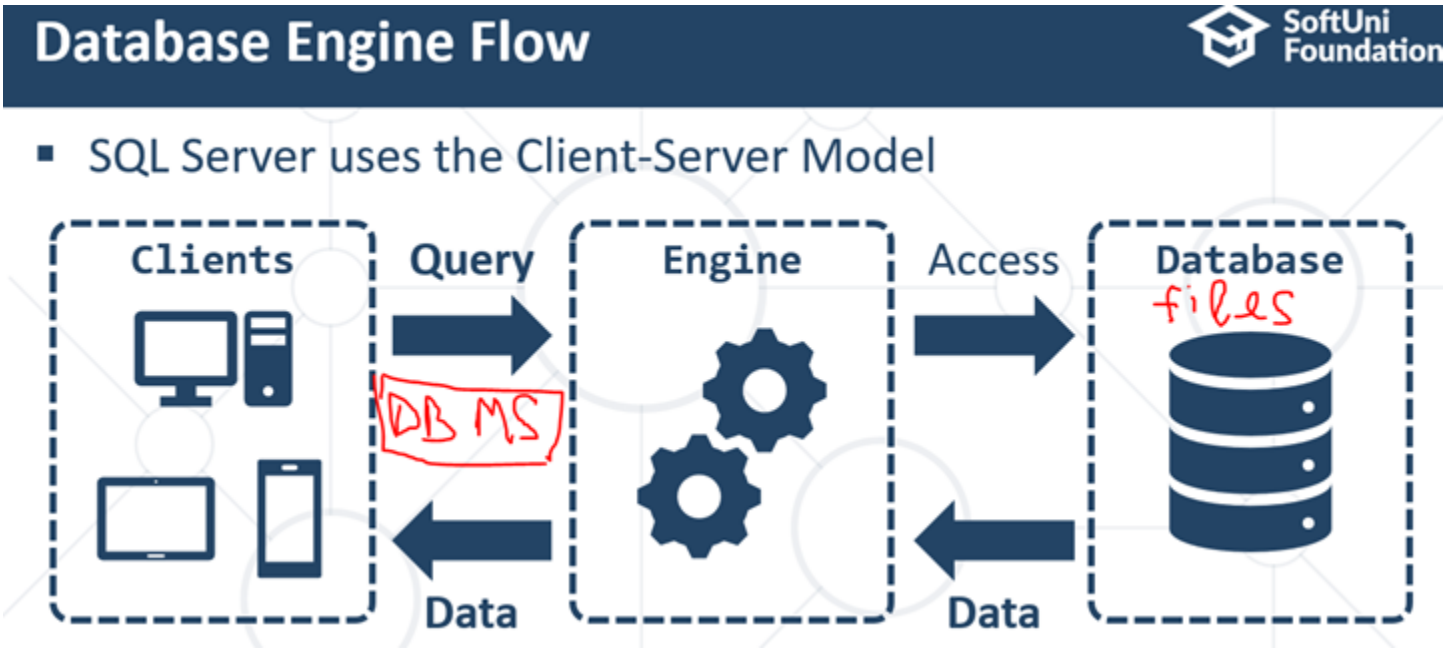
ALTER USER ‘student’@’localhost’
WITH MAX_QUERIES_PER_HOUR 100
MAX_UPDATES_PER_HOUR 50
MAX_CONNECTIONS_PER_HOUR 20
MAX_USER_CONNECTIONS 10;

- PostgreSQL does not allow same as for MySQL resource limitations to be set

-- Grant all priovileges on all tables in a specific schema
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO student;

ALTER ROLE student CONNECTION LIMIT 10;

VI. NoSQL



1. General info

- NoSQL - stands for "**Not only SQL**", non-tabular databases
- First used in 1998
- Tackle (да се пребори с) problems of performance, volume and variety of data
- Handling of unstructured or semi-structured data
- Scale horizontally - добавяме все повече и повече load-ове, партитишъни, клъстъри и сървъри (вместо повече процесор и рам памет - вертикално скалиране)
- Flexibility, scalability and speed.

2. NoSQL vs Relational Model

Similarities:

- Both are used for storing and managing data
- Both support CRUD (Create, Read, Update, Delete) operations
- Both provide mechanisms for ensuring data consistency and integrity

Differences:

- **Schema:** Relational databases have a fixed schema while NoSQL databases are usually schema-less.
- **Scalability**: Relational databases traditionally scale vertically while NoSQL databases are designed to scale horizontally
- **Consistency**: Relational databases provide strong consistency (ACID [Atomicity-Consistency-Isolation-Durability] properties) while NoSQL databases often offer eventual consistency
- **Data model**: Relational databases use tables with rows and columns for data storage while NoSQL databases use a variety of data models: including key-value, document, columnar and graph.

NoSQL Databases (Non-Relational Databases)

A NoSQL database, has dynamic schema for unstructured data

- има задължителни полета, но има и полета, които може да не са попълнени
- Може да се променя формата на данните, които се съхраняват чрез Dynamic data – това е плюс за бизнес, който често сменя нещата в базата си от данни поради характера на бизнеса си.

https://www.xplenty.com/blog/the-sql-vs-nosql-difference/

SQL могат и хоризонтално, но е по-трудно

NoSQL – в 3 през нощта 300 000 души се записват и искат да се логнат. Тогава се активират хоризонтални shardins/клъстери/паралелни сървъри и те успяват да запишат всички 300 000 души, които се регистрират 2 3 през нощта.

3. Fundamental concepts in NoSQL

CAP Theorem

- We can guarantee two out of these three properties:
  - **C**onsistency
  - **A**vailability
  - **P**artition Tolerance

- **Consistency**: every read gets the most recent write or error
- **Availability**: every request receives a response without guaranteeing it contains most recent write (последно записаната информация) - т.е. ако даден клъстър/сървър падне, то ще вземем информацията от друг сървър и потребителят няма да разбере, че получава неактуални стари данни
- **Partition Tolerance**: system can continue operation even if there are communication breakdown between nodes

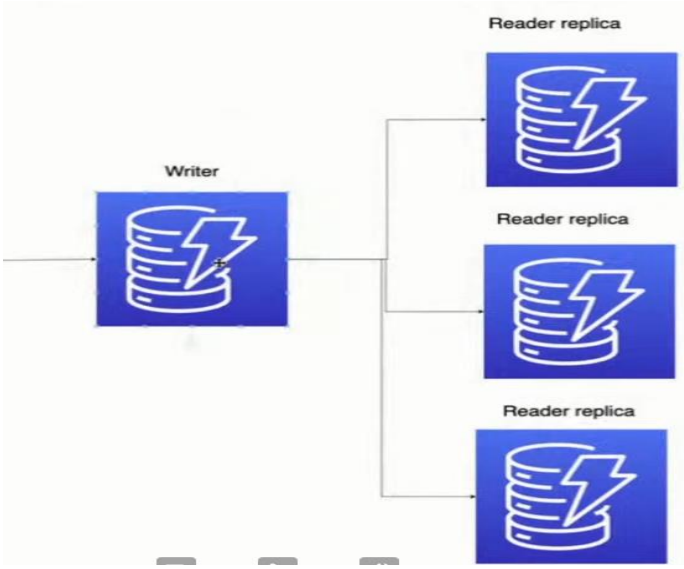
CAP Theorem - Impact

- If a system chooses consistency and availability (CA), then it can't tolerate network partitions
- If a system that opts for consistency and partition tolerance (CP), then it may not always be available
- If a system chooses availability and partition tolerance (AP), then it may occasionally serve stale or inconsistent old data
- Network partitions are almost inevitable in distributed systems (срив в комуникацията е напълно възможно да се случи) and **most databases prioritize partition tolerance** plus the expense of availability **or** consistency

Consistency

Уточнение какво имаме в предвид под сървър. Сървърно приложение, което вдига инстанция на база данни. Т.е. Когато си пуснем локално база данни е една инстанция. Но на практика могат да бъдат много инстанции на едно и също приложение. Начина по който си комуникират е въпрос на архитектура. Примерно в Кубернетис може да имаме няколко клъстъра за един и същи микросървис, по-точно говорейки за бази данни няколко клъстъра/аплиакции за една и съща база данни.

- **Data is the same across all nodes at any given time.**
  - A write operation on one Node will be reflected by reading data from any node - като запишем една информация, то от който и да е Reader Replica Node трябва да достъпим същите новозаписани данни



- **Strong consistency**
  - Write operation is not considered complete until the changes have been propagated to all nodes - т.е. докато не се запише записа **в/на** всяка една reader replica, то не го считаме за complete. Т.е. ако някоя reader replica е долу, може да чакаме доста или може и изобщо да не се финализира/осъществи записа
  - Higher latency and less availability

- **Eventual consistency**
  - Write operation is considered complete once it's written to any node - или в нашия пример когато информацията се запише в основния Writer replica, то не ни интересува дали веднага се е пропагирала към останалите Reader replica.
  - Changes are propagated to other nodes eventually/в последствие
  - Stale/old data read possible

Partitioning

- Breaking up large datasets into smaller ones `partitions` or also called sharding
- Each partition is stored in a separate database, node or server
- Better distribution of read/write loads and improved scalability and performance

Комуникацията между всяка реплика/партитишън най-често се осъществява чрез хеш кода за бързина.

Types of Partitioning

Horizontal (Sharding)

- Data is split **into rows** and each set of rows is stored in different partition - but **same number of columns**!!
- Each partition forms a separate table with same schema and number of columns

Vertical

- Dividing a table into smaller tables with subset of data
- Each partition contains a **different set of columns, and same number of rows possible**
- Used to split rarely accessed or wide columns

Partitioning strategies

Range partitioning:

- Created based on a certain range of the partition key (PK). E.g. Employees with IDs from 1 to 10 000 in one partition and from 10 001 to 20 000 in another

Hash partitioning:

- Applying a hash function to some attribute which results in a partition number. Ensures a **more balanced distribution of data**.

List partitioning:

- DBA explicitly maps each partition to a list of discrete values of the partition key.

Partitioning - Challenges

- Data distribution to be balanced, otherwise we get what is called `**hot partitions**`, т.е. всички данни отиват в един партитишън, което не е добре - няма да бъде възможно да се възползваме от перформънса. Например ако primary key е first_name и има много Антоновци да речем.
- Data locality is crucial and operations involving multiple partitions can be slow
- Partitioning adds complexity and makes Database management and tuning harder.

Schema-less Design

- Traditional relational databases need to have the schema defined in advance and adhered (придържам / спазвам) strictly
- NoSQL provide Shema-less or schema-flexible models
- **Structure of data is not predefined -** т.е. можем да работим без схема в самото начало/ не е необходимо и да указваме тип на данните/на колоните
- **We can change the structure of the data on the fly**

Benefits:

- **Flexibility:** Schema-less models are highly flexible, enabling quick adaptation to changes.
- **Speed and Efficiency:** The lack of a rigid(твърда) schema allows for faster development and easier scalability
- **Complex and Hierarchical Data:** Schema-less models can easily store and manage complex, nested, or hierarchical data, which can be difficult to represent in a rigid schema

Challenges:

- **Inconsistency:** The flexibility of a schema-less model can lead to data inconsistency if not managed correctly.
- **Lack of Data Integrity:** In a rigid/constant schema, constraints can be applied to ensure data integrity. In a schema-less model, this responsibility often moves to the application layer. Т.е. ще се наложи в нашето Java приложение да си правим ръчни проверки за null value, за тип на колоната, и т.н. тъй като базата може да ни върне нещо което не би следвало да го обработваме. (местим един вид логика/функционалност от базата към външно приложение)
- **Query Complexity:** Without a consistent schema, querying data can become more complex. Например липсват JOIN в доста от NoSQL езиците, правим по-скъпи операции, и т.н.

NoSQL databases and implementation of schema-less design:

- **Document Databases** store data in a flexible, JSON-like format. Each document can have its own unique structure. The schema can be changed in real-time, allowing fields to be added or removed as necessary.
- **Key-Value Stores** have an entirely schema-less design. The value associated with a key can be of any type, and there's no structure enforced on the values
- **Wide Column Stores** do have a schema, but it is flexible. You can easily add or remove columns in a column family without having to modify the rest of the data
- **Graph Databases** also have flexible schemas. While they do enforce a certain level of structure (with nodes and relationships), the properties of those nodes and relationships can be very flexible

Scaling

Types of scaling:

- **Vertical Scaling**, or "**scaling up**", involves increasing the capacity of a single machine. You can do this by adding more powerful CPUs, adding more RAM, increasing disk storage, or improving network capacity

- **Horizontal Scaling**, or "**scaling out**", involves adding more machines to the system and distributing the load among them. This is the approach typically employed by NoSQL databases.

Normalization

Нормализацията е контра продуктивна при NoSQL. Данните ни трябва да не са нормализирани.
Можем да имам всички данни в една таблица, и вече примерно в Java или JS приложението да си ги обработваме.
Нормализацията е свързана също така и с JOIN операции, които дори и да ги има в NoSQL то са доста скъпи. Т.е. няма JOIN-ове, значи няма и нормализация.

4. Types of NoSQL databases

Key-Value

- Store data as a collection of key-value pairs, similar to a Map
  - Key is unique identifier
  - The value can be anything: number, string, **JSON**, BLOB, etc

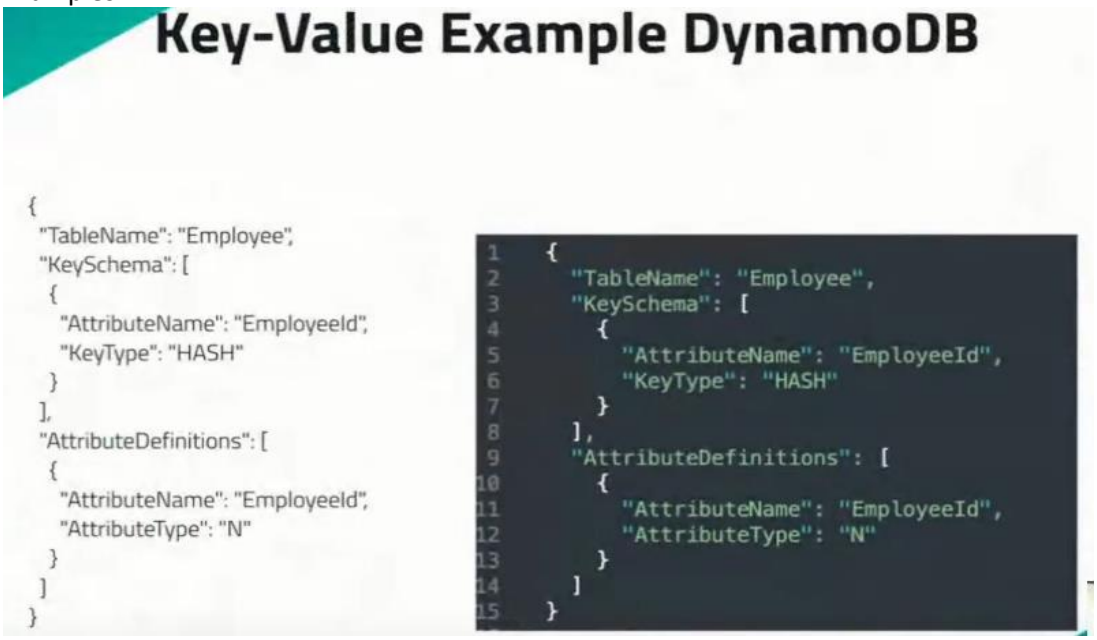
- Example: **Redis, DynamoDB** - използват само/главно key-value

- Use cases:
  - Constant, fast, read/write operations, e.g. caching, session management, shopping carts, etc

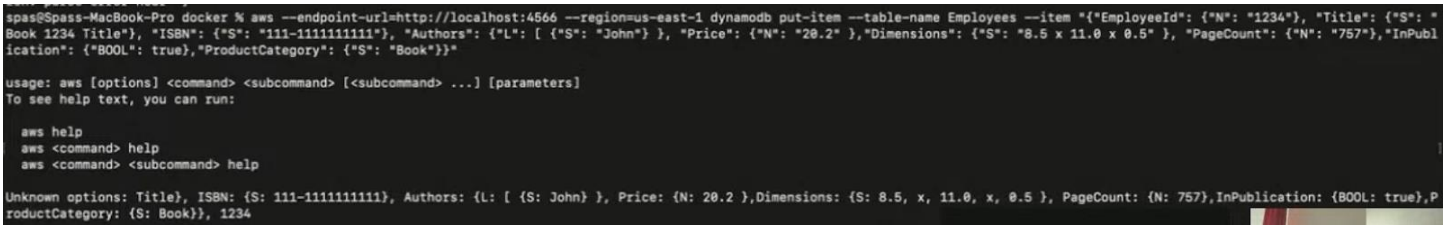
Limitations:

- No complex querying
- No relationships
- No joining
- Need to handle data structure at application level

Examples:



CLI(Command Line Interface) of DynamoDB via AWS



Document Databases

Subcategory of key-value

Store data in a semi-structured format (often **JSON**)

- Allow for nested data
- Each document has a unique key identifier and a value

Examples: **MongoDB** and CouchDB and **ElasticSearch**(https://www.elastic.co/)

Use cases:

- Content management systems
- Real-time analytics
- IoT

Limitations:

- Comlex transactions
- Data integrity across documents (joins and complex multi-document transactions)

В MongoDB липсва използването на join-ове!

CLI of MongoDB

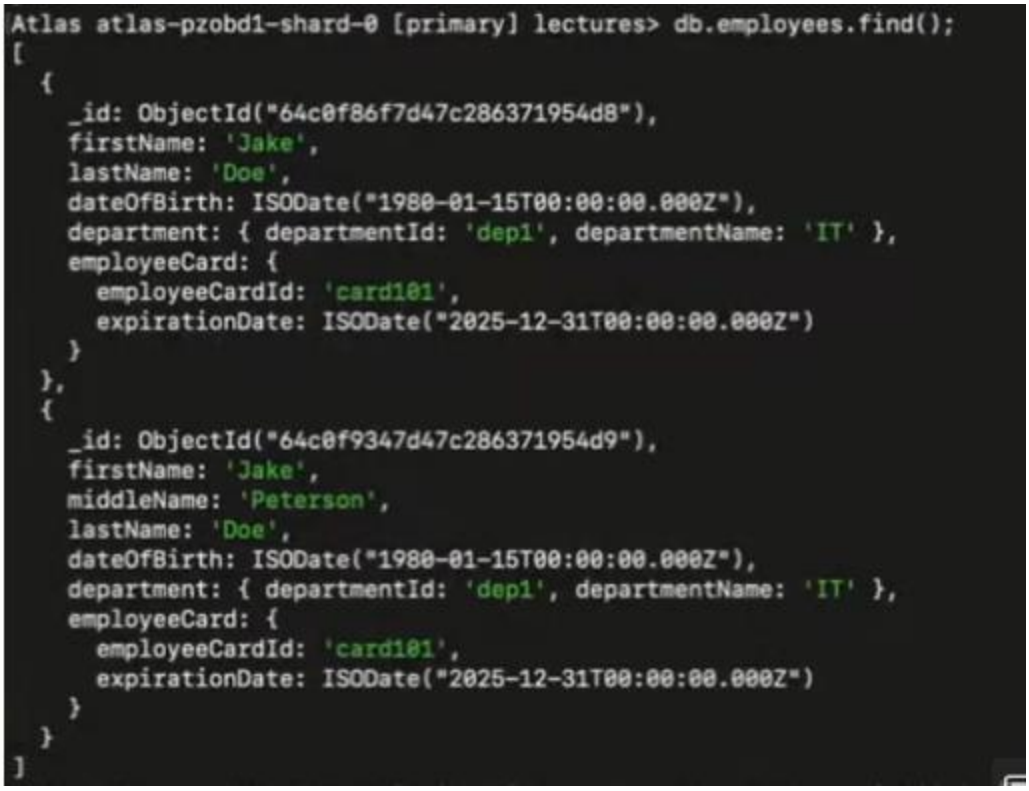
```
Atlas atlas-pzobd1-shard-0 [primary] test> use lectures
switched to db lectures
Atlas atlas-pzobd1-shard-0 [primary] lectures> db.employees.insertOne({
...     firstName: "John",
...     lastName: "Doe",
...     dateOfBirth: new Date("1980-01-15"),
...     department: {
...         departmentId: "dep1",
...         departmentName: "IT"
...     },
...     employeeCard: {
...         employeeCardId: "card101",
...         expirationDate: new Date("2025-12-31")
...     }
... });
{
  acknowledged: true,
  insertedId: ObjectId("64c0ba4c7d47c286371954d7")
}
```

```
Atlas atlas-pzobd1-shard-0 [primary] lectures> db.employees.find({_id: ObjectId("64c0ba4c7d47c286371954d7")});
[
  {
    _id: ObjectId("64c0ba4c7d47c286371954d7"),
    firstName: 'John',
    lastName: 'Doe',
    dateOfBirth: ISODate("1980-01-15T00:00:00.000Z"),
    department: { departmentId: 'dep1', departmentName: 'IT' },
    employeeCard: {
      employeeCardId: 'card101',
      expirationDate: ISODate("2025-12-31T00:00:00.000Z")
    }
  }
]
```

```
Atlas atlas-pzobd1-shard-0 [primary] lectures> db.employees.updateOne({_id: ObjectId("64c0ba4c7d47c286371954d7")}, {$set: {firstName: "Spas"}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
Atlas atlas-pzobd1-shard-0 [primary] lectures> db.employees.deleteOne({_id: ObjectId("64c0ba4c7d47c286371954
{ acknowledged: true, deletedCount: 1 }
Atlas atlas-pzobd1-shard-0 [primary] lectures>
```

След като сме изтрили един запис, само .find() ни намира всички записи в момента



```
Atlas atlas-pzobd1-shard-0 [primary] lectures> db.employees.find();
[
  {
    _id: ObjectId("64c0f86f7d47c286371954d8"),
    firstName: 'Jake',
    lastName: 'Doe',
    dateOfBirth: ISODate("1980-01-15T00:00:00.000Z"),
    department: { departmentId: 'dep1', departmentName: 'IT' },
    employeeCard: {
      employeeCardId: 'card101',
      expirationDate: ISODate("2025-12-31T00:00:00.000Z")
    }
  },
  {
    _id: ObjectId("64c0f9347d47c286371954d9"),
    firstName: 'Jake',
    middleName: 'Peterson',
    lastName: 'Doe',
    dateOfBirth: ISODate("1980-01-15T00:00:00.000Z"),
    department: { departmentId: 'dep1', departmentName: 'IT' },
    employeeCard: {
      employeeCardId: 'card101',
      expirationDate: ISODate("2025-12-31T00:00:00.000Z")
    }
  }
]
```

Wide Column Databases

- Store data in columns rather than stores.
    - Each column family can contain a virtually unlimited number of columns - **колона в колоната**

- Examples: Cassandra and Google's Bigtable

- Use cases:
    - Querying and analysis of large distributed data sets

- **Limitations**
    - Complex transactions
    - Data integrity across documents (joins or complex multi-document transactions)

Graph Databases

Build around the concept of Nodes and Edges

- Nodes represent entities
- Edges represent relationships between nodes

Examples: Neo4j and Amazon Neptune

Use cases:

- Social networks
- Recommendation engines

Limitations:

- System with high update rates
- Performance can degrade with data volume

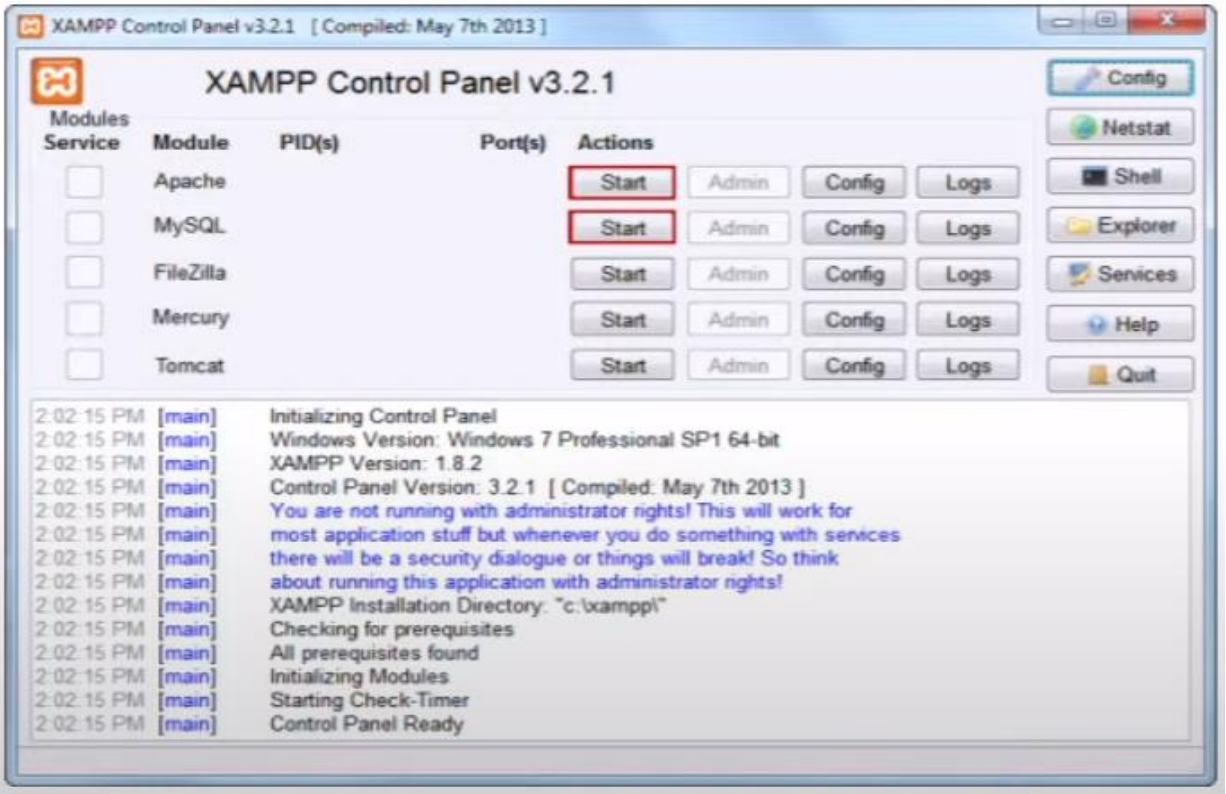
X. Other tools for using

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**