Co-funded by the
Erasmus+ Programme
of the European Union

Project 609854-EPP-1-2019-1-FR-EPPKA2-CBHE-JP - ASEAN FACTORI 4.0:
From Automation and Control Training to the Overall Roll-out of Industry 4.0 across South East Asian Nations

Database Basics and operations with MySQL

University of Health Sciences (UHS)
&
University of Ruse “Angel Kanchev” (UR)

2023

Project 609854-EPP-1-2019-1-FR-EPPKA2-CBHE-JP - ASEAN FACTORI 4.0:
From Automation and Control Training to the Overall Roll-out of Industry 4.0 across South East Asian Nations

BACKGROUND

The e-textbook titled “Database Basics and operations with MySQL” is intended for students registered in the University of Health Sciences (UHS) in Vientiane, Laos.

The purpose of this e-textbook is to present to the students the basic concepts of the modern databases and the most basic characteristics and operations of the Structured Query Language – SQL.

OBJECTIVES

The objectives are as follows:

- To present the concepts of data management and the basic the modern databases
- To introduce the students to the different data types and the data definition concepts
- To present the basic SQL operations and queries
- To introduce the students to MySQL Server and its characteristics
- To show to the students the basic steps for database design and the related rules

PROFILE OF THE INSTRUCTORS

Detailed agenda and instructor’s profile are shown below.

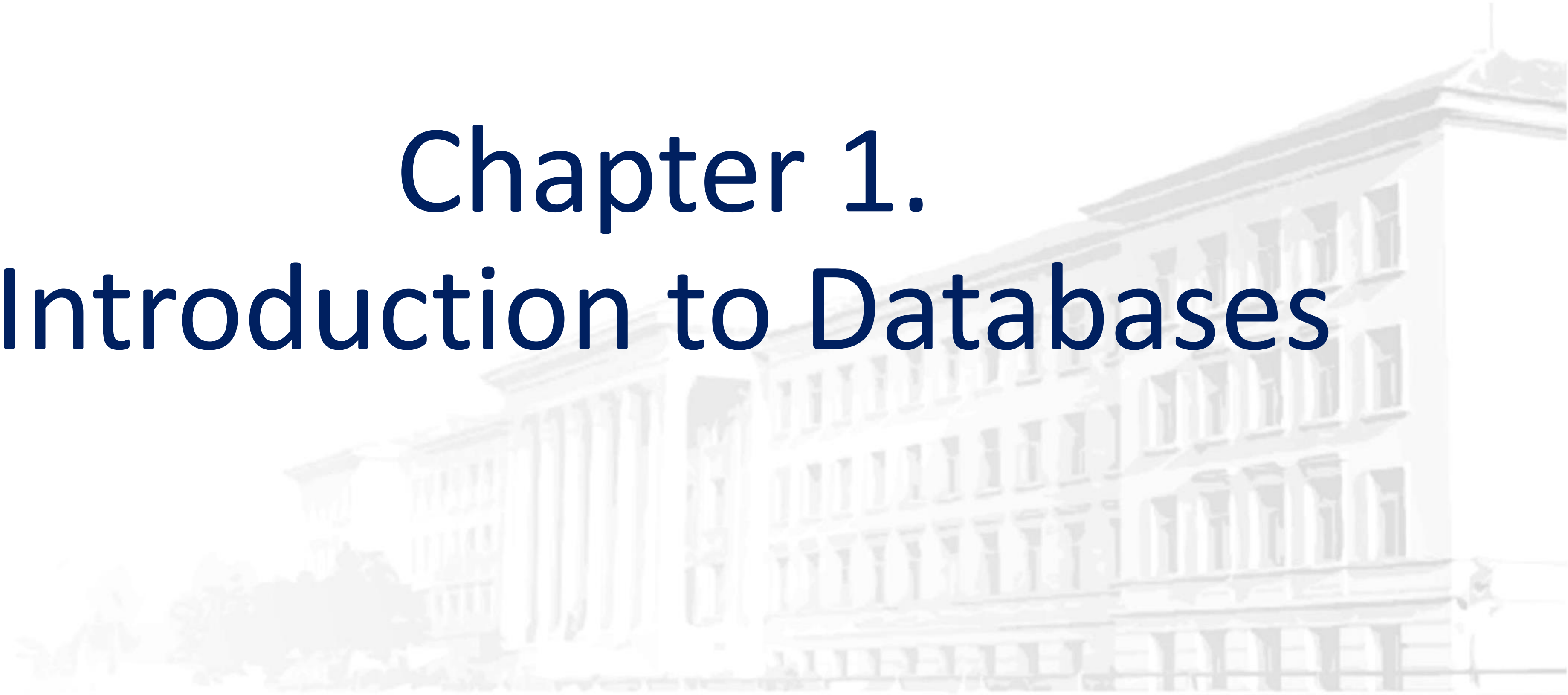
Name	Affiliation
Mr. Lattanavong Thammabavong	University of Health Sciences, Vientiane, Laos
Mr. Seksith Vangkonevilay	University of Health Sciences, Vientiane, Laos
Ms. Noy Lovanhuk	University of Health Sciences, Vientiane, Laos
Assoc. Prof. Dr. Nina Bencheva	University of Ruse “Angle Kanchev”, Ruse, Bulgaria
Prof. Dr. Georgi Hristov	University of Ruse “Angle Kanchev”, Ruse, Bulgaria
Assoc. Prof. Dr. Plamen Zahariev	University of Ruse “Angle Kanchev”, Ruse, Bulgaria

This E-textbook is a part of the ASEAN FACTORI 4.0 project, supported by ERASMUS+ Program of the European Union.

ASEAN FACTORI 4.0: From Automation and Control Training to the Overall Roll-out of Industry 4.0 across South East Asian Nations, reference number: 609854-EPP-1-2019-1-FR-EPPKA2-CBHE-JP.

Disclaimer: This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

TABLE OT CONTENTS

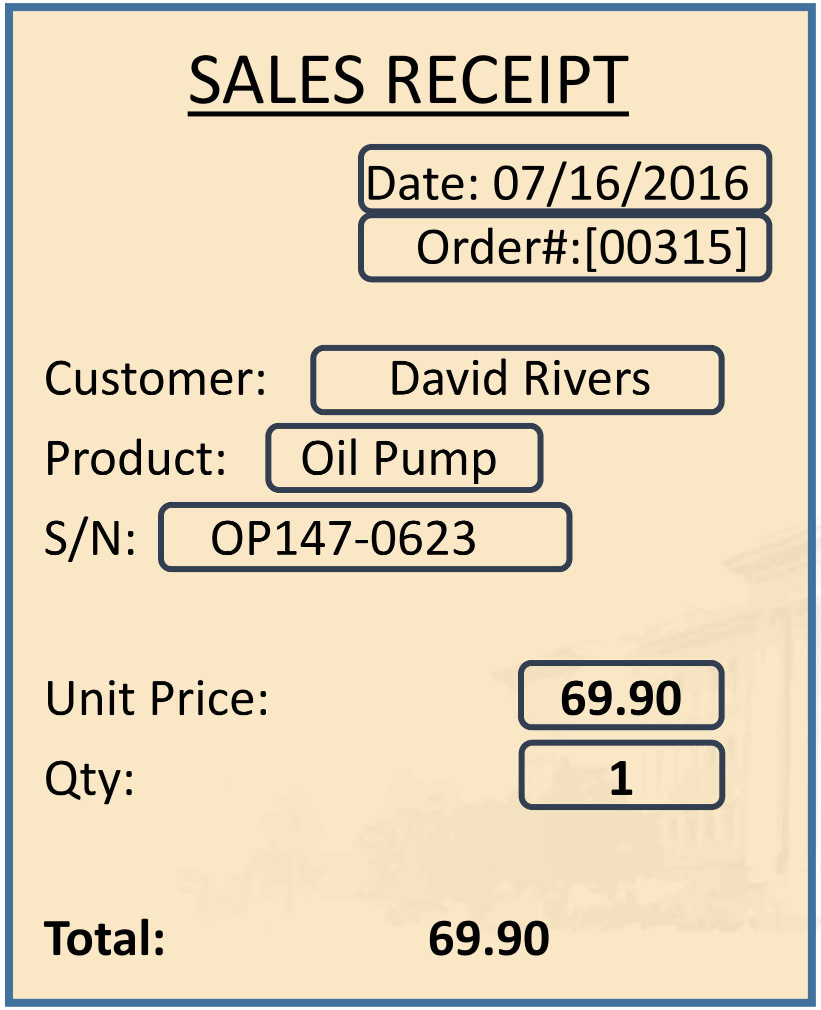


Chapter 1.
Introduction to Databases

Data Management

When Do We Need a Database?

Storage vs. Management



SALES RECEIPT

Date: 07/16/2016
Order#:[00315]

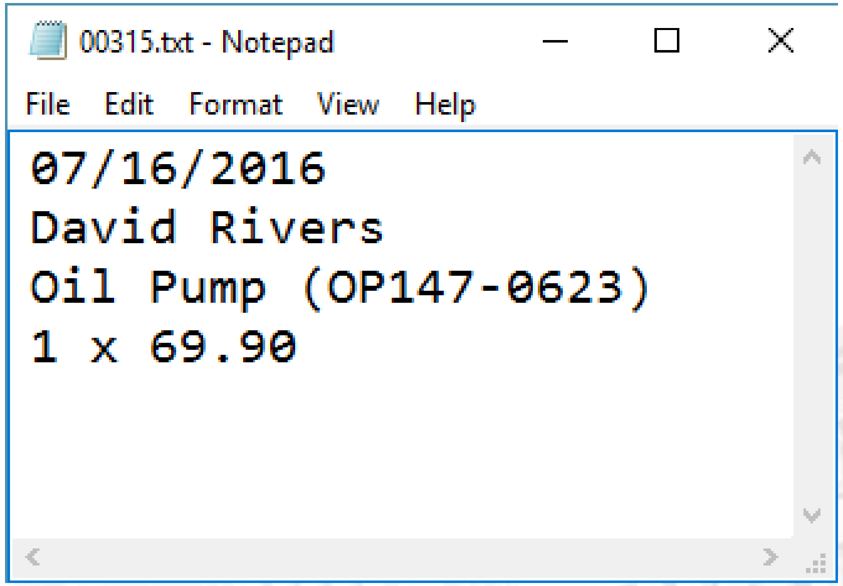
Customer: David Rivers
Product: Oil Pump
S/N: OP147-0623

Unit Price: **69.90**
Qty: **1**

Total: 69.90

00315 – 07/16/2016
David Rivers
Oil Pump (OP147-0623)
1 x 69.90

Storage vs. Management



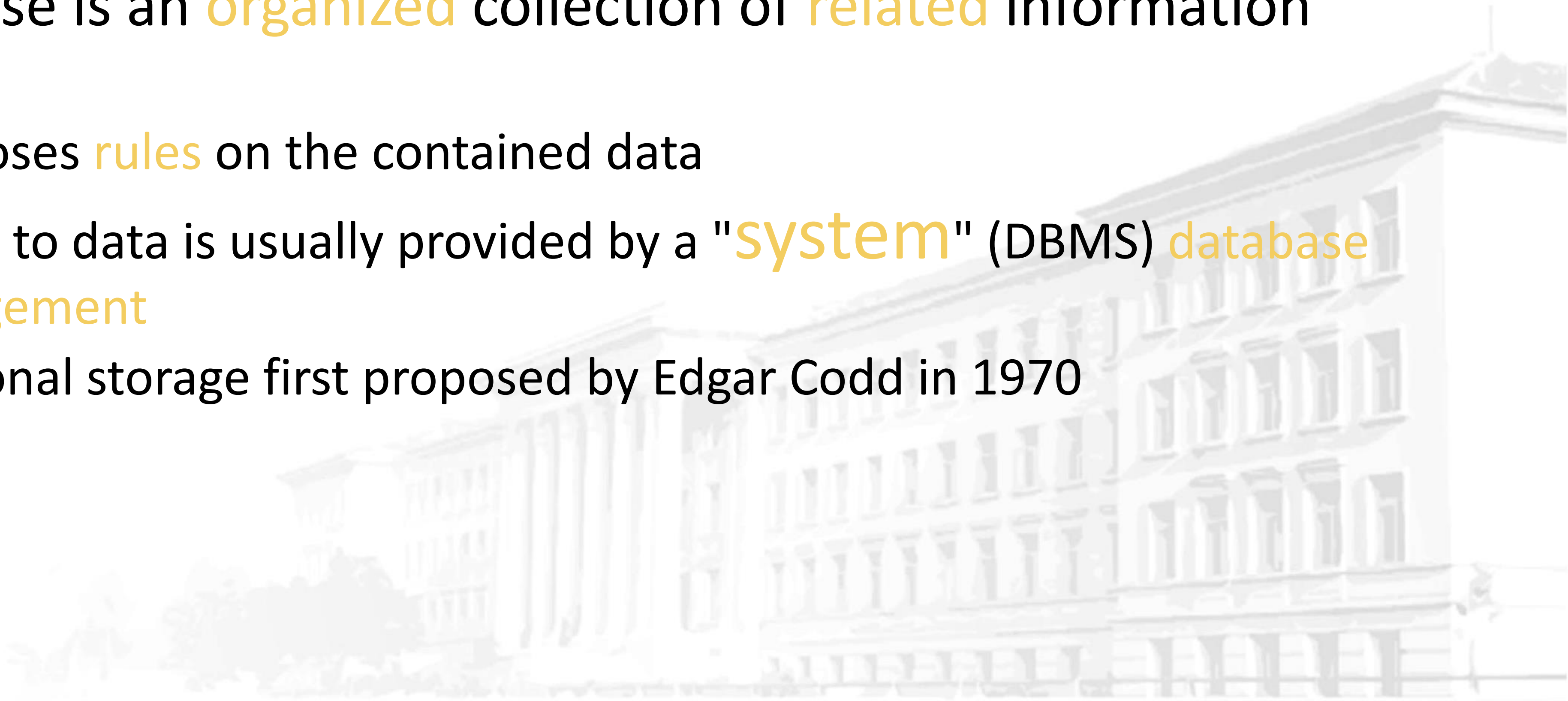
```
07/16/2016
David Rivers
Oil Pump (OP147-0623)
1 x 69.90
```

Order#	Date	Customer	Product	S/N	Qty
00315	07/16/2016	David Rivers	Oil Pump	OP147-063	1

Storage vs. Management

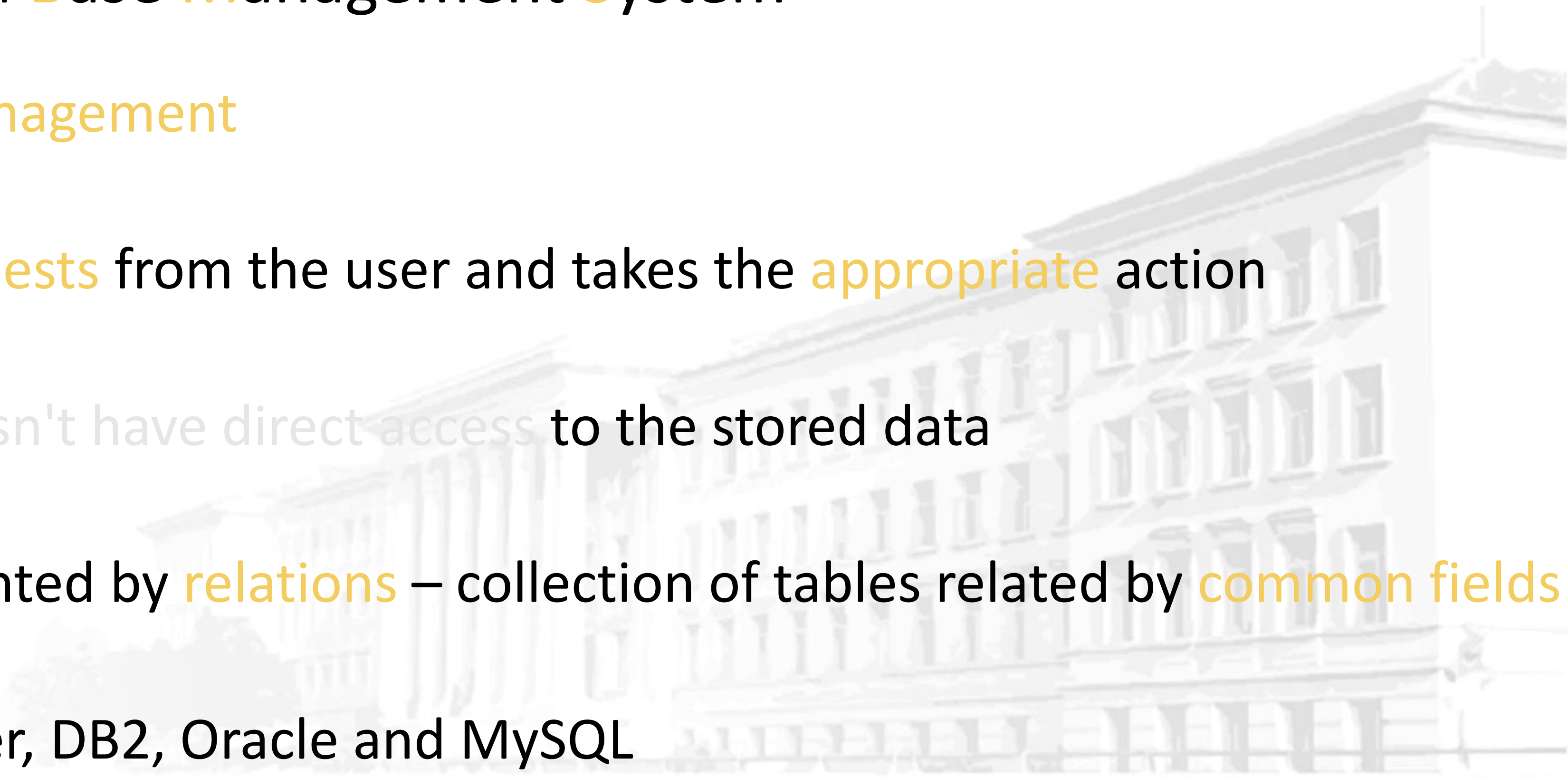
- Storing data is not the primary reason to use a database
- Flat storage eventually runs into issues with
  - Size
  - Ease of updating
  - Accuracy
  - Security
  - Redundancy
  - Importance

Databases



- 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

RDBMS

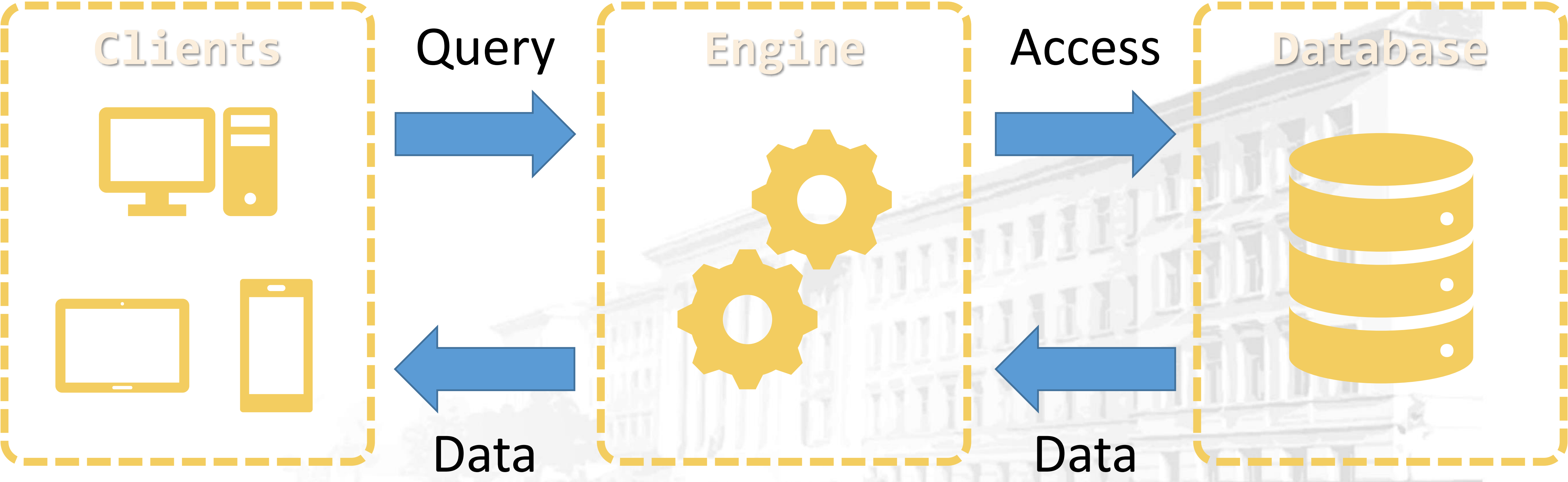


- Relational Data Base Management System
  - Database management
  - It parses requests from the user and takes the appropriate action
  - The user doesn't have direct access to the stored data
  - Data is presented by relations – collection of tables related by common fields
  - MS SQL Server, DB2, Oracle and MySQL

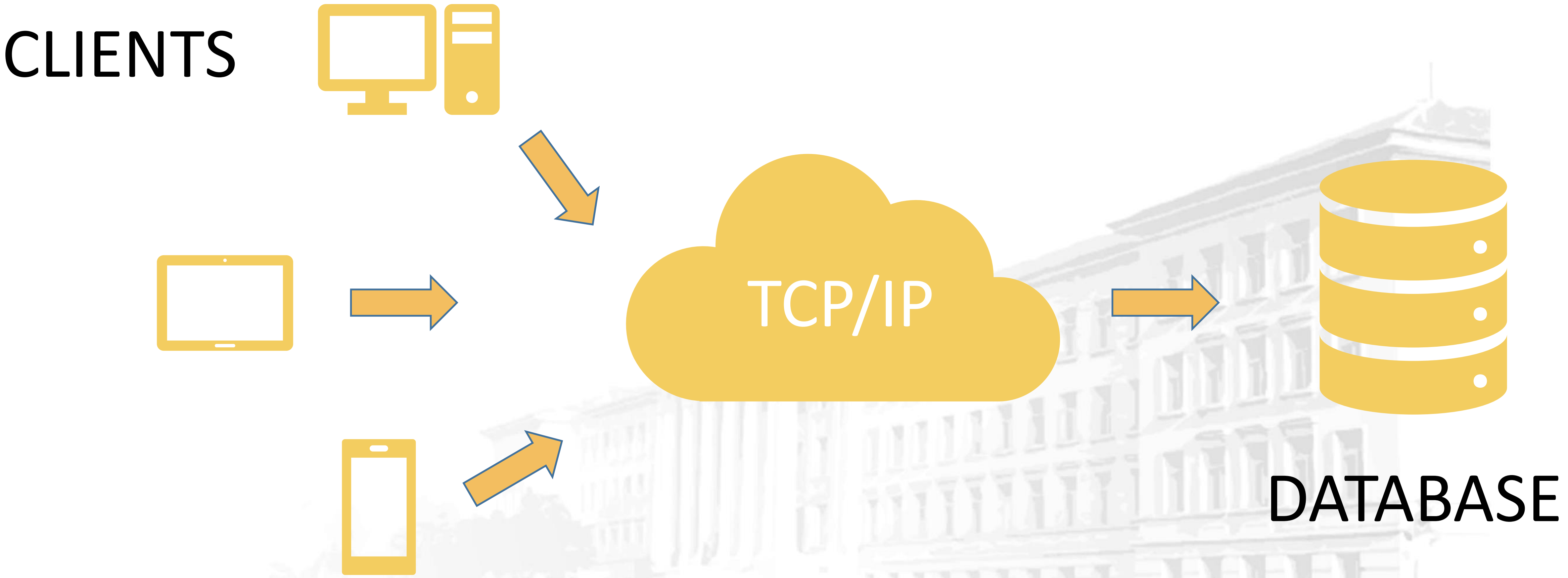
Database Engines

Database Engine Flow

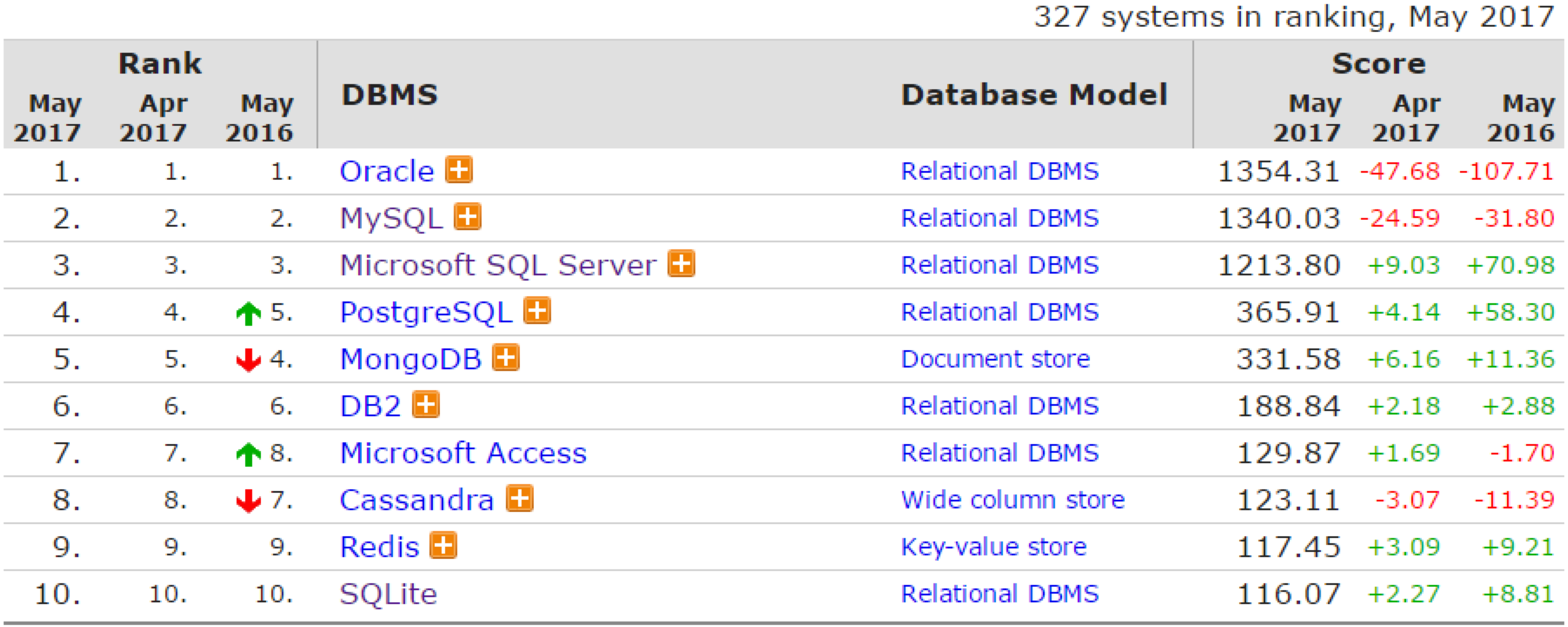
- SQL Server uses the Client-Server Model



Client-Server Model



Top Database Engines



327 systems in ranking, May 2017

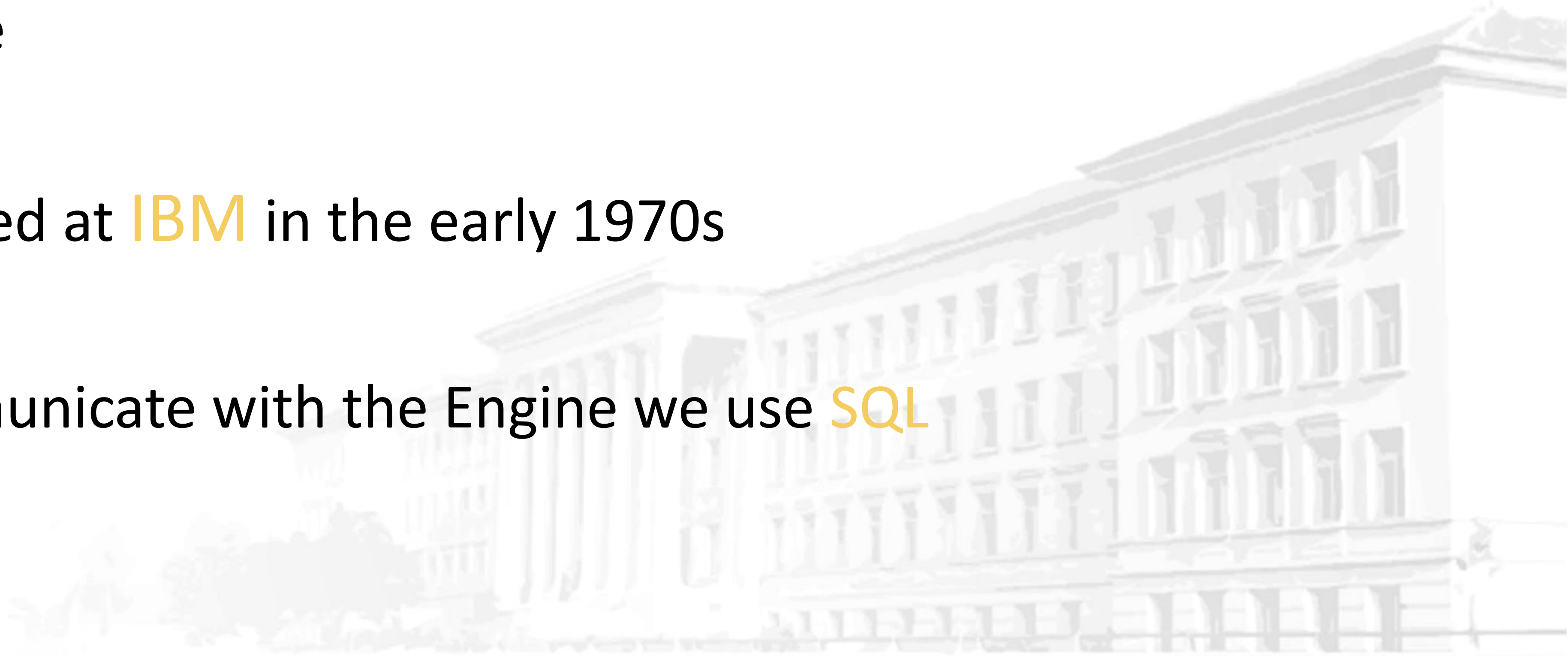
Rank			DBMS	Database Model	Score		
May 2017	Apr 2017	May 2016			May 2017	Apr 2017	May 2016
1.	1.	1.	Oracle	Relational DBMS	1354.31	-47.68	-107.71
2.	2.	2.	MySQL	Relational DBMS	1340.03	-24.59	-31.80
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1213.80	+9.03	+70.98
4.	4.	↑ 5.	PostgreSQL	Relational DBMS	365.91	+4.14	+58.30
5.	5.	↓ 4.	MongoDB	Document store	331.58	+6.16	+11.36
6.	6.	6.	DB2	Relational DBMS	188.84	+2.18	+2.88
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	129.87	+1.69	-1.70
8.	8.	↓ 7.	Cassandra	Wide column store	123.11	-3.07	-11.39
9.	9.	9.	Redis	Key-value store	117.45	+3.09	+9.21
10.	10.	10.	SQLite	Relational DBMS	116.07	+2.27	+8.81

Source: http://db-engines.com/en/ranking

The Structured Query Language

Query Components

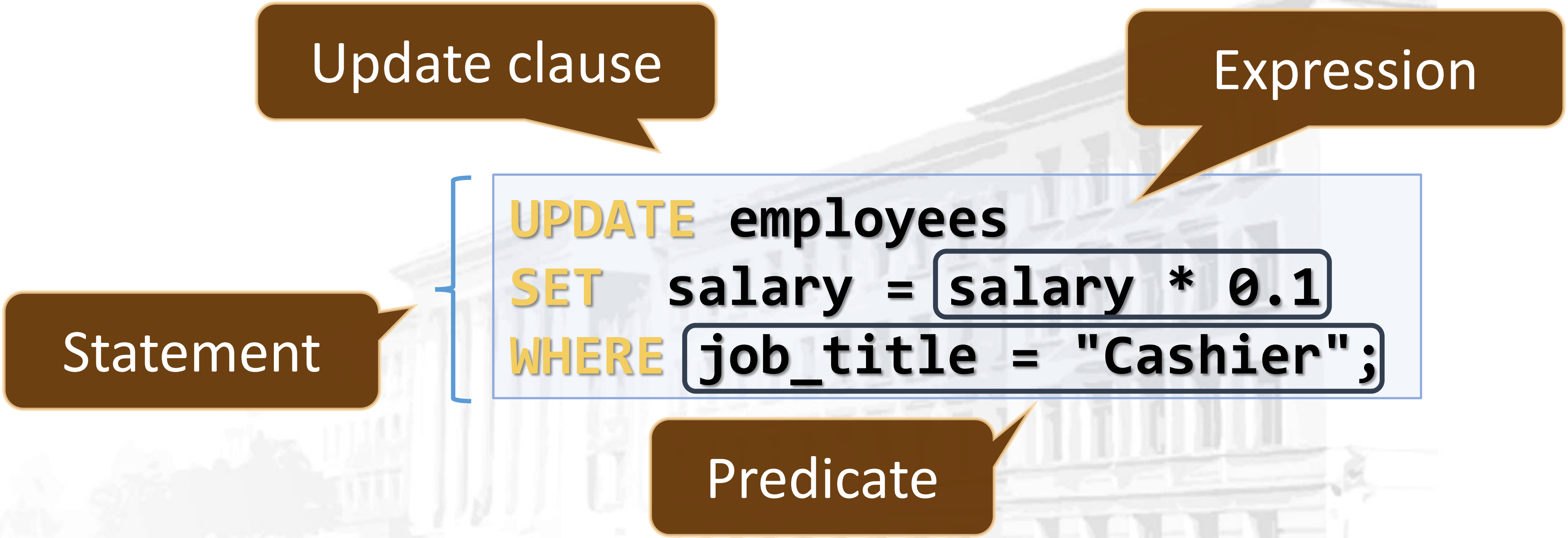
Structured Query Language



- Programming language designed for managing data in a relational database
- Developed at IBM in the early 1970s
- To communicate with the Engine we use SQL

Structured Query Language

- Subdivided into several language elements
  - Queries
  - Clauses
  - Expressions
  - Predicates
  - Statements

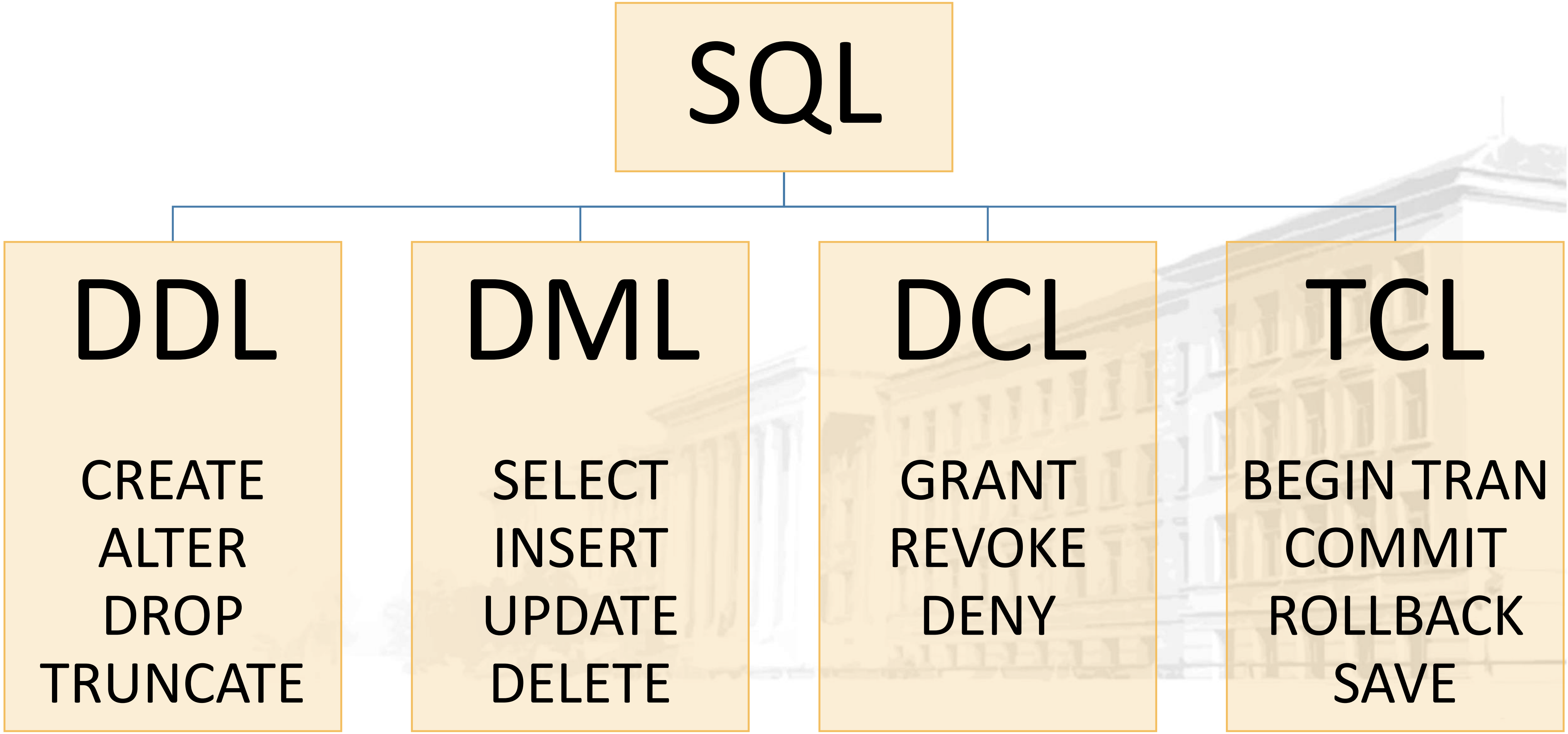


```
UPDATE employees
SET   salary = salary * 0.1
WHERE job_title = "Cashier";
```

Structured Query Language

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

Structured Query Language



SQL

DDL
CREATE
ALTER
DROP
TRUNCATE

DML
SELECT
INSERT
UPDATE
DELETE

DCL
GRANT
REVOKE
DENY

TCL
BEGIN TRAN
COMMIT
ROLLBACK
SAVE

MySQL

Relational DB Management

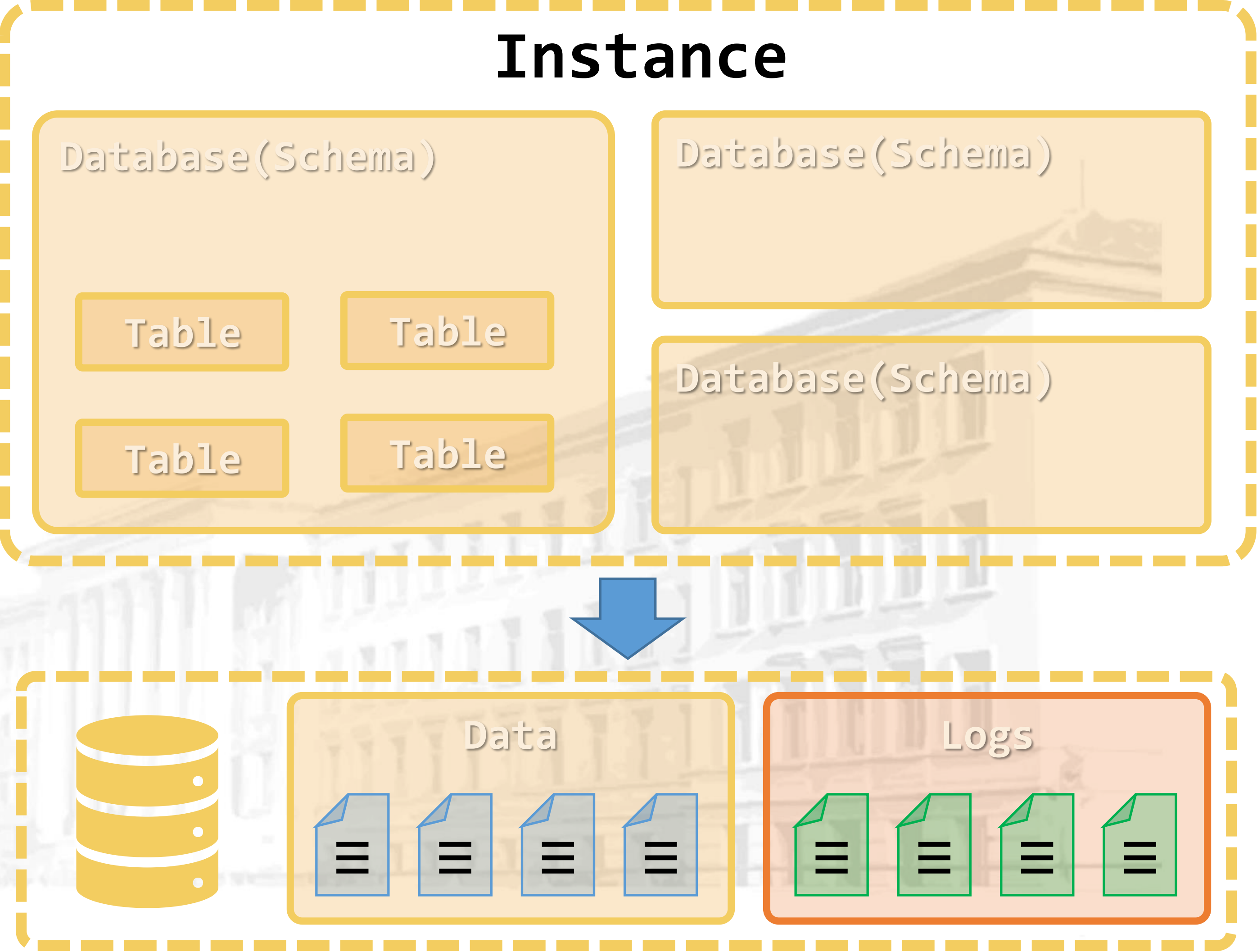
MySQL

- Open-source relational database management system
- Used in many large-scale websites like including Google, Facebook, YouTube etc.
- Works on many system platforms –

  MAC OS, Windows, Linux
- Download MySQL Server
  - Windows: dev.mysql.com/downloads/windows/installer/
  - Ubuntu/Debian: dev.mysql.com/downloads/repo/apt/

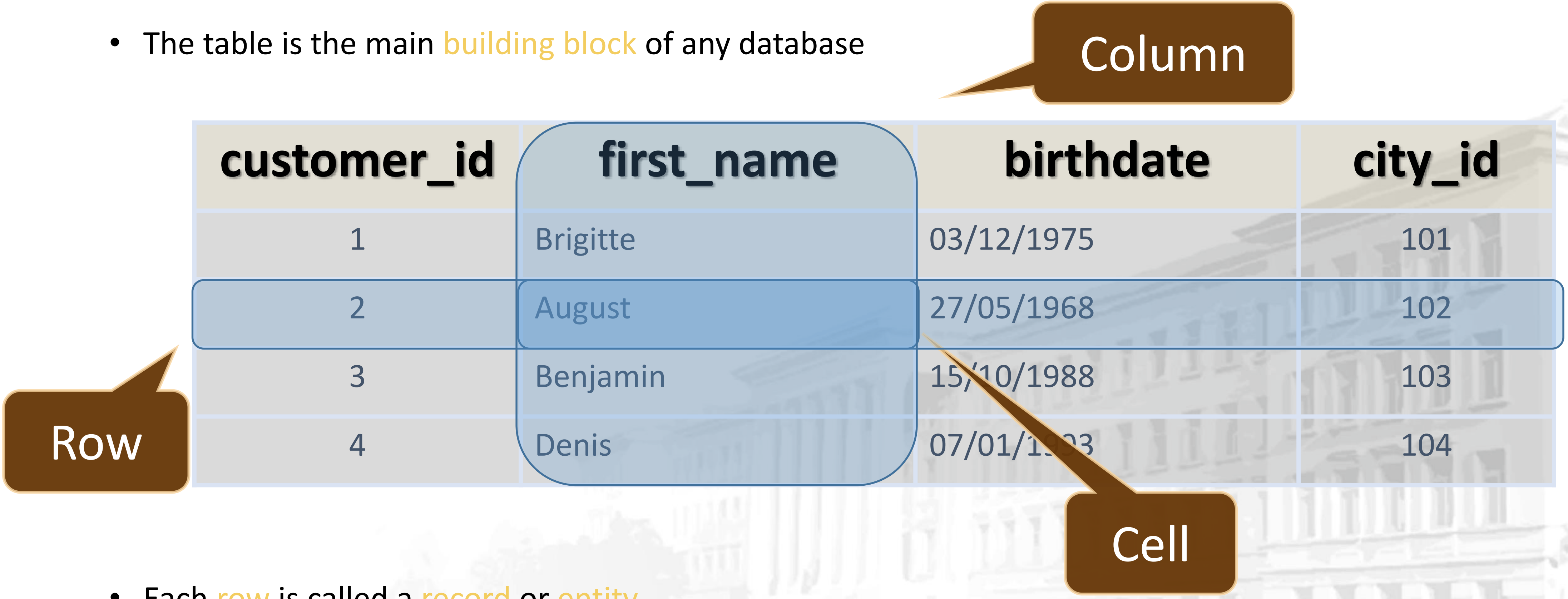
MySQL Server Architecture

- Logical Storage
  - Instance
  - Database/Schema
  - Table
- Physical Storage
  - Data files and Log files
  - Data pages



Database Table Elements

- The table is the main building block of any database



customer_id	first_name	birthdate	city_id
1	Brigitte	03/12/1975	101
2	August	27/05/1968	102
3	Benjamin	15/10/1988	103
4	Denis	07/01/1993	104

- Each row is called a record or entity
- Columns (fields) define the type of data they contain

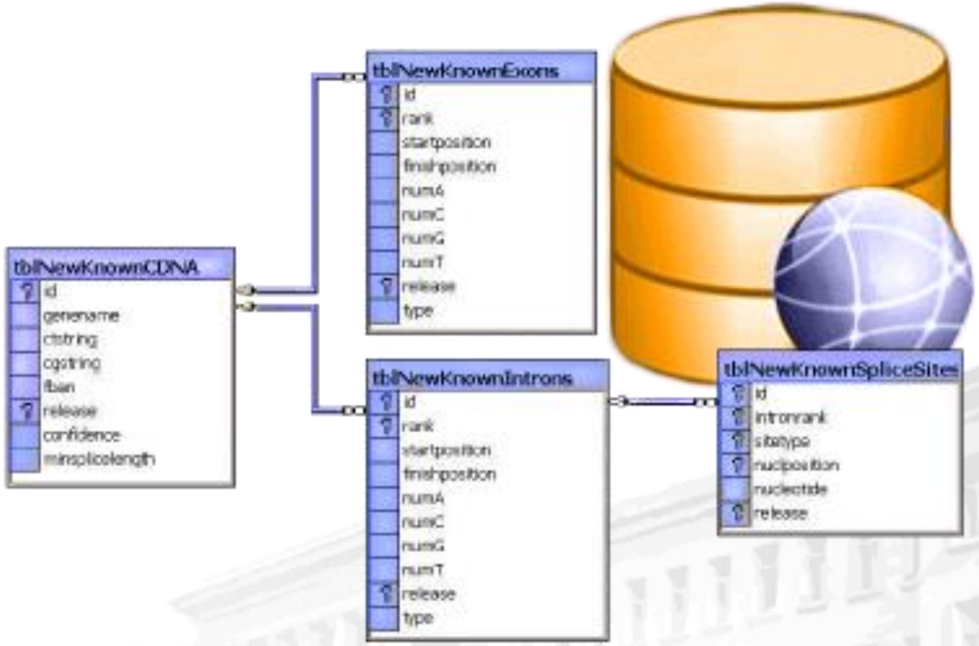


Table Relationships

Splitting data in tables

Why Split Related Data?

Empty records

first	last	registered	email	email2
David	Rivers	05/02/2016	drivers@mail.cx	david@homedomain.cx
Sarah	Thorne	07/17/2016	sarah@mail.cx	*NULL*
Michael	Walters	11/23/2015	walters_michael@mail.cx	*NULL*

Redundant information

order_id	date	customer	product	s/n	price
00315	07/16/2016	David Rivers	Oil Pump	OP147-0623	69.90
00315	07/16/2016	David Rivers	Accessory Belt	AB544-1648	149.99
00316	07/17/2016	Sarah Thorne	Wiper Fluid	WF000-0001	99.90
00317	07/18/2016	Michael Walters	Oil Pump	OP147-0623	69.90

Related Tables

- We split the data and introduce relationships between the tables to avoid repeating information

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

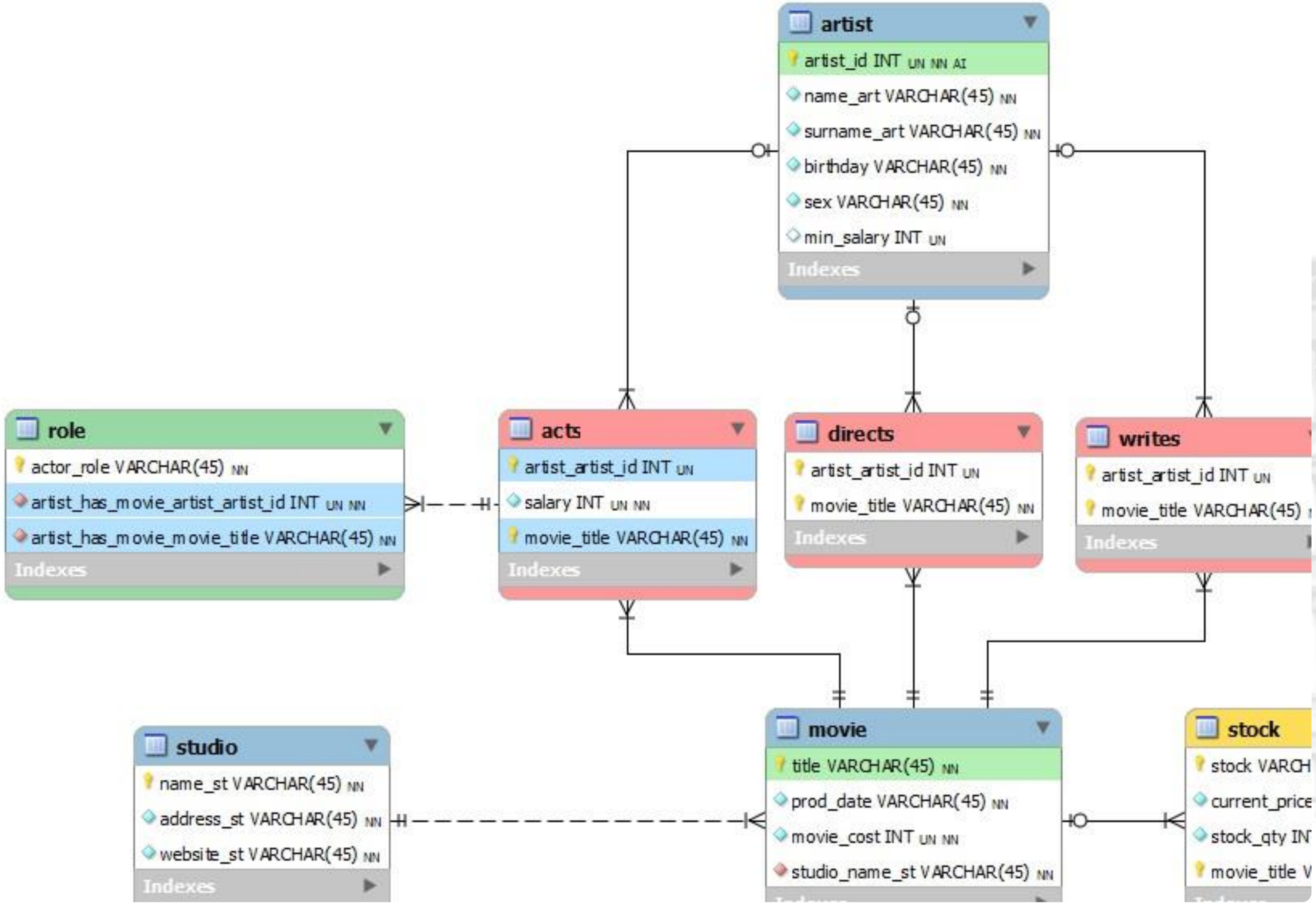
user_id	email
203	drivers@mail.cx
204	sarah@mail.cx
205	walters_michael@mail.cx
203	david@homedomain.cx

Primary Key

Foreign Key

- Connection via Foreign Key in one table pointing to the Primary Key in another

Entity Relationship (E/R) Diagrams



artist
artist_id INT UN NN AI
name_art VARCHAR(45) NN
surname_art VARCHAR(45) NN
birthday VARCHAR(45) NN
sex VARCHAR(45) NN
min_salary INT UN
Indexes

role
actor_role VARCHAR(45) NN
artist_has_movie_artist_artist_id INT UN NN
artist_has_movie_movie_title VARCHAR(45) NN
Indexes

acts
artist_artist_id INT UN
salary INT UN NN
movie_title VARCHAR(45) NN
Indexes

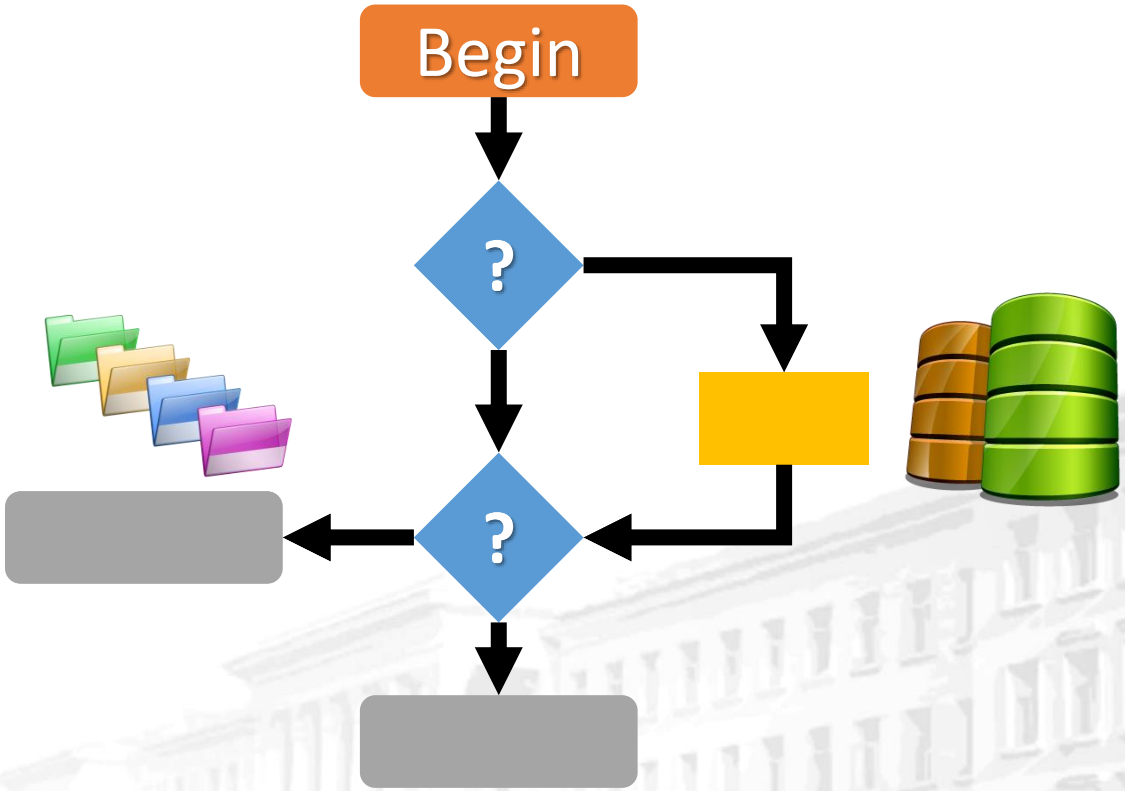
directs
artist_artist_id INT UN
movie_title VARCHAR(45) NN
Indexes

writes
artist_artist_id INT UN
movie_title VARCHAR(45)
Indexes

studio
name_st VARCHAR(45) NN
address_st VARCHAR(45) NN
website_st VARCHAR(45) NN
Indexes

movie
title VARCHAR(45) NN
prod_date VARCHAR(45) NN
movie_cost INT UN NN
studio_name_st VARCHAR(45) NN

stock
stock VARCH
current_price
stock_qty IN
movie_title V

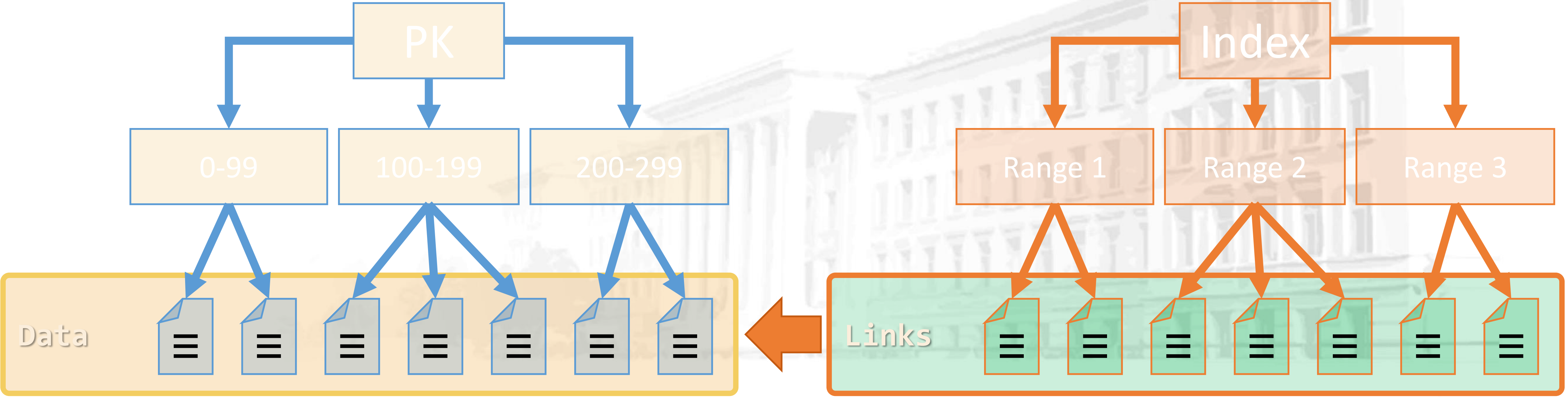


Programmability

Customizing Database Behavior

Indices

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



Views

- Views are prepared queries for displaying sections of our data

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

```
SELECT * FROM v_employee_names
```

- Evaluated at run time – they do not increase performance

Procedures, Functions and Triggers

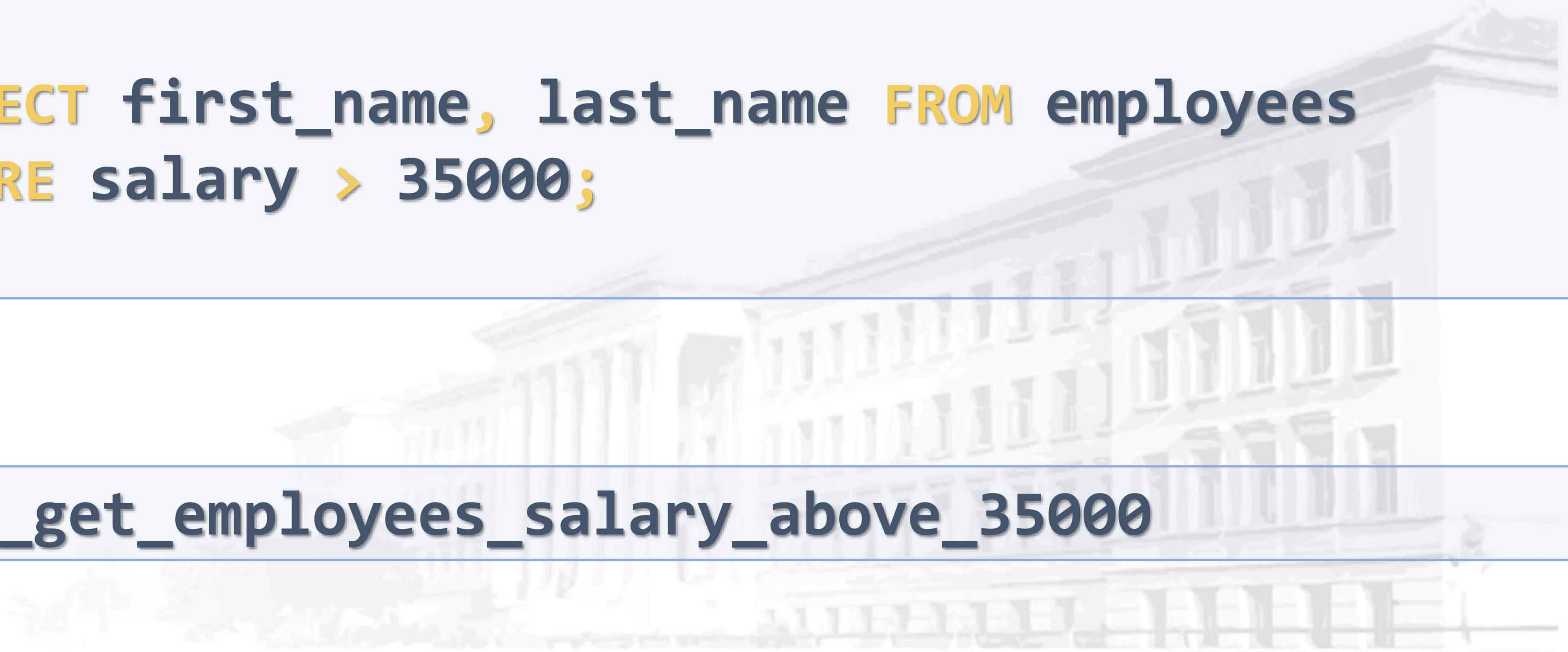
- A database can further be customized with reusable code

- 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

Procedures



```
CREATE PROCEDURE udp_get_employees_salary_above_35000()
BEGIN
    SELECT first_name, last_name FROM employees
    WHERE salary > 35000;
END
```

```
CALL udp_get_employees_salary_above_35000
```

Functions

```
CREATE FUNCTION udf_get_age (dateValue DATE)
RETURNS INT
      BEGIN
      DECLARE result INT;
      SET result = TIMESTAMPDIFF(YEAR, dateValue, NOW());
      RETURN result;
      END
```

```
SELECT udf_get_age('1988-12-21');
```

Summary

- RDBMS stores and manages data

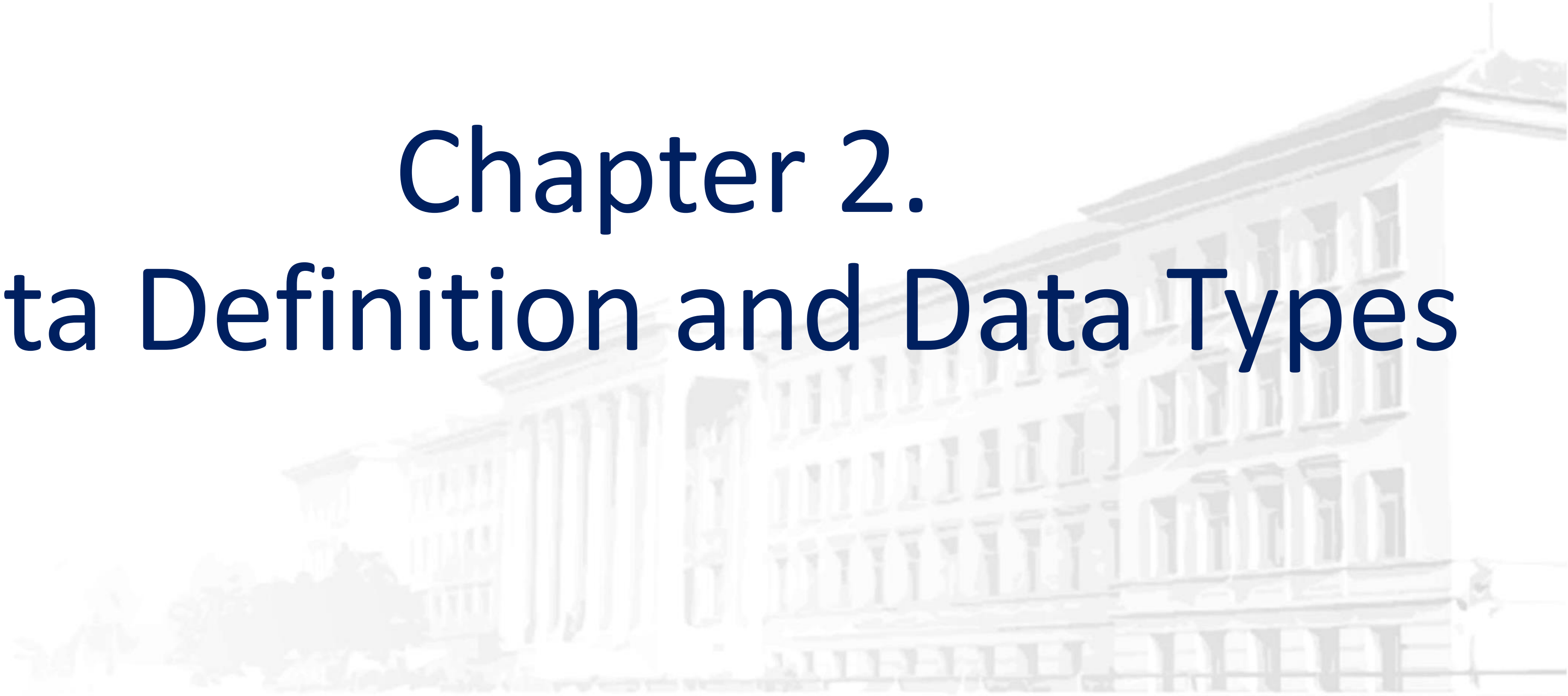
- We communicate with the DB engine via SQL

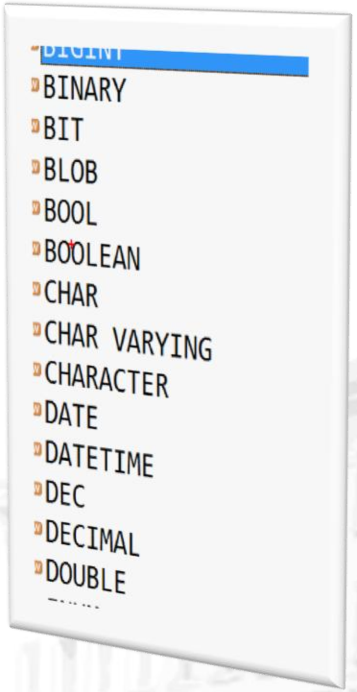
- MySQL is a multiplatform RDBMS using SQL

- Table relations reduce repetition and complexity

- Databases can be customized with functions and procedures

Chapter 2.
Data Definition and Data Types





BIGINT
BINARY
BIT
BLOB
BOOL
BOOLEAN
CHAR
CHAR VARYING
CHARACTER
DATE
DATETIME
DEC
DECIMAL
DOUBLE

Data Types in MySQL Server

Numeric, String and Data Types

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

Numeric Data Types

- **INT** [(*M*)] [UNSIGNED]
  - TINYINT, SMALLINT, MEDIUMINT, BIGINT
- **DOUBLE** [(*M, D*)] [UNSIGNED]

Digits stored for value

Decimals after floating point

  - E.g. DOUBLE[5, 2] – 999.99
- **DECIMAL** [(*M, D*)] [UNSIGNED] [ZEROFILL]

String Types

- String column definitions include attributes that specify the character set or collation
  - CHARACTER SET (Encoding)
    - E.g. utf8, ucs2
  - CHARACTER COLLATION – rules for encoding comparison
    - E.g. latin1_general_cs, Traditional_Spanish_ci_ai etc.

Determines the storage of each character (single or multiple bytes)

Determines the sorting order and case-sensitivity

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

CHARACTER COLLATION - Example

- ORDER BY with different collations

latin1_swedish_ci	latin1_german1_ci	latin1_german2_ci
Muffler	Muffler	Müller
MX Systems	Müller	Muffler
Müller	MX Systems	MX Systems
MySQL	MySQL	MySQL

String Types

- **CHAR** [(M)] - up to 30 characters
- **VARCHAR(M) –** up to 255 characters
- **TEXT** [(M)] – up to 65 535 characters
  - TINYTEXT, MEDIUMTEXT, LONGTEXT
- **BLOB -** Binary Large OBject [(M)] - 65 535 (2^{16} – 1) characters
  - TINYBLOB, MEDIUMBLOB, LONGBLOB

Column name	Column Type
title	VARCHAR(CHAR)
content	TEXT(LONGTEXT)
picture	BLOB(LONGBLOB)

Date Types

- **DATE** - for values with a date part but no time part
- **TIME** - for values with time but no date part
- **DATETIME** - values that contain both date and time parts
- **TIMESTAMP** - both date and time parts

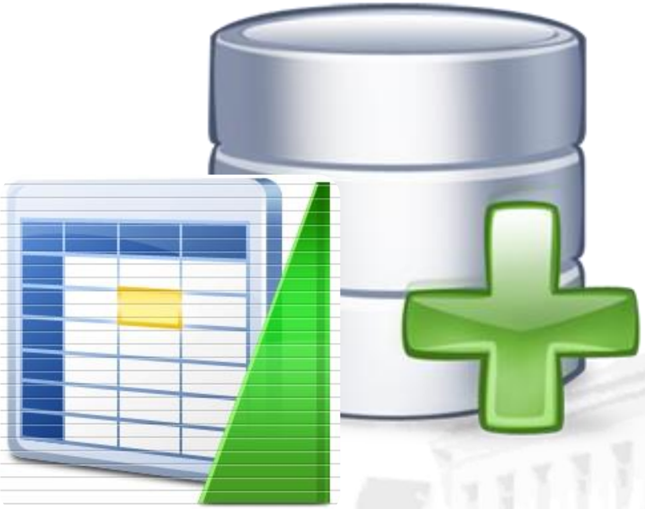
Column name	Column Type
`birthdate`	`DATE`
`last_time_online`	`TIMESTAMP`
`start_at`	`TIME`
`deleted_on`	`DATETIME`

DATETIME and TIMESTAMP have different time ranges

Date Types

- 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'

Data Type	Column Type
DATE	'0000-00-00'
TIME	'00:00:00'
DATETIME	'0000-00-00 00:00:00'
TIMESTAMP	'0000-00-00 00:00:00'
YEAR	0000



Database Modeling

Data Definition using GUI Clients

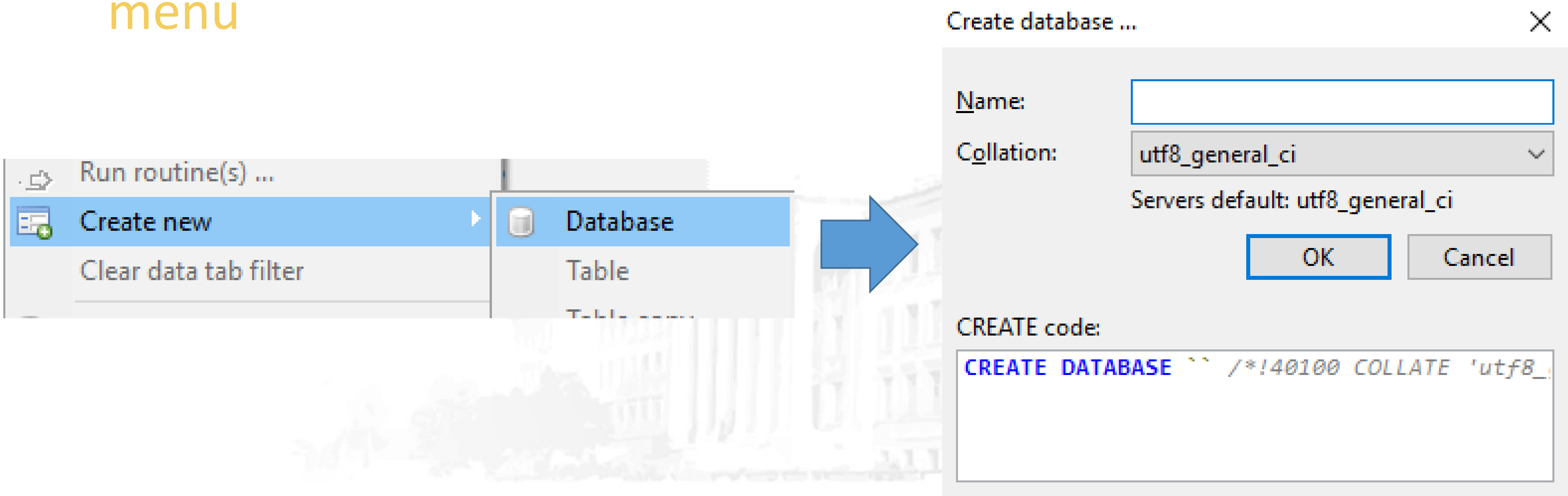
Co-funded by the
Erasmus+ Programme
of the European Union

Working with IDEs

- We will manage databases with HeidiSQL
- 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

Creating a New Database

- Select the instance Create new -> Database from the context menu



Name:

Collation: utf8_general_ci

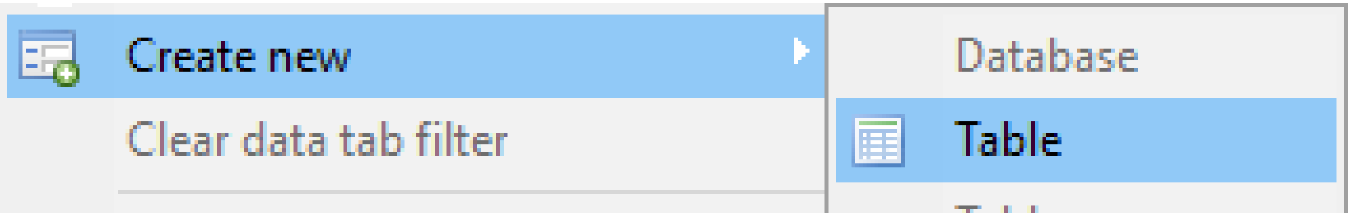
Servers default: utf8_general_ci

CREATE code:

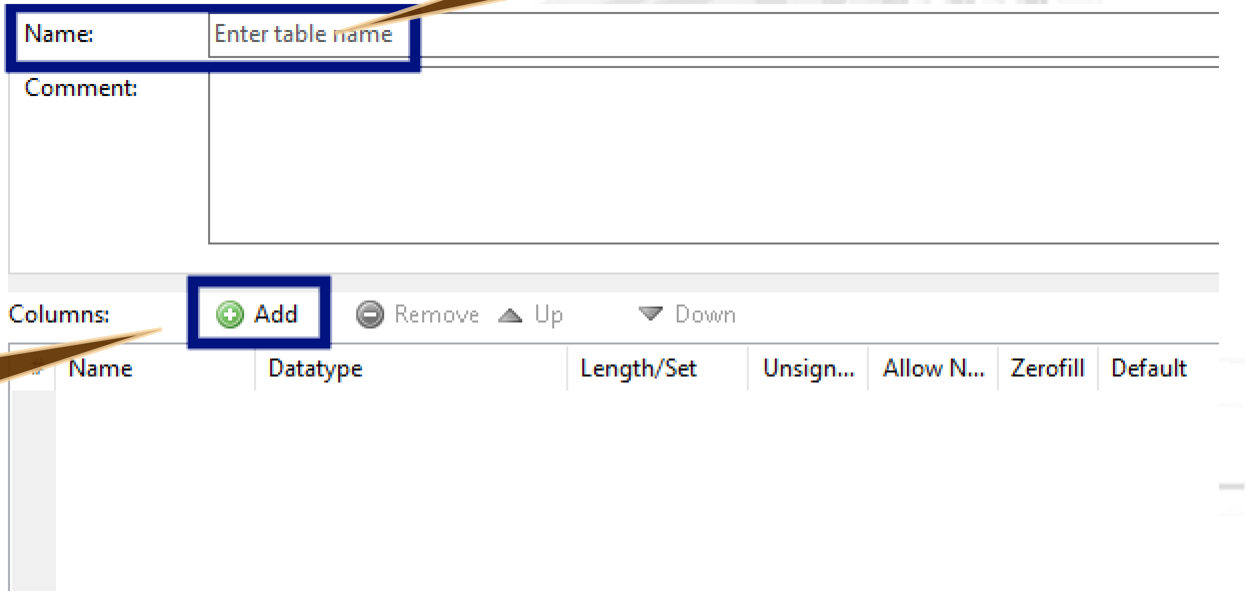
```
CREATE DATABASE `` /*!40100 COLLATE 'utf8_
```

Creating Tables

- Right click on database Select Create new -> Table



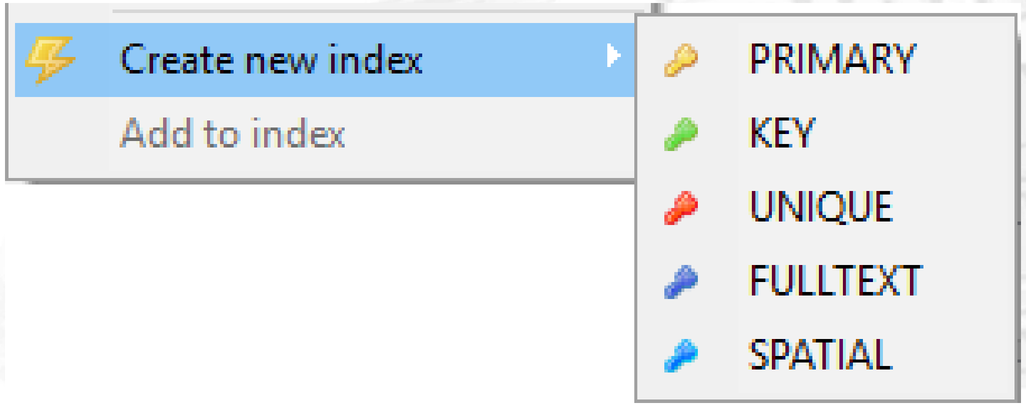
Set up table name



Add new record

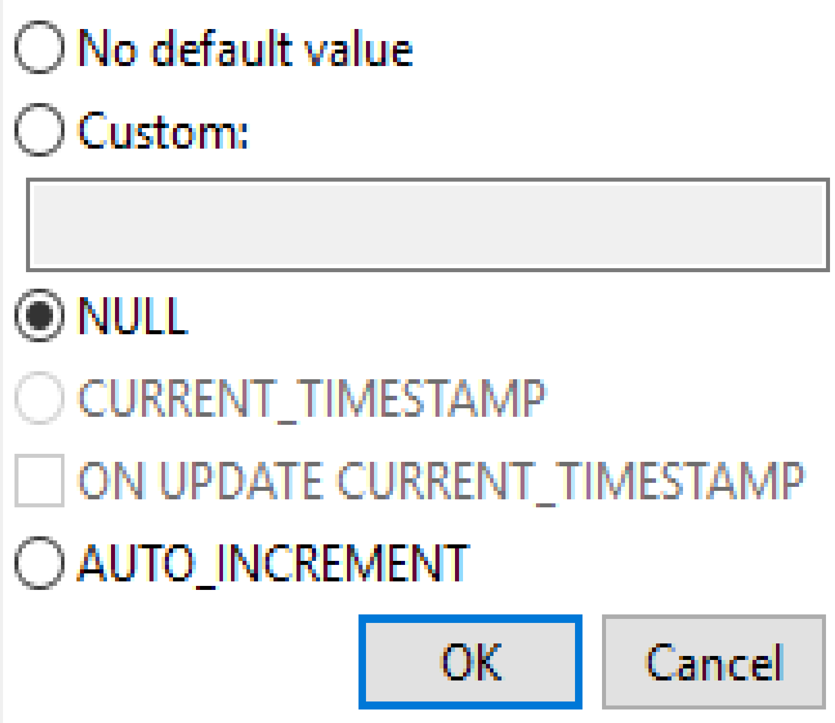
Creating Tables

- A Primary Key is used to uniquely identify and index records
- Click on row Create new index -> Primary from the context menu of the desired row



Creating Tables

- Auto increment – on the "Default" field



○ No default value

○ Custom:

● NULL

○ CURRENT_TIMESTAMP

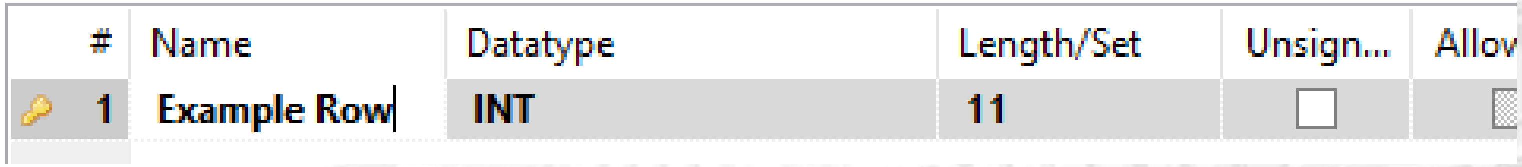
□ ON UPDATE CURRENT_TIMESTAMP

○ AUTO_INCREMENT

Storing and Retrieving Data

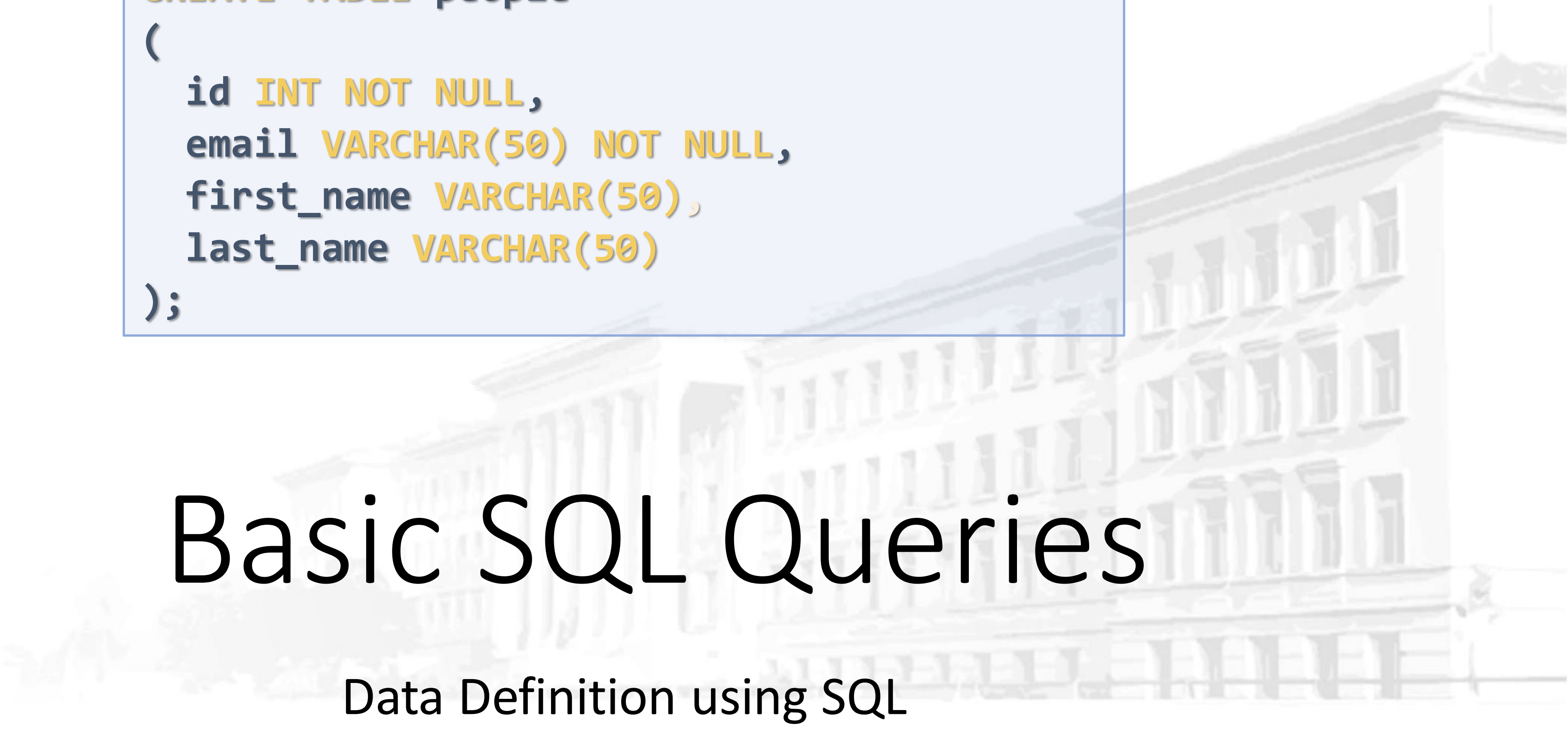
- We can add, modify and read records with GUI Clients

- To insert or edit a record, click inside the cell



	#	Name	Datatype	Length/Set	Unsign...	Allov
	1	Example Row	INT	11	□	□

```
CREATE TABLE people
(
  id INT NOT NULL,
  email VARCHAR(50) NOT NULL,
  first_name VARCHAR(50),
  last_name VARCHAR(50)
);
```



Basic SQL Queries

Data Definition using SQL

SQL Queries

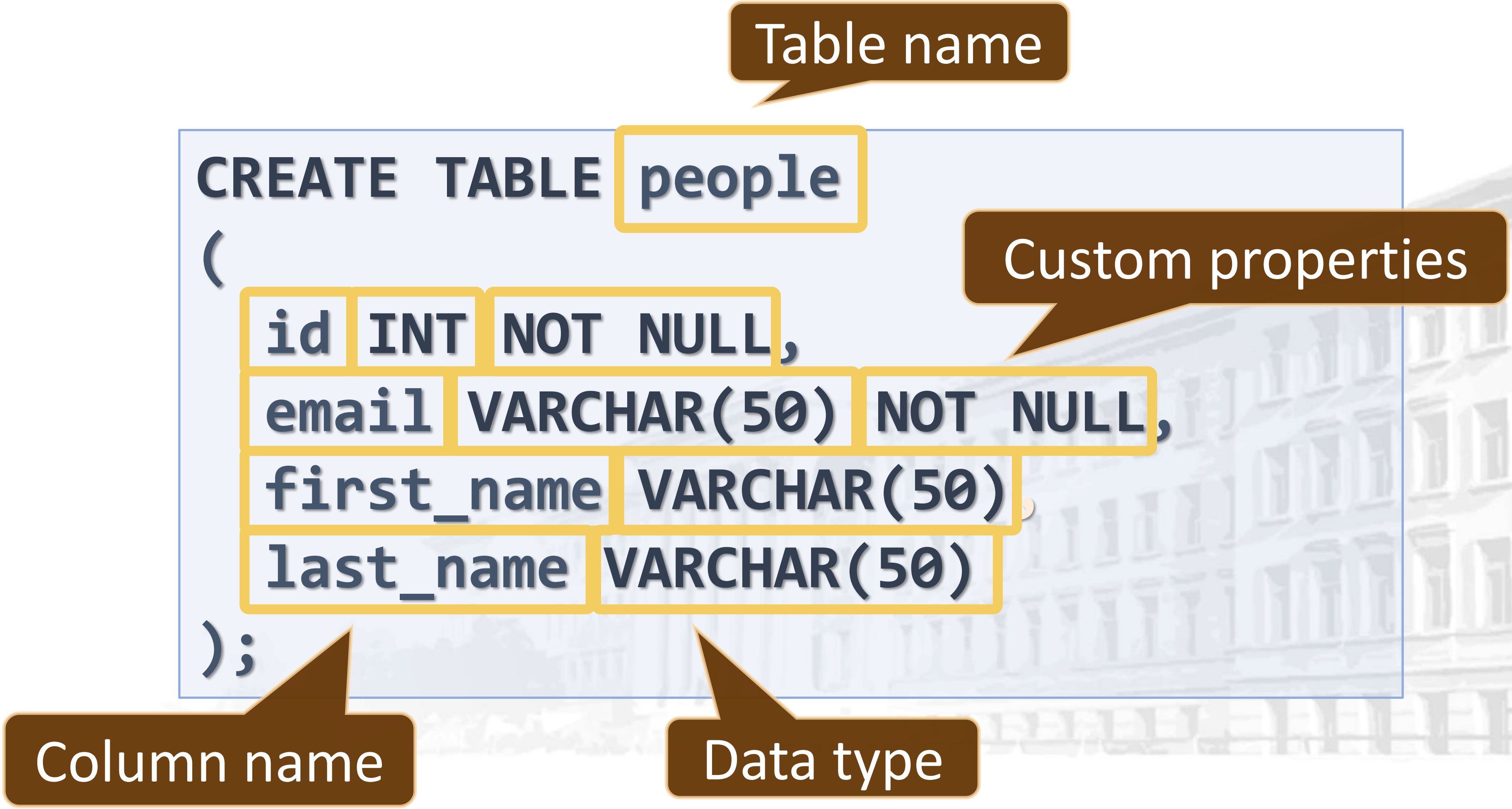
- We communicate with the database engine using SQL
- Queries provide greater control and flexibility
- To create a database using SQL:

Database name

```
CREATE DATABASE employees;
```

- SQL keywords are conventionally capitalized

Table Creation in SQL



```
CREATE TABLE people
(
  id INT NOT NULL,
  email VARCHAR(50) NOT NULL,
  first_name VARCHAR(50),
  last_name VARCHAR(50)
);
```

Table name

Custom properties

Column name

Data type

Retrieve Records in SQL

- Get all information from a table

Table name

```
SELECT * FROM employees;
```

- You can limit the columns and number of records

```
SELECT first_name, last_name FROM employees
LIMIT 5;
```

List of columns

Number of records

Table Customization

Adding Rules, Constraints and Relationships

Custom Column Properties

- Primary Key

```
id INT NOT NULL PRIMARY KEY
```

- Auto-Increment (Identity)

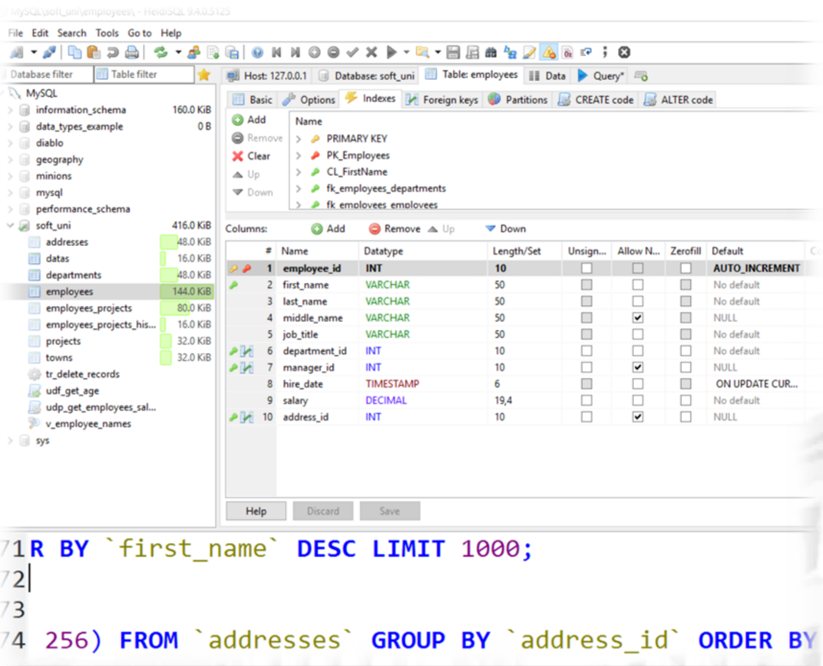
```
id INT AUTO_INCREMENT PRIMARY KEY
```

- 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
```



Name
- PRIMARY KEY
- PK_Employees
- CL_FirstName
- fk_employees_departments
- fk_employees_employees

#	Name	Datatype	Length/Set	Unsign...	Allow N...	Zerofill	Default
1	employee_id	INT	10	□	□	□	AUTO_INCREMENT
2	first_name	VARCHAR	50	□	□	□	No default
3	last_name	VARCHAR	50	□	□	□	No default
4	middle_name	VARCHAR	50	□	☑	□	NULL
5	job_title	VARCHAR	50	□	□	□	No default
6	department_id	INT	10	□	□	□	No default
7	manager_id	INT	10	□	☑	□	NULL
8	hire_date	TIMESTAMP	6	□	□	□	ON UPDATE CUR...
9	salary	DECIMAL	19,4	□	□	□	No default
10	address_id	INT	10	□	☑	□	NULL

```
R BY `first_name` DESC LIMIT 1000;


 256) FROM `addresses` GROUP BY `address_id` ORDER BY
```

Altering Tables

Changing Table Properties After Creation

Altering Tables Using SQL

- A table can be changed using the keywords **ALTER TABLE**

```
ALTER TABLE employees;
```

Table name

- Add new column

```
ALTER TABLE employees
ADD salary DECIMAL;
```

Column name

Data type

Altering Tables Using SQL

- Delete existing column

```
ALTER TABLE people
DROP COLUMN full_name;
```

Column name

- Modify data type of existing column

```
ALTER TABLE people
MODIFY COLUMN email VARCHAR(100);
```

Column name

New data type

Altering Tables Using SQL

- Add primary key to existing column

```
ALTER TABLE people
ADD CONSTRAINT pk_id
PRIMARY KEY (id);
```

Constraint name

Column name
(more than one for composite key)

- Add unique constraint

```
ALTER TABLE people
ADD CONSTRAINT uq_email
UNIQUE (email)
```

Constraint name

Columns name(s)

Altering Tables Using SQL

- Set default value

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

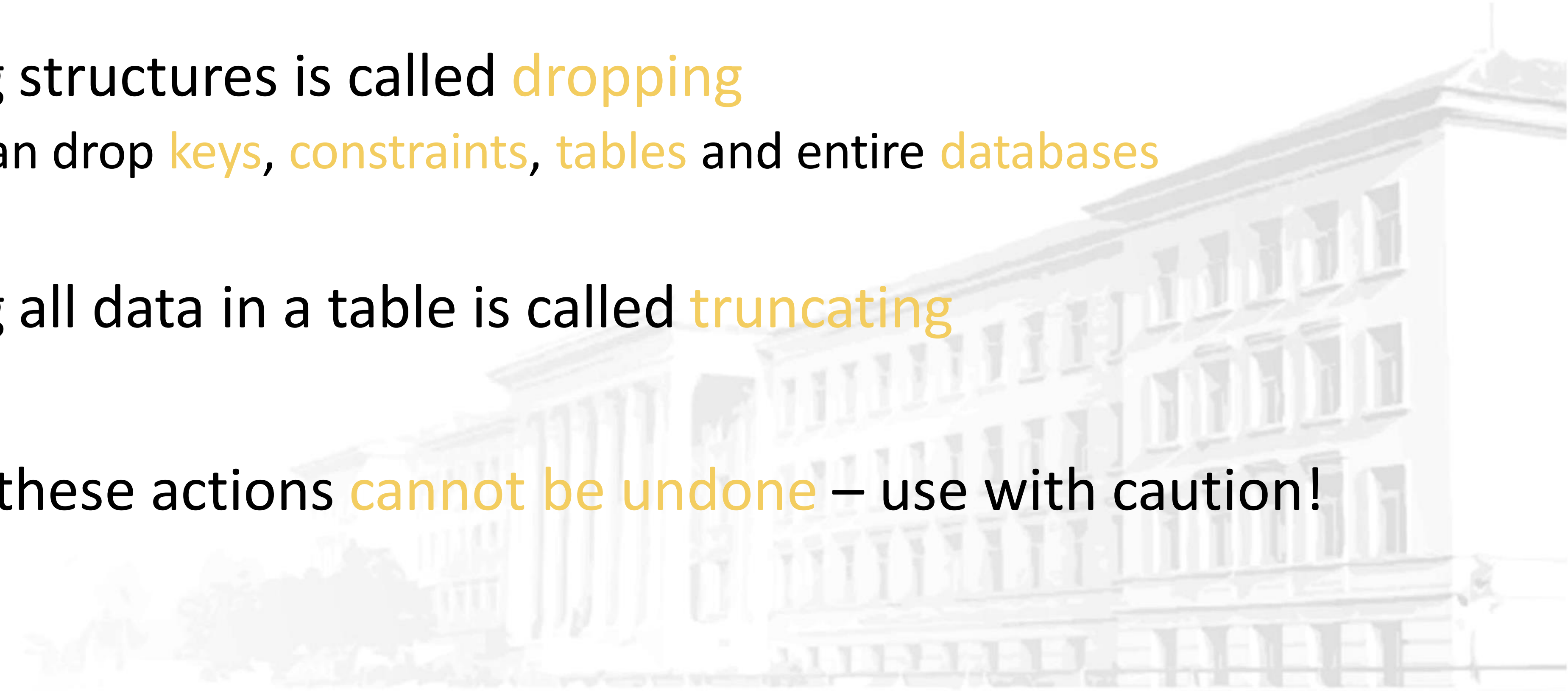
Default value

Column name

Deleting Data and Structures

Dropping and Truncating

Deleting from Database



- 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!

Dropping and Truncating

- To delete all the entries in a table

```
TRUNCATE TABLE employees;
```

Table name

- To drop a table – delete data and structure

```
DROP TABLE employees;
```

Table name

- To drop entire database

Database name

```
DROP DATABASE uni_ruse;
```

Dropping and Truncating

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

```
ALTER TABLE employess
DROP CONSTRAINT pk_id;
```

Table name

Constraint name

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

```
ALTER TABLE employess
ALTER COLUMN clients
DROP DEFAULT;
```

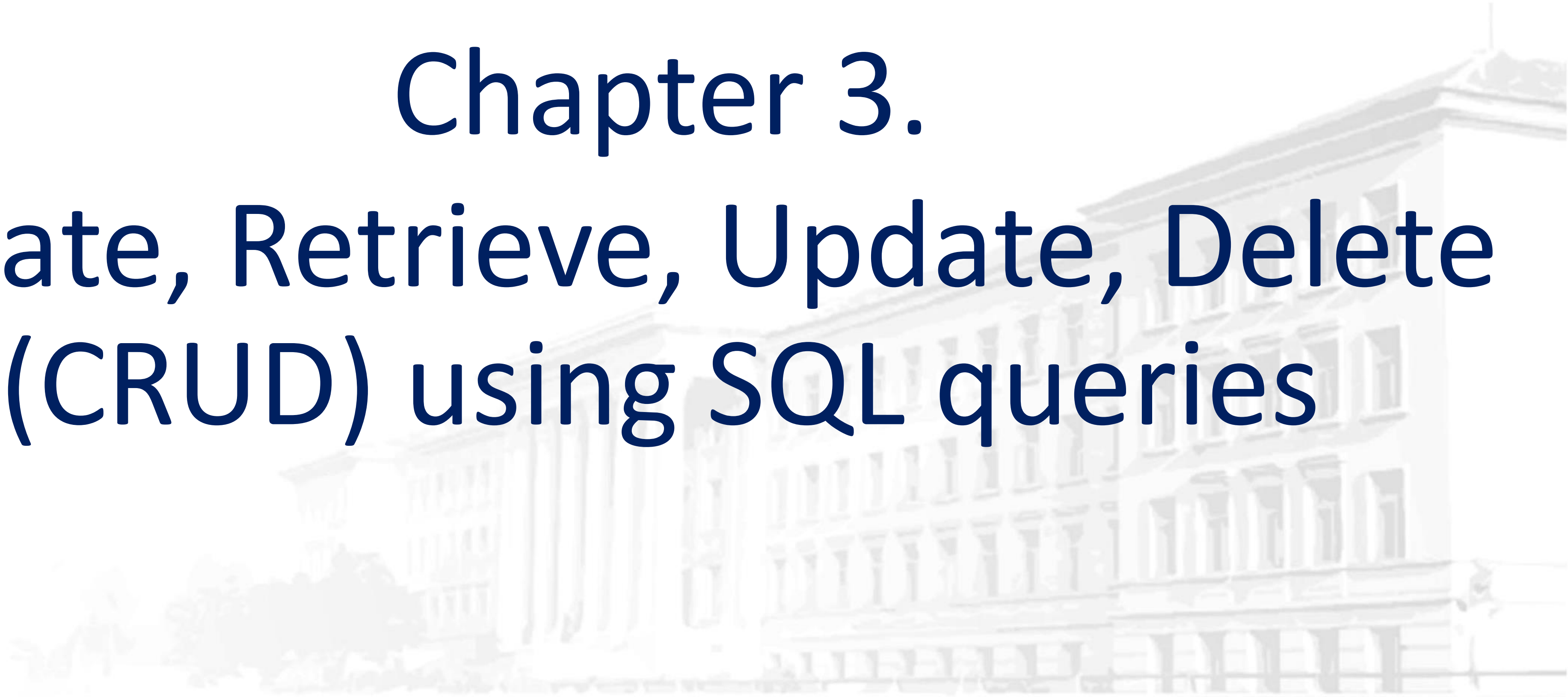
Table name

Columns name

Summary

- Table columns have a fixed type
- We can use GUI Clients to create and customize tables
- SQL provides greater control

```
CREATE TABLE people
(
  id INT NOT NULL,
  email VARCHAR(50) NOT NULL,
  first_name VARCHAR(50),
  last_name VARCHAR(50)
);
```



Chapter 3.
Create, Retrieve, Update, Delete (CRUD) using SQL queries

Query Basics

SQL Introduction

SQL Queries – Few Examples

- 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:

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

SQL Queries – Few Examples

- Update end date of specific projects:

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

- Delete specific projects:

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

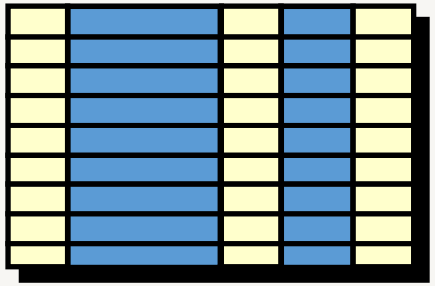
Retrieving Data

Using SQL SELECT

Capabilities of SQL SELECT

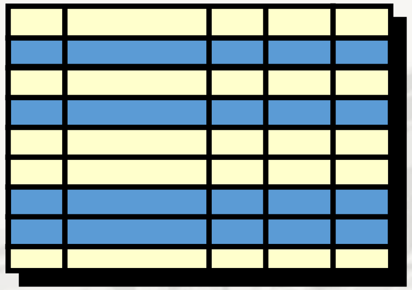
Projection

Take a subset of the columns



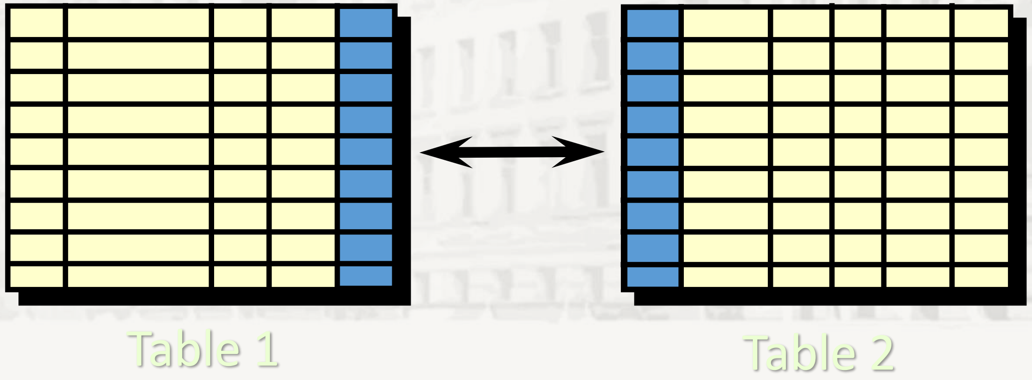
Selection

Take a subset of the rows



Join

Combine tables by some column



SELECT – Examples

- Selecting all columns from the "departments" table

```sql
SELECT * FROM departments;
```

List of columns
(* for all)

Table name

department_id	name	manager_id
1	Engineering	12
2	Tool design	4
3	Sales	273
…	…	…

- Selecting specific columns

```sql
SELECT department_id, name
  FROM departments
```

department_id	name
1	Engineering
2	Tool design
3	Sales
…	…

Column Aliases

- Aliases rename a table or a column heading

```
SELECT employee_id AS id, first_name, last_name
  FROM employees;
```

Display name

id	first_name	last_name
1	Guy	Gilbert
2	Kevin	Brown
…	…	…

- You can shorten fields or clarify abbreviations

```
SELECT c.duration,
       c.acg AS 'Access Control Gateway'
  FROM calls AS c;
```

Concatenation

- You can concatenate column names or strings using the **concat()** function
  - 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`;
```

Problem: Employee Summary

- Find information about all employees, listing their:
  - Full Name
  - Job title
  - Salary

- Use concatenation to display first and last names as one field

- Note: Query Hospital database

Employee Summary - Solution

Concatenation

```
SELECT concat(`first_name`,' ',`last_name`) AS
    'full_name',
    `job_title` as 'job_title',
      `salary` AS `salary`
  FROM `employees` WHERE salary >= 1000;
```

Column alias

Filtering the Selected Rows

- Use **DISTINCT** to eliminate duplicate results

```
SELECT DISTINCT `department_id`
  FROM `employees`;
```

- 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 greater control

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

Other Comparison Conditions

- Conditions ca 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;
```

- 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);
```

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

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

This is always false!

```
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

- 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`;
```

ASC is the default sorting order

LastName	HireDate
Gilbert	1998-07-31
Brown	1999-02-26
Tamburello	1999-12-12
…	…

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

LastName	HireDate
Valdez	2005-07-01
Tsoflias	2005-07-01
Abbas	2005-04-15
…	…

Views

- 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

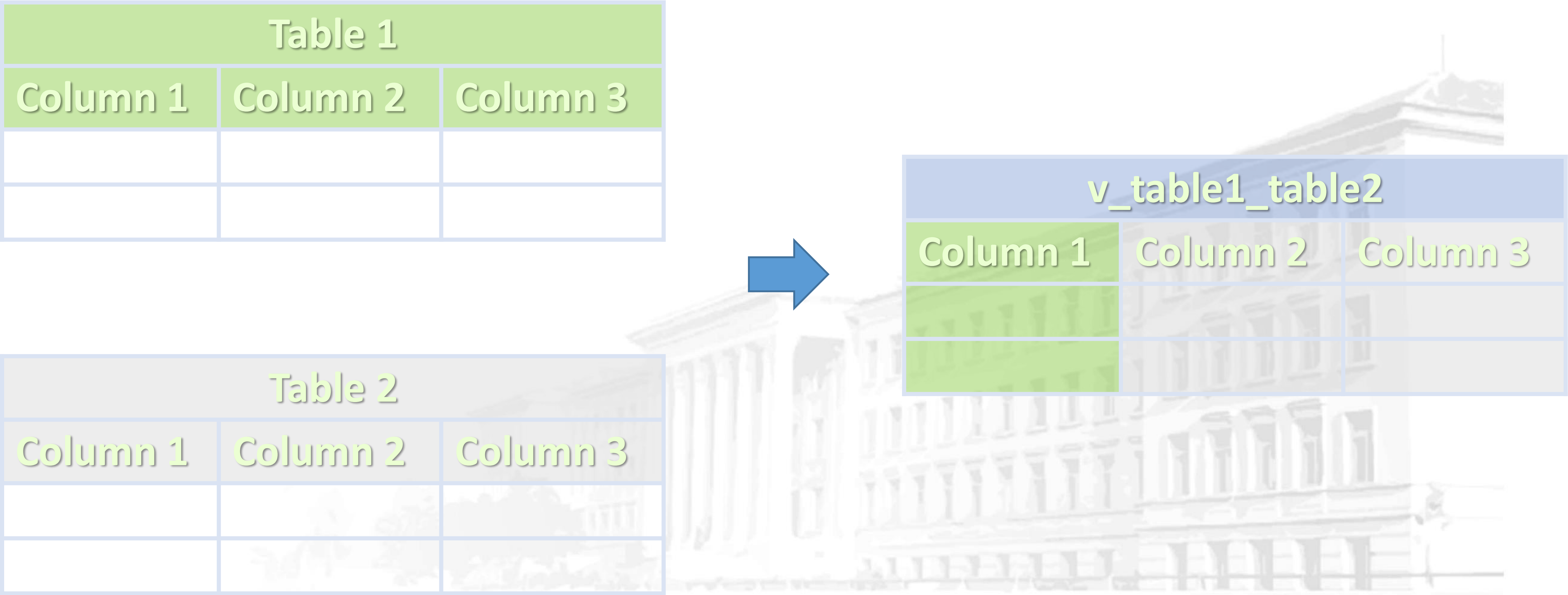


Table 1		
Column 1	Column 2	Column 3

Table 2		
Column 1	Column 2	Column 3

v_table1_table2		
Column 1	Column 2	Column 3

Views - Example

- Get employee names and salaries, by department

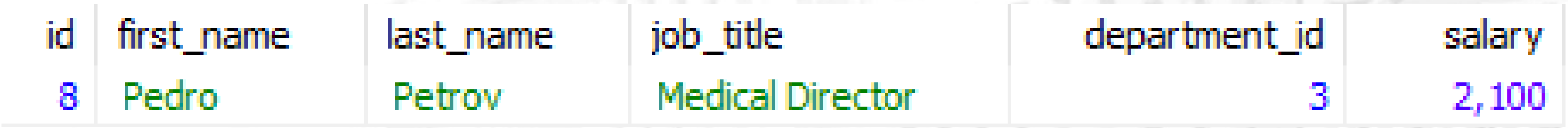
```
CREATE VIEW `v_hr_result_set` AS
SELECT
    CONCAT(`first_name`,' ',`last_name`) AS 'Full Name', `salary`
FROM `employees` ORDER BY `department_id`;
```

```
SELECT * FROM `v_hr_result_set`;
```

Problem: Top Paid Employee

- Create a view that selects all information about the top paid employee
  - Name the view `v_top_paid_employee`

```
SELECT * FROM `v_top_paid_employee`;
```



id	first_name	last_name	job_title	department_id	salary
8	Pedro	Petrov	Medical Director	3	2,100

- Note: Query Geography database

Solution: Top Paid Employee

```
CREATE VIEW `v_top_paid_employee`
AS
    SELECT * FROM `employees`
    ORDER BY `salary` DESC LIMIT 1;
```

Sorting column

Greatest value first

Writing Data in Tables

Using SQL INSERT

Inserting Data

- The SQL INSERT command

```
INSERT INTO `towns` VALUES (33, 'Paris');
```

Values for all columns

```
INSERT INTO projects(`name`, `start_date`)
     VALUES ('Reflective Jacket', NOW())
```

Specify columns

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

```
INSERT INTO `employees_projects`
     VALUES (229, 1),
            (229, 2),
            (229, 3), …
```

Inserting Data

- You can use existing records to create a new table

```
CREATE TABLE `customer_contacts`
AS SELECT `customer_id`, `first_name`, `email`, `phone`
FROM `customers`;
```

New table name

Existing source

- Or into an existing table

List of columns

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

Modifying Existing Records

Using SQL UPDATE and DELETE

UNIVERSITY OF RUSE
“ANGEL KANCHEV”

Co-funded by the
Erasmus+ Programme
of the European Union

Deleting Data

- Deleting specific rows from a table

Condition

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

  - Note: Don’t forget the WHERE clause!

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

```
TRUNCATE TABLE users;
```

Updating Data

- The SQL **UPDATE** command

New values

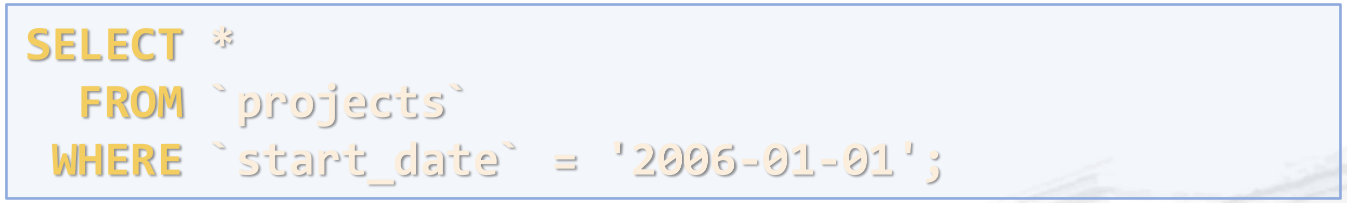
```
UPDATE `employees`
   SET `last_name` = 'Brown'
 WHERE `employee_id` = 1;
```

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

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

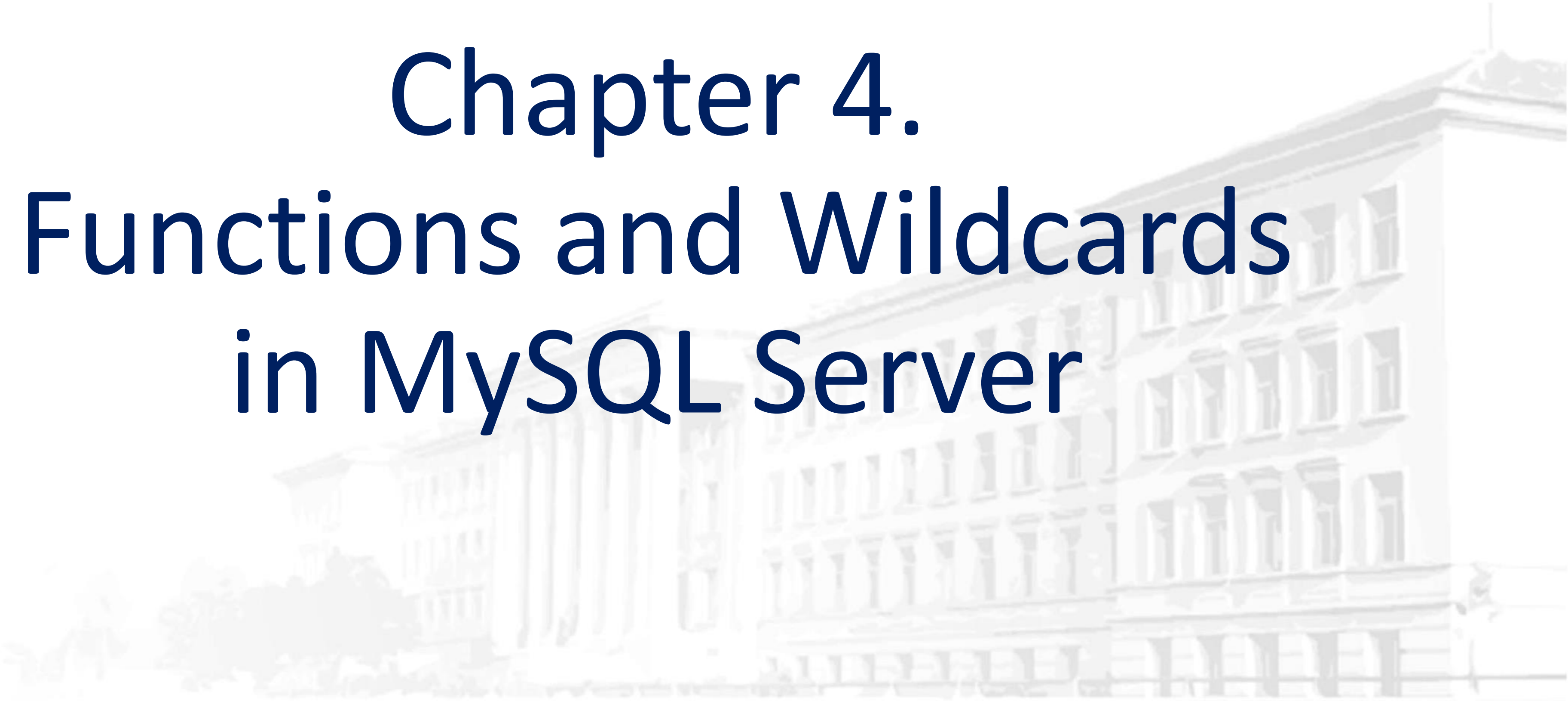
Summary

- We can easy manipulate our database with SQL queries



```
SELECT *
  FROM `projects`
 WHERE `start_date` = '2006-01-01';
```

- Queries provide a flexible and powerful method to manipulate records



Chapter 4.
Functions and Wildcards in MySQL Server

Functions in MySQL Server

SQL Functions

- String Functions – for manipulating text, both from table values or user input
  - E.g. concatenate column values
- Math Functions – calculations and working with aggregate data
  - E.g. perform geometry and currency operations
- Date and Time Functions
  - E.g. find length of timespan
- Other

String Functions

String Functions

- SUBSTRING() – extracts part of a string

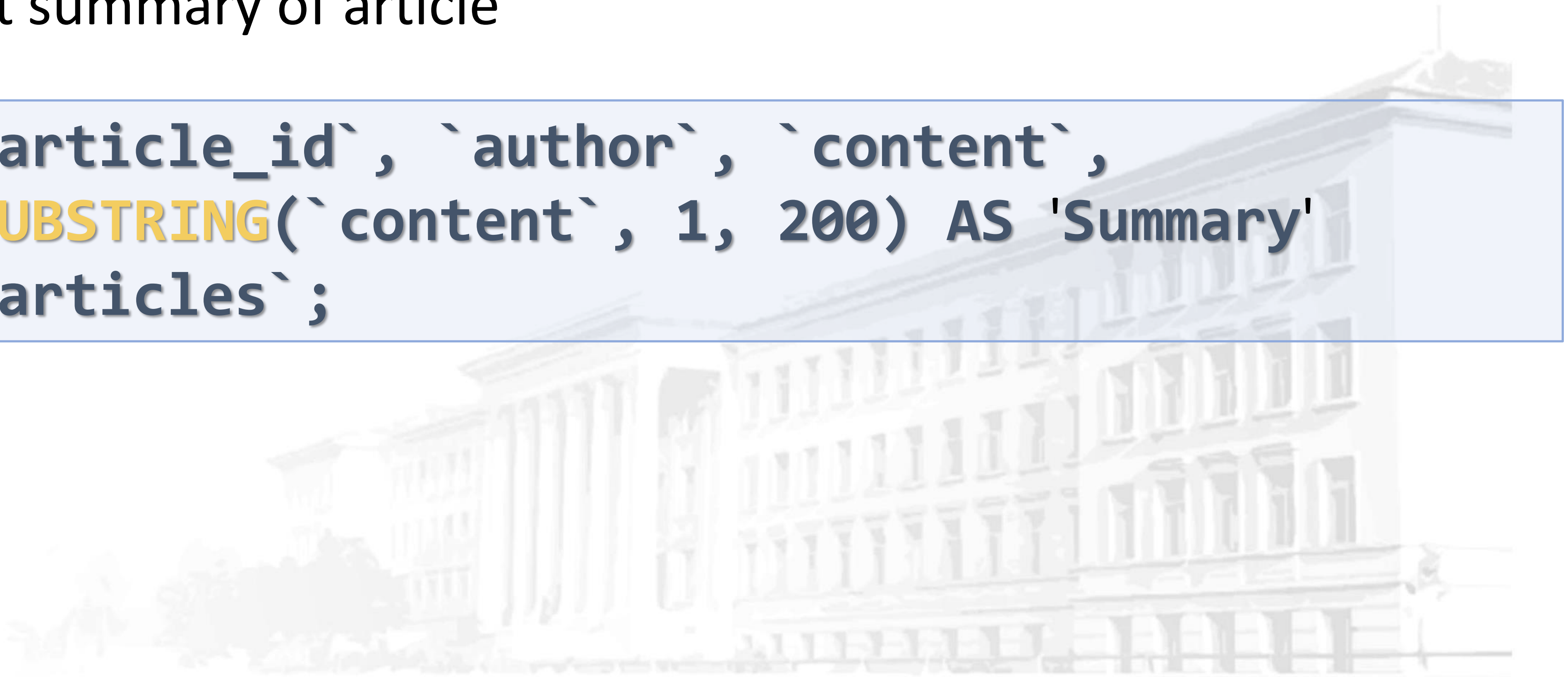
```
SUBSTRING(String, Position)
```

```
SUBSTRING(String, Position, Length)
```

```
SUBSTRING(String FROM Position FOR Length)
```

SUBSTRING - Example

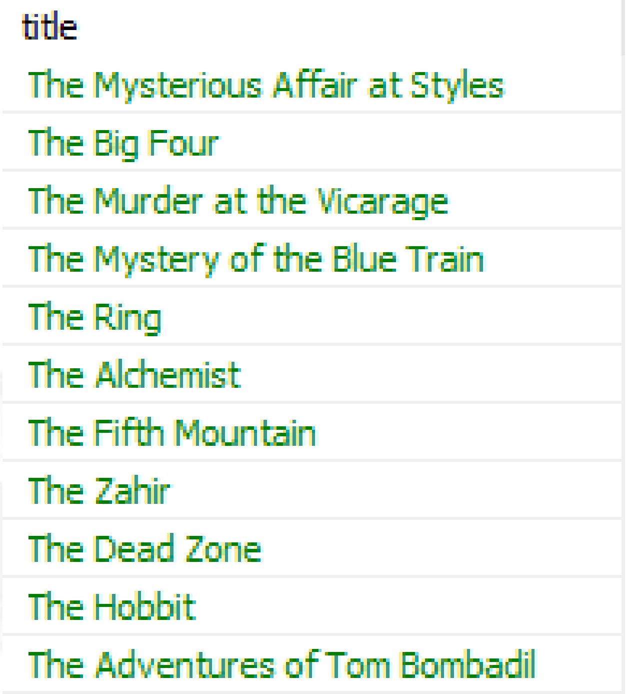
- Get short summary of article



```
SELECT `article_id`, `author`, `content`,
       SUBSTRING(`content`, 1, 200) AS 'Summary'
  FROM `articles`;
```

Problem: Find Book Titles

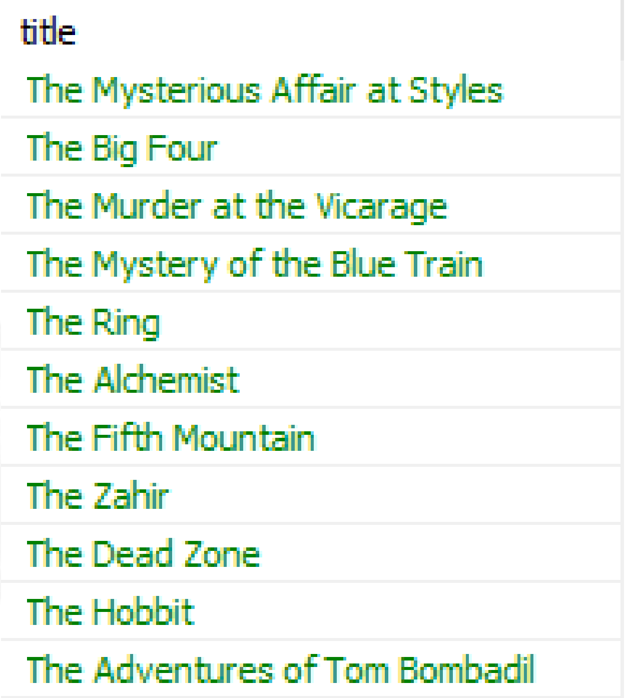
- Write a query to find all book titles that start with "The"
  - Query book_library database



title
The Mysterious Affair at Styles
The Big Four
The Murder at the Vicarage
The Mystery of the Blue Train
The Ring
The Alchemist
The Fifth Mountain
The Zahir
The Dead Zone
The Hobbit
The Adventures of Tom Bombadil

Solution: Find Book Titles

```
SELECT title FROM books WHERE
SUBSTRING(title, 1, 3) = "The";
```



```
title
The Mysterious Affair at Styles
The Big Four
The Murder at the Vicarage
The Mystery of the Blue Train
The Ring
The Alchemist
The Fifth Mountain
The Zahir
The Dead Zone
The Hobbit
The Adventures of Tom Bombadil
```

String Functions

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

String to replace

```
REPLACE(String, Pattern, Replacement)
```

Field from table

Replacement pattern

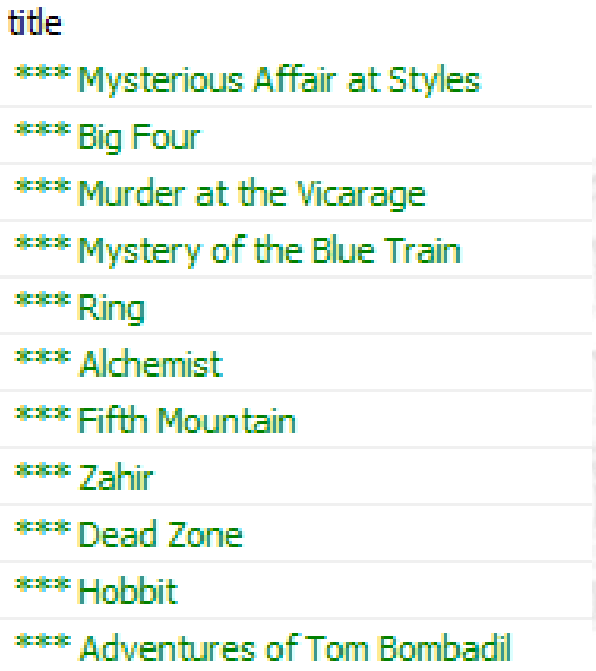
REPLACE - Example

- Censor the word blood from album names

```
SELECT REPLACE(`title`, 'blood', '*****')
    AS 'Title'
  FROM `album`;
```

Problem: Replace Titles

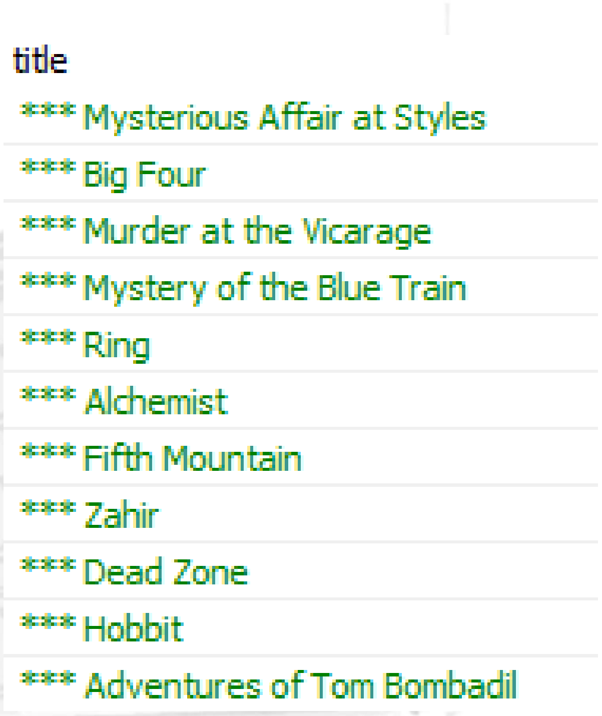
- Write a query to find all book titles that start with "The" and replace the substring with "***"
    - Query book_library database



title
*** Mysterious Affair at Styles
*** Big Four
*** Murder at the Vicarage
*** Mystery of the Blue Train
*** Ring
*** Alchemist
*** Fifth Mountain
*** Zahir
*** Dead Zone
*** Hobbit
*** Adventures of Tom Bombadil

Solution: Replace Titles

```
UPDATE books
SET title = REPLACE(title,"The","***")
WHERE SUBSTRING(title, 1, 3) = "The";
SELECT title from books
WHERE SUBSTRING(title, 1, 3) = "***";
```



```
title
*** Mysterious Affair at Styles
*** Big Four
*** Murder at the Vicarage
*** Mystery of the Blue Train
*** Ring
*** Alchemist
*** Fifth Mountain
*** Zahir
*** Dead Zone
*** Hobbit
*** Adventures of Tom Bombadil
```

String Functions

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

```
LTRIM(String)
RTRIM(String)
```

- CHAR_LENGTH – count number of characters

```
CHAR_LENGTH(String)
```

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

```
LENGTH(String)
```

String Functions

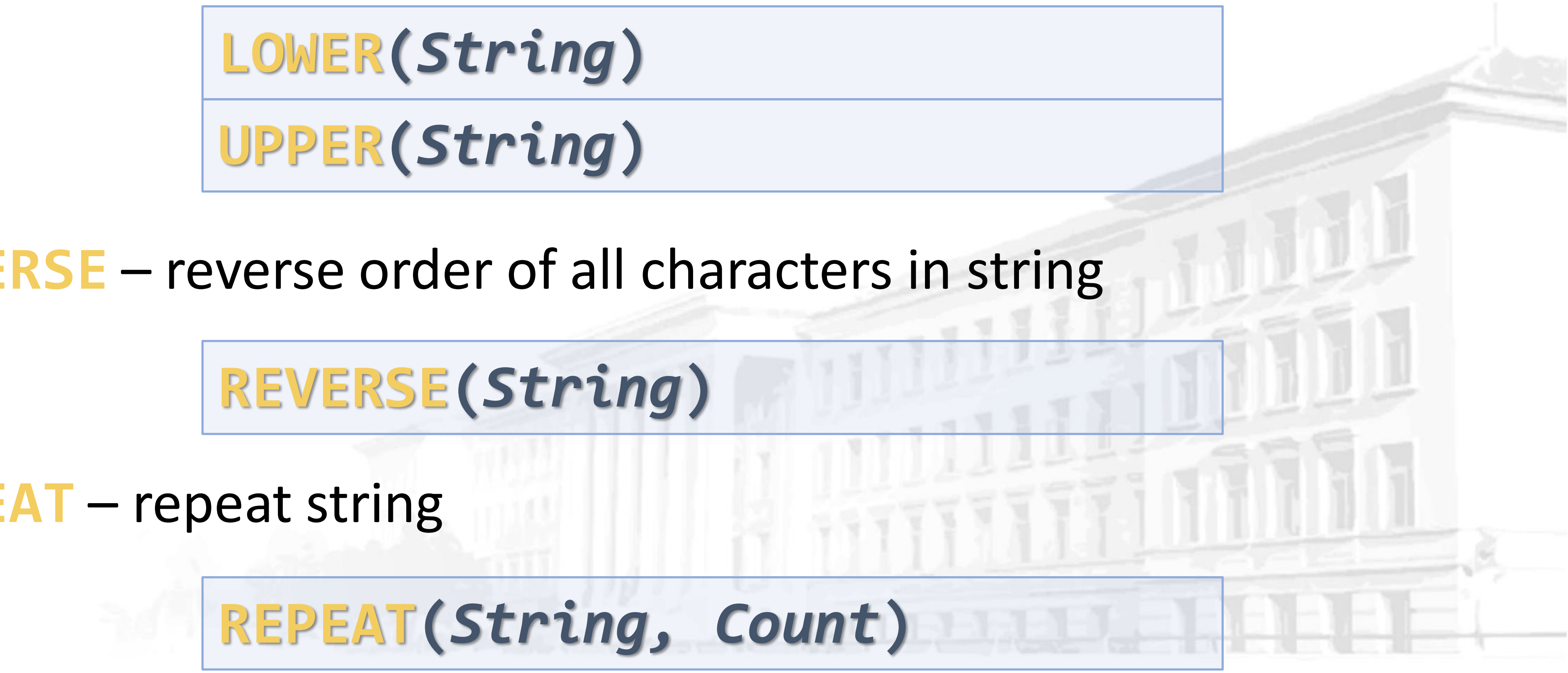
- LEFT & RIGHT – get characters from beginning or end of string

```
LEFT(String, Count)
RIGHT(String, Count)
```

- Example: name shorthand (first 3 letters)

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

String Functions



- LOWER & UPPER – change letter casing

```
LOWER(String)
UPPER(String)
```

- REVERSE – reverse order of all characters in string

```
REVERSE(String)
```

- REPEAT – repeat string

```
REPEAT(String, Count)
```

String Functions

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

If omitted, begins at 1

```
LOCATE(Pattern, String,[Position])
```

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

```
INSERT(String, Position, Length, Substring)
```

Number of characters to delete

Arithmetical Operators and Numeric Functions

Arithmetical Operators

- Supported common arithmetic 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)
```

Numeric Functions

- SQRT – square root

```
SQRT(Value)
```

- POW – raise value to desired exponent

```
POW(Value, Exponent)
```

Math Functions

- CONV – Converts numbers between different number bases

```
CONV(Value,from_base,to_base)
```

- ROUND – obtain desired precision

```
ROUND(Value, Precision)
```

Can be negative

- FLOOR & CEILING – return the nearest integer

```
FLOOR(Value)
```

```
CEILING(Value)
```

Math Functions

- 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

```
RAND()
RAND(Seed)
```

Date Functions

Date Functions

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

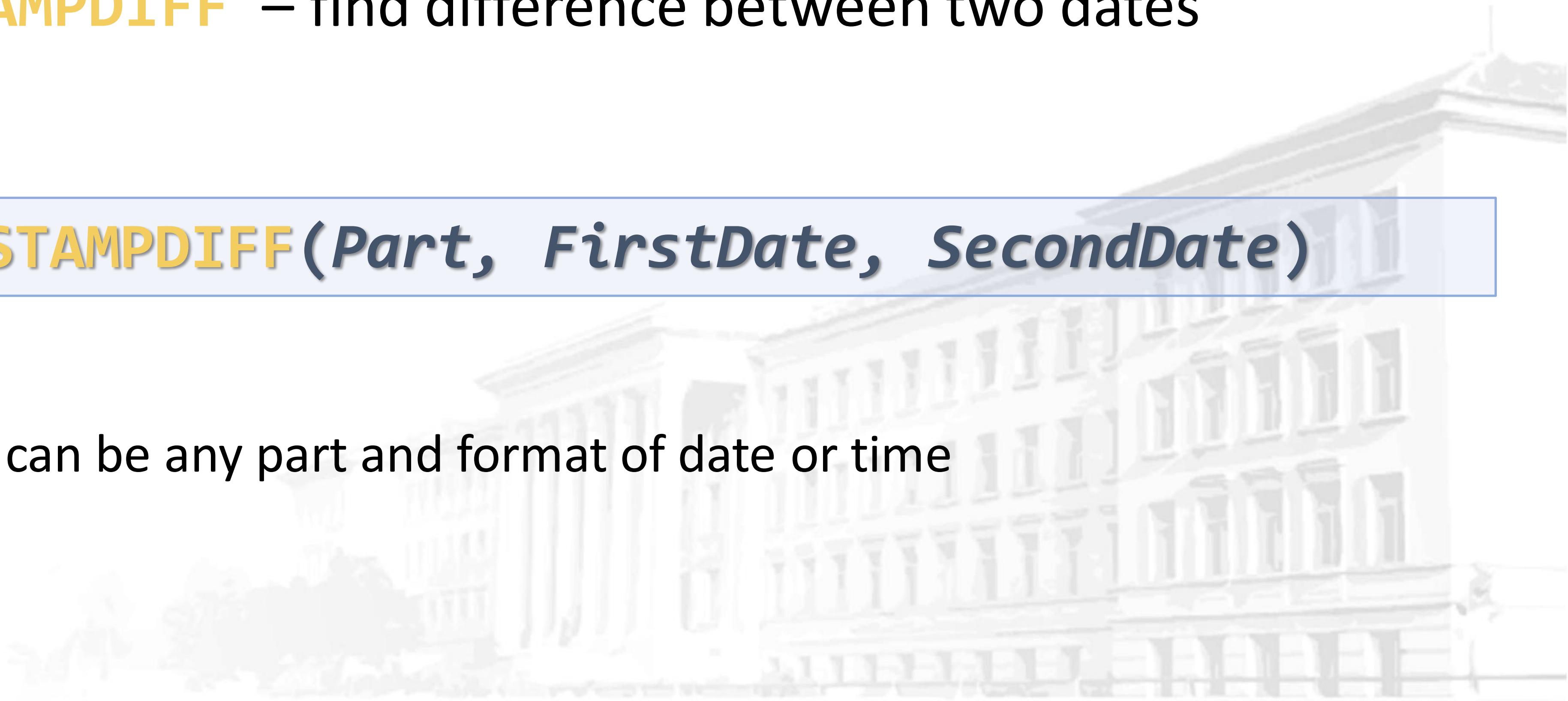
```
EXTRACT(Part FROM Date)
```

  - ***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)`

- For a full list, see the official documentation

Date Functions



- TIMESTAMPDIFF – find difference between two dates

```
TIMESTAMPDIFF(Part, FirstDate, SecondDate)
```

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

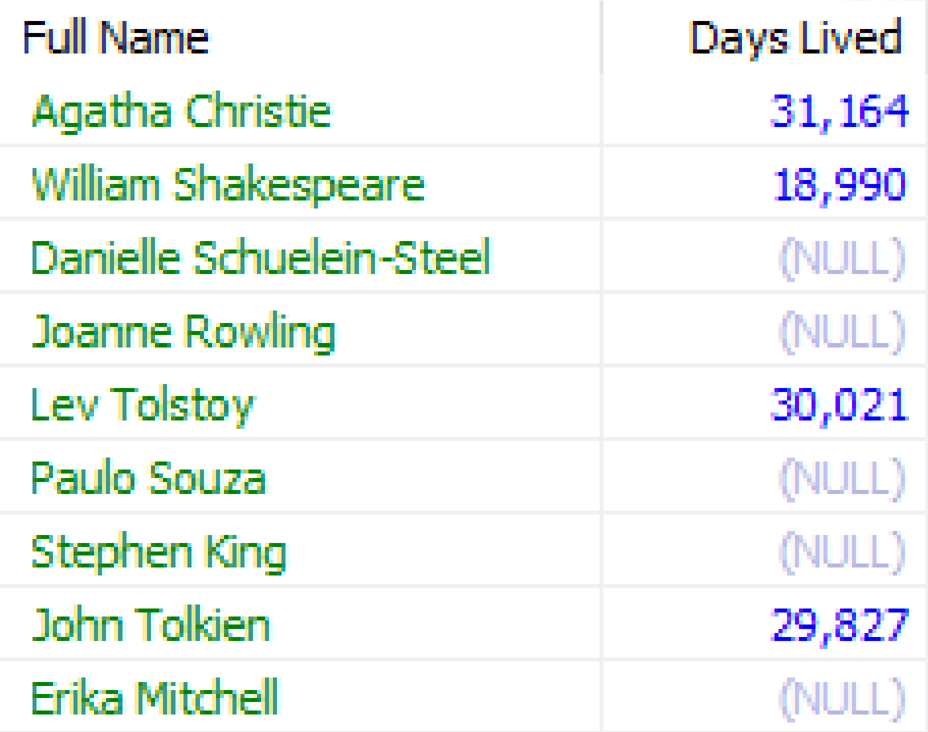
Date Functions - Example

- Show employee experience

```
SELECT `employee_id`, `first_name`, `last_name`,
       TIMESTAMPDIFF(year, `hire_date`, '2017-05-31')
    AS 'Years In Service'
  FROM `employees`;
```

Problem: Days Lived

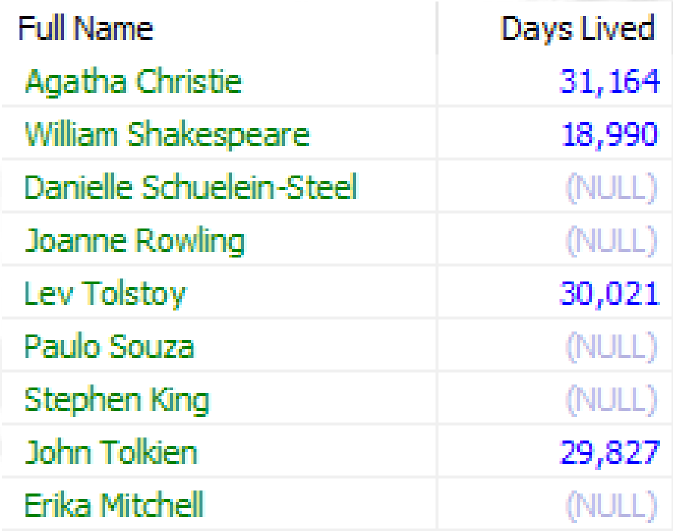
- Write a query to calculate how many days have authors lived
  - Use TIMESTAMPDIFF
  - Query book_library database



Full Name	Days Lived
Agatha Christie	31,164
William Shakespeare	18,990
Danielle Schuelein-Steel	(NULL)
Joanne Rowling	(NULL)
Lev Tolstoy	30,021
Paulo Souza	(NULL)
Stephen King	(NULL)
John Tolkien	29,827
Erika Mitchell	(NULL)

Days Lived - Solution

```
SELECT  concat(first_name, ' ', last_name) as 'Full Name', TIMESTAMPDIFF(DAY,
born, died) as 'Days Lived'
FROM authors;
```



Full Name	Days Lived
Agatha Christie	31,164
William Shakespeare	18,990
Danielle Schuelein-Steel	(NULL)
Joanne Rowling	(NULL)
Lev Tolstoy	30,021
Paulo Souza	(NULL)
Stephen King	(NULL)
John Tolkien	29,827
Erika Mitchell	(NULL)

Date Functions

- DATE_FORMAT – formats the date value according to the format

```
SELECT DATE_FORMAT('2017/05/31', '%Y %b %D') AS 'Date';
```

- NOW – obtain current date and time

```
SELECT NOW();
```

Wildcards

Selecting results by partial match

Wildcards

- 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

Wildcards - Examples

- Find any values that start with "a"

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

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

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

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

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

Wildcard Characters

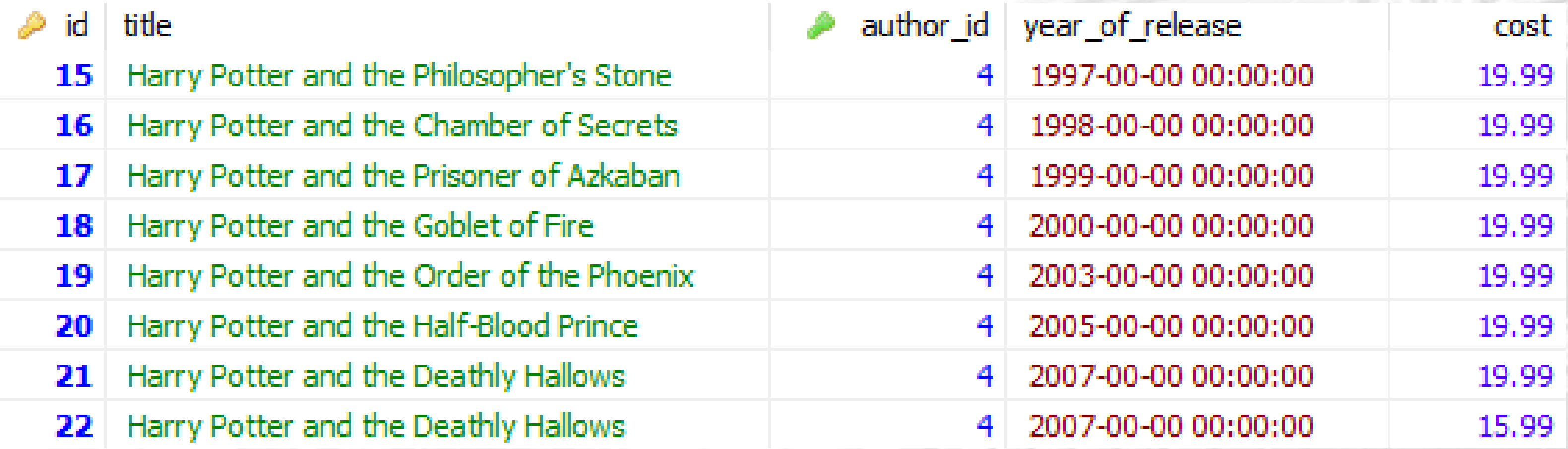
- Supported characters also include:
  - \ – specify prefix to treat special characters as normal
  - [charlist] – specifying which characters to look for
    - [!charlist] – excluding characters

```
SELECT * FROM `customers`
WHERE `city` LIKE '[a-c]%';
```

"a", "b", or "c"

Problem: Harry Potter Books

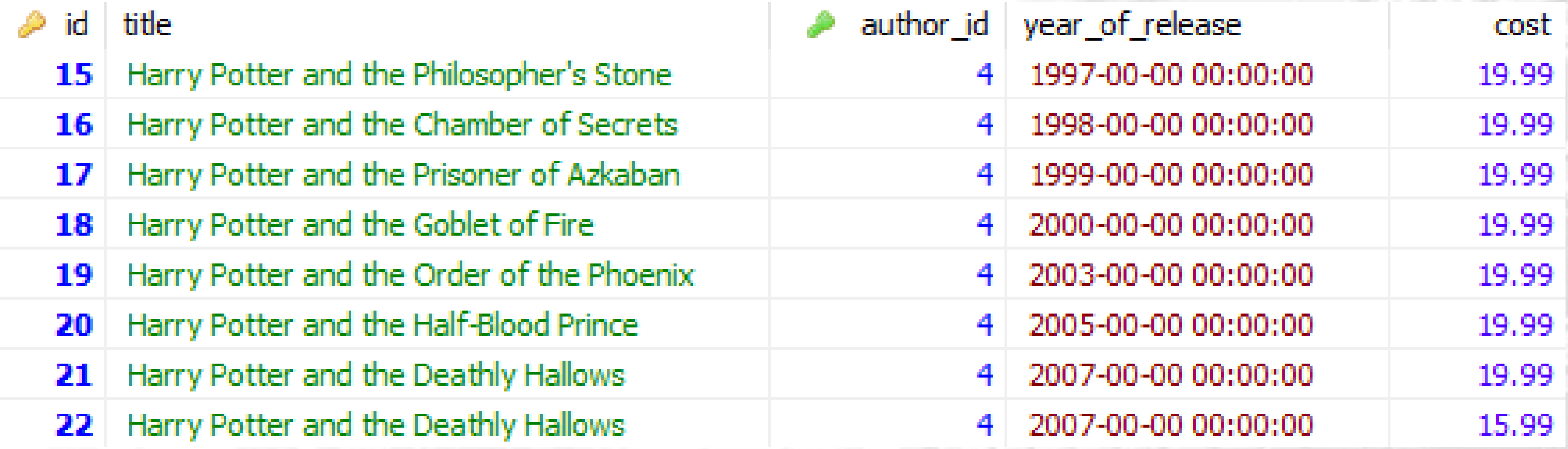
- Write a query to retrieve information about the titles of all Harry Potter books
  - Use Wildcards
  - Query book_library database



id	title	author_id	year_of_release	cost
15	Harry Potter and the Philosopher's Stone	4	1997-00-00 00:00:00	19.99
16	Harry Potter and the Chamber of Secrets	4	1998-00-00 00:00:00	19.99
17	Harry Potter and the Prisoner of Azkaban	4	1999-00-00 00:00:00	19.99
18	Harry Potter and the Goblet of Fire	4	2000-00-00 00:00:00	19.99
19	Harry Potter and the Order of the Phoenix	4	2003-00-00 00:00:00	19.99
20	Harry Potter and the Half-Blood Prince	4	2005-00-00 00:00:00	19.99
21	Harry Potter and the Deathly Hallows	4	2007-00-00 00:00:00	19.99
22	Harry Potter and the Deathly Hallows	4	2007-00-00 00:00:00	15.99

Harry Potter Books - Solution

```
SELECT title FROM books
WHERE title LIKE 'Harry Potter%';
```



id	title	author_id	year_of_release	cost
15	Harry Potter and the Philosopher's Stone	4	1997-00-00 00:00:00	19.99
16	Harry Potter and the Chamber of Secrets	4	1998-00-00 00:00:00	19.99
17	Harry Potter and the Prisoner of Azkaban	4	1999-00-00 00:00:00	19.99
18	Harry Potter and the Goblet of Fire	4	2000-00-00 00:00:00	19.99
19	Harry Potter and the Order of the Phoenix	4	2003-00-00 00:00:00	19.99
20	Harry Potter and the Half-Blood Prince	4	2005-00-00 00:00:00	19.99
21	Harry Potter and the Deathly Hallows	4	2007-00-00 00:00:00	19.99
22	Harry Potter and the Deathly Hallows	4	2007-00-00 00:00:00	15.99

Using Regular Expression

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

```
SELECT `employee_id`, `first_name`, `last_name`
 FROM `employees`
 WHERE `first_name` REGEXP '^\[^K\]{3}\$';
```

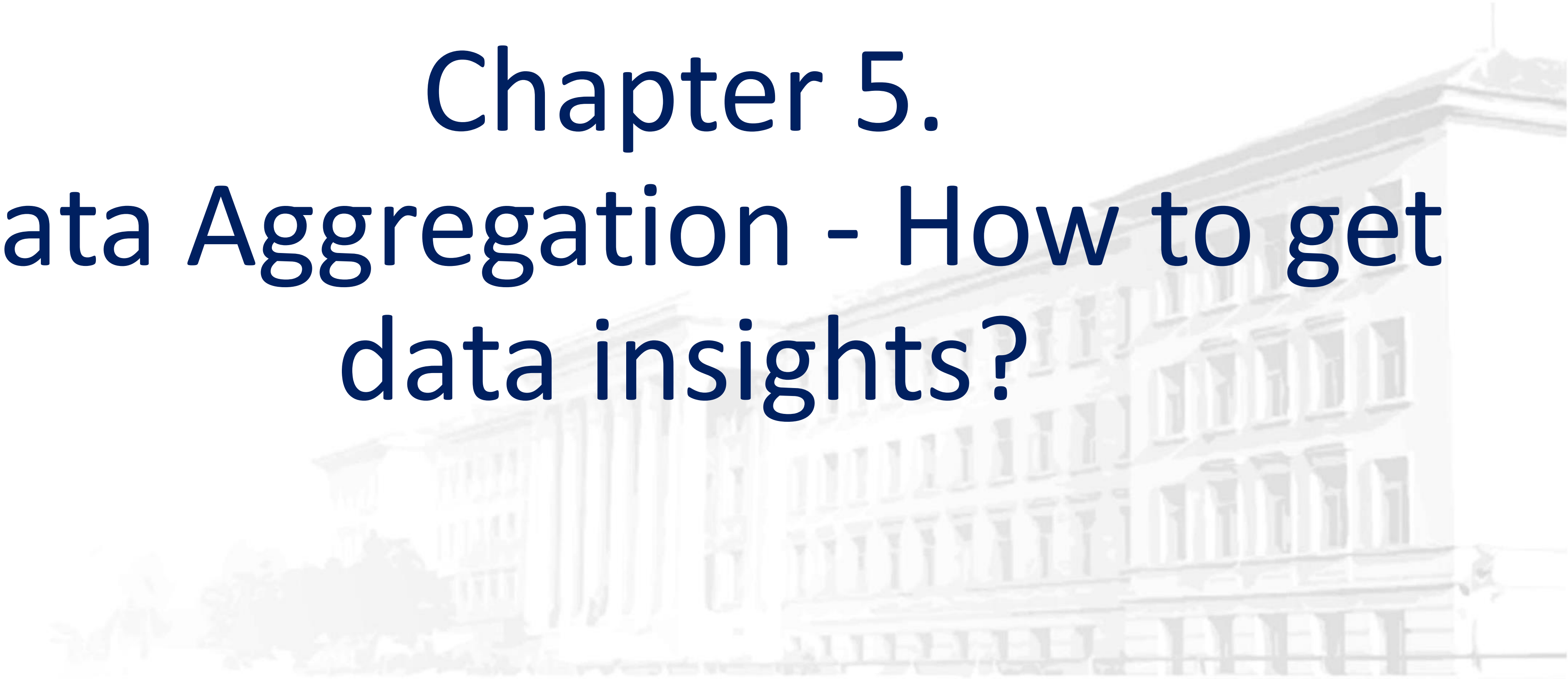
Regular expression

Summary

- MySQL Server provides various built-in functions
  - Numerical functions
  - String functions

- Using Wildcards, we can obtain results by partial string matches
  - Regular expressions

Chapter 5.
Data Aggregation - How to get data insights?



Grouping

Consolidating data based on criteria

Grouping

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

Grouping column

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

Can be aggregated

GROUP BY

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

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

Grouping Columns

DISTINCT

- With **DISTINCT** you will get all unique values:

```
SELECT DISTINCT e.`job_title`
  FROM `employees` AS e;
```

Unique Values

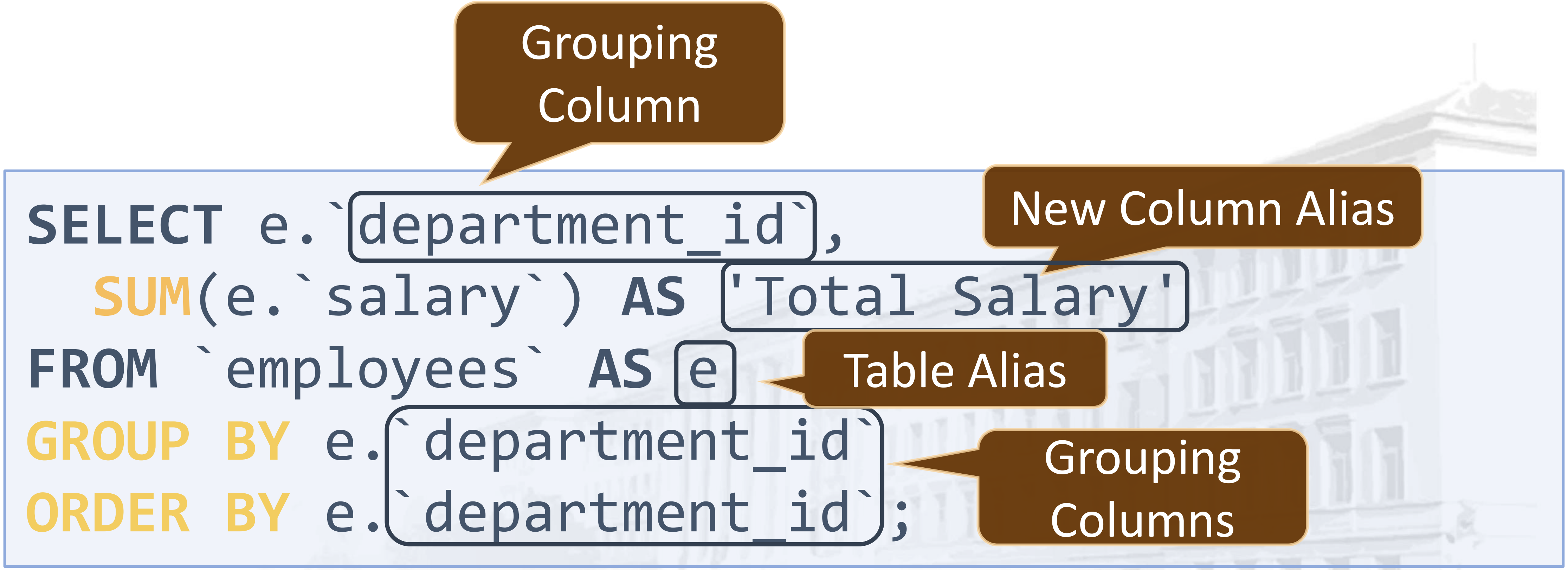
Problem: Departments Total Salaries

- Write a query which prints the total sum of salaries for each department in the uni_ruse database
  - Order them by DepartmentID (ascending)

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

Solution: Departments Total Salaries



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

Grouping Column

New Column Alias

Table Alias

Grouping Columns

Aggregate Functions

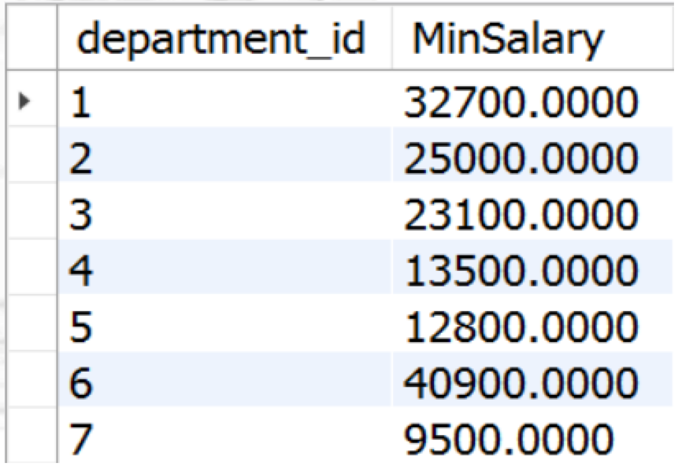
COUNT, SUM, MAX, MIN, AVG…

Aggregate Functions

- Used to operate over one or more groups performing data analysis on every one
  - MIN, MAX, AVG, COUNT etc.

- They usually ignore **NULL** values

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



department_id	MinSalary
1	32700.0000
2	25000.0000
3	23100.0000
4	13500.0000
5	12800.0000
6	40900.0000
7	9500.0000

COUNT

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

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_name	SalaryCount
Database Support	2
Application Support	3
Software Support	1

COUNT Syntax

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

Grouping Column

New Column Alias

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

Grouping Columns

SUM

- SUM - sums the values in a column

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_name	total_salary
Database Support	20,000
Application Support	30,000
Software Support	15,000

SUM Syntax

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

Grouping Column

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

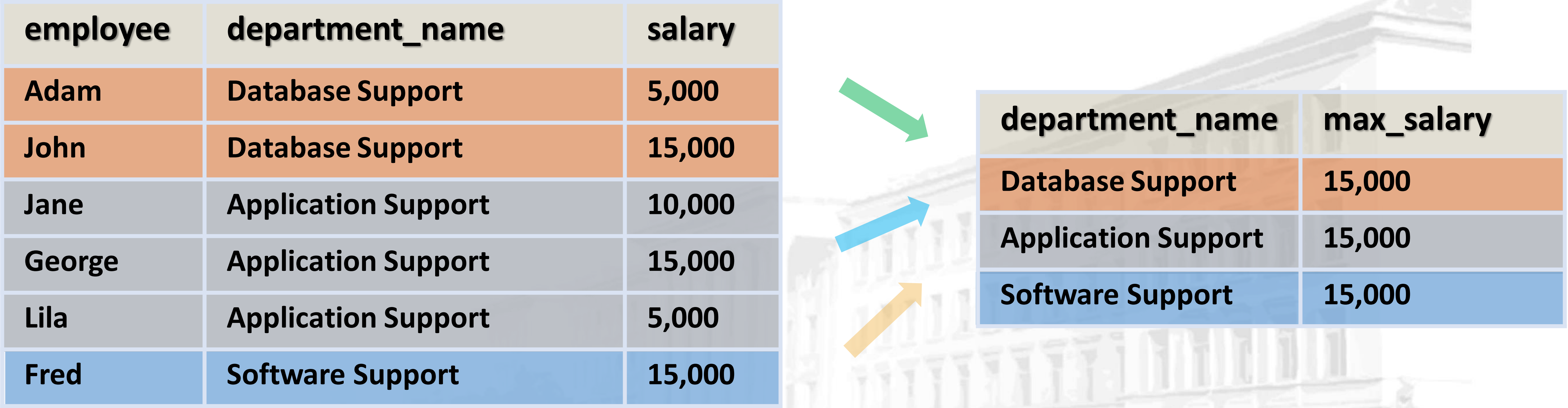
New Column Alias

Table Alias

Grouping Columns

MAX

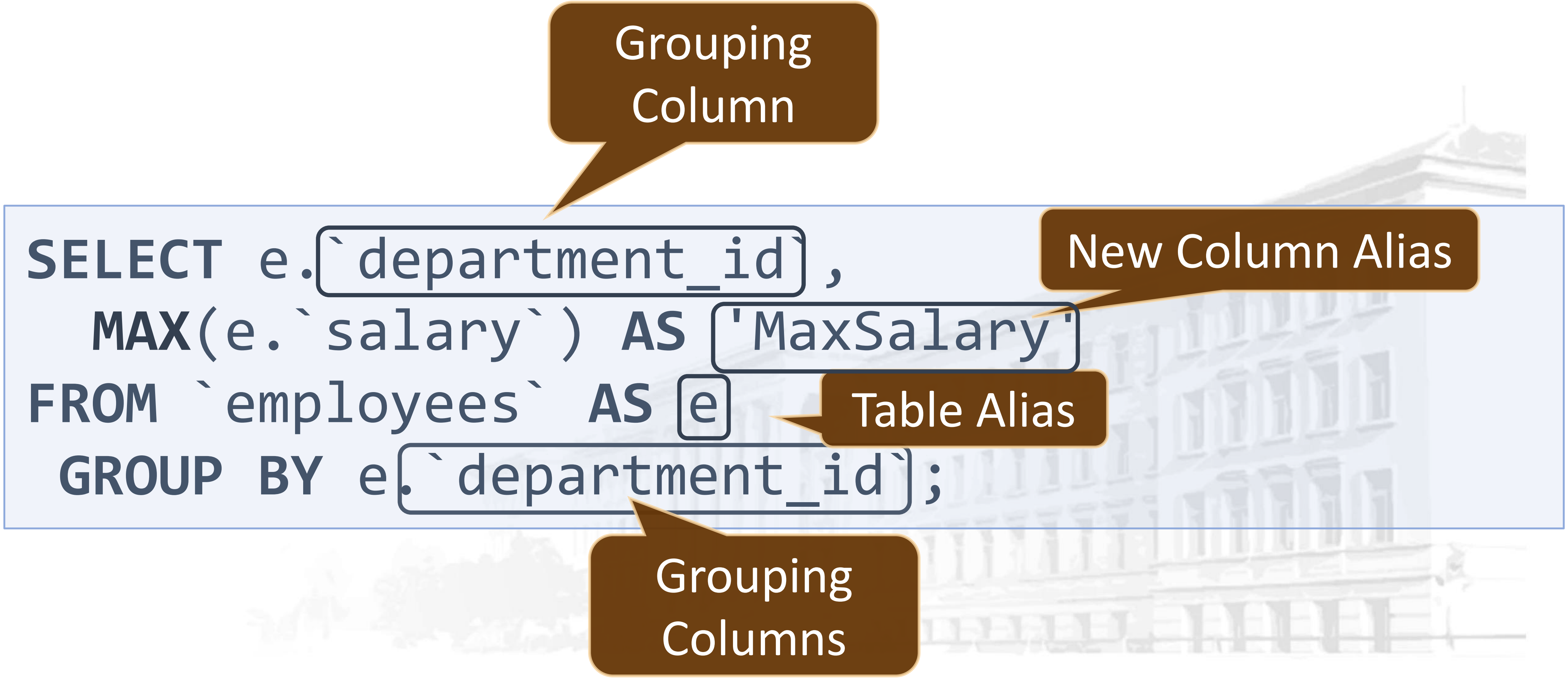
- **MAX -** takes the maximum value in a column.



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_name	max_salary
Database Support	15,000
Application Support	15,000
Software Support	15,000

MAX Syntax



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

Grouping Column

New Column Alias

Table Alias

Grouping Columns

MIN

- **MIN** takes the minimum value in a column.

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_name	min_salary
Database Support	5,000
Application Support	5,000
Software Support	15,000

MIN Syntax

Grouping Column

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

New Column Alias

Table Alias

Grouping Columns

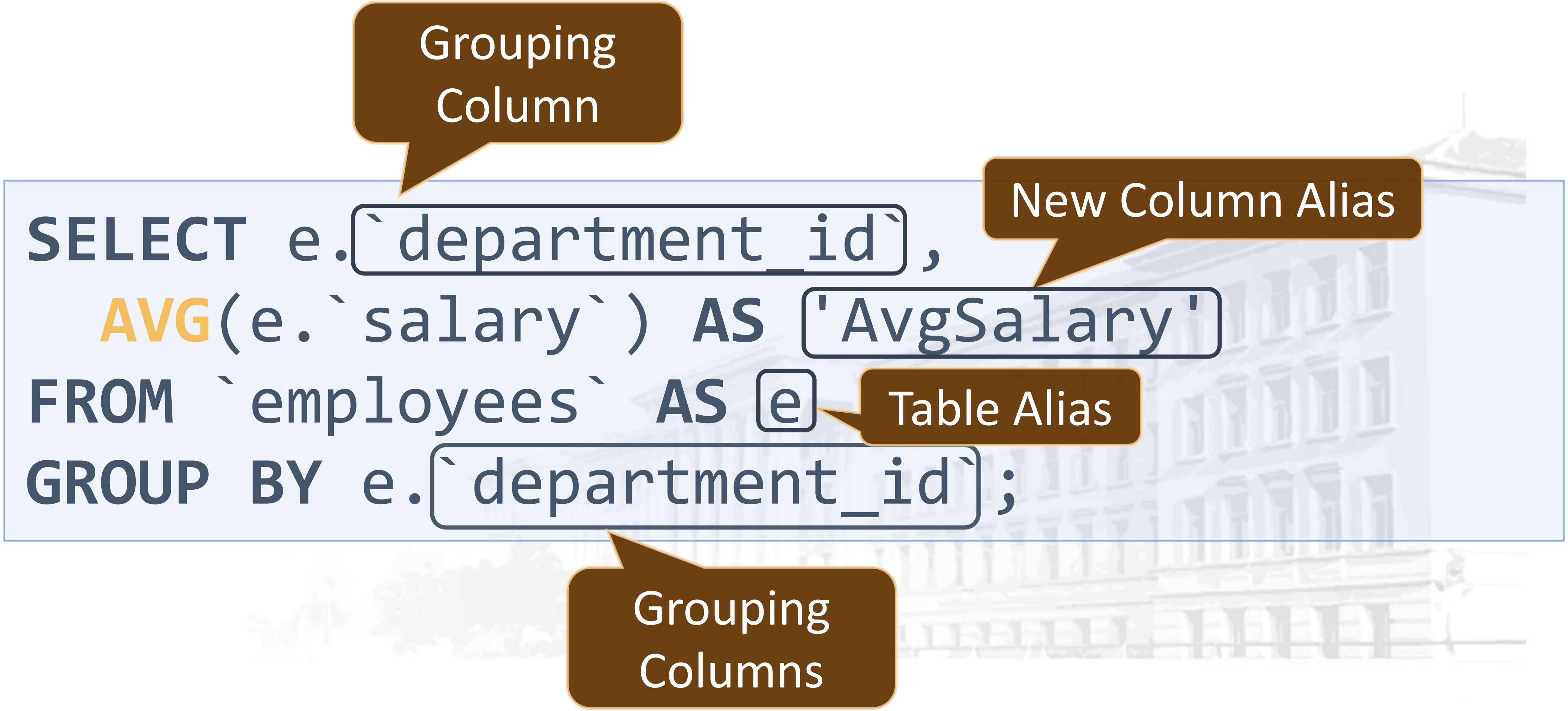
AVG

- **AVG** calculates the average value in a column.

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_name	average_salary
Database Support	10,000
Application Support	10,000
Software Support	15,000

AVG Syntax



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

Grouping Column

New Column Alias

Table Alias

Grouping Columns

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

Having Clause: Example

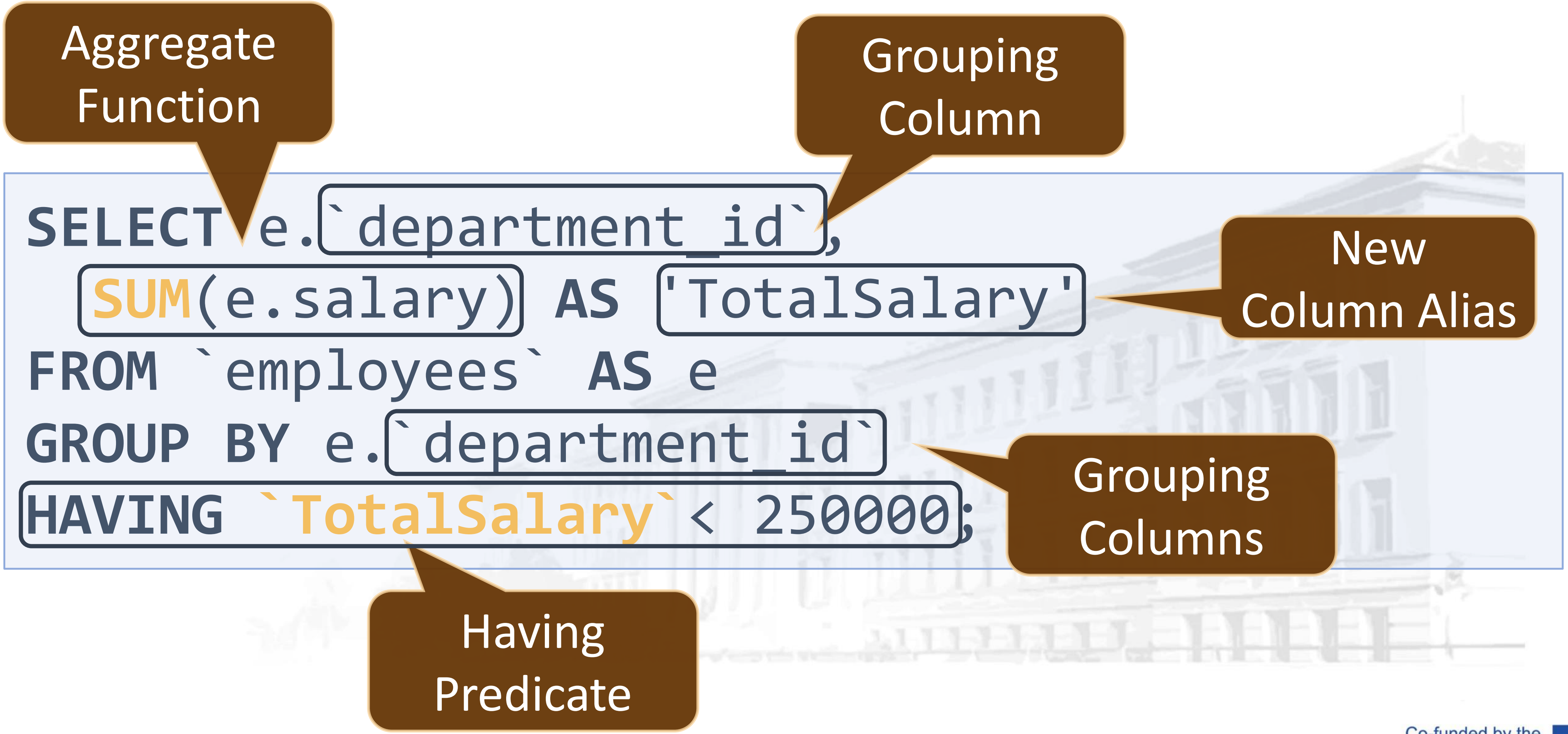
- Filter departments which have total salary more or equal 15,000.

Aggregated value

employee	department_name	salary	Total Salary
Adam	Database Support	5,000	20,000
John	Database Support	15,000	
Jane	Application Support	10,000	10,000
George	Application Support	15,000	
Lila	Application Support	5,000	
Fred	Software Support	15,000	15,000

department_name	average_salary
Database Support	10,000
Software Support	15,000

HAVING Syntax

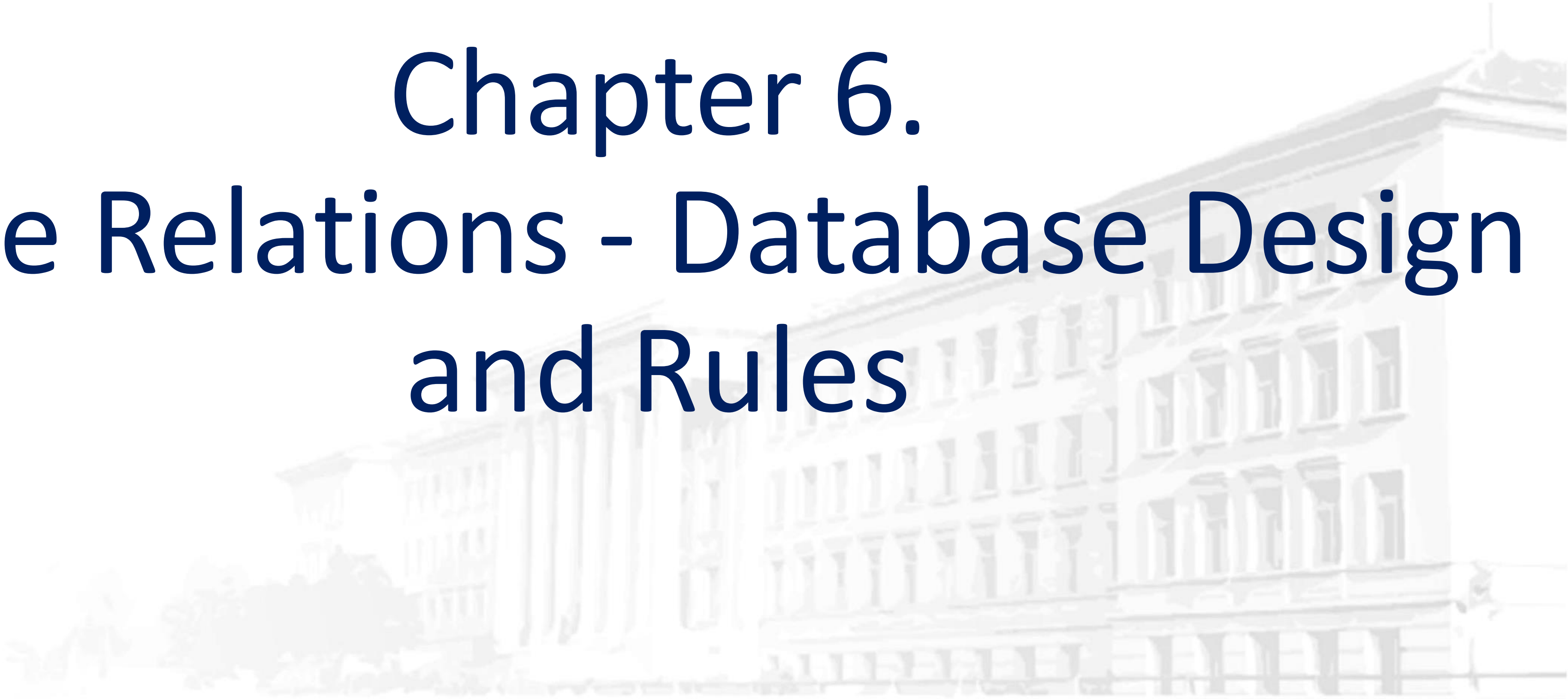


```sql
SELECT e.`department_id`,
  SUM(e.salary) AS 'TotalSalary'
FROM `employees` AS e
GROUP BY e.`department_id`
HAVING `TotalSalary` < 250000;
```

Summary

- Grouping
- Aggregate Functions
- Having

```
SELECT
  SUM(e.`salary) AS 'TotalSalary'
FROM `employees` AS e
GROUP BY e.`department_id`
HAVING SUM(e.`salary`) < 250000;
```



Chapter 6.
Table Relations - Database Design and Rules

Co-funded by the
Erasmus+ Programme
of the European Union

Database Design

Fundamental Concepts

UNIVERSITY OF RUSE
“ANGEL KANCHEV”

Co-funded by the
Erasmus+ Programme
of the European Union

Steps in Database Design

1 Identification of the entities

2 Defining table columns

3 Defining primary keys

4 Modeling relationships

5 Defining constraints

6 Filling test data

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

Identification of the 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

How to Choose a Primary Key?

- Always define an additional column for the primary key
  - Don't use an existing column
  - Must 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)

Identification of Relationships

- Relationships are dependencies between the entities:

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

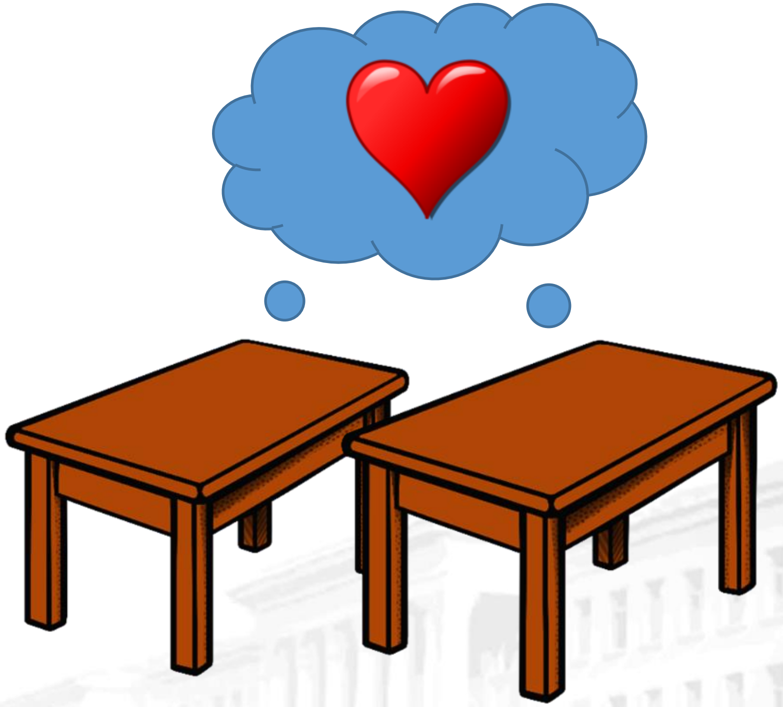


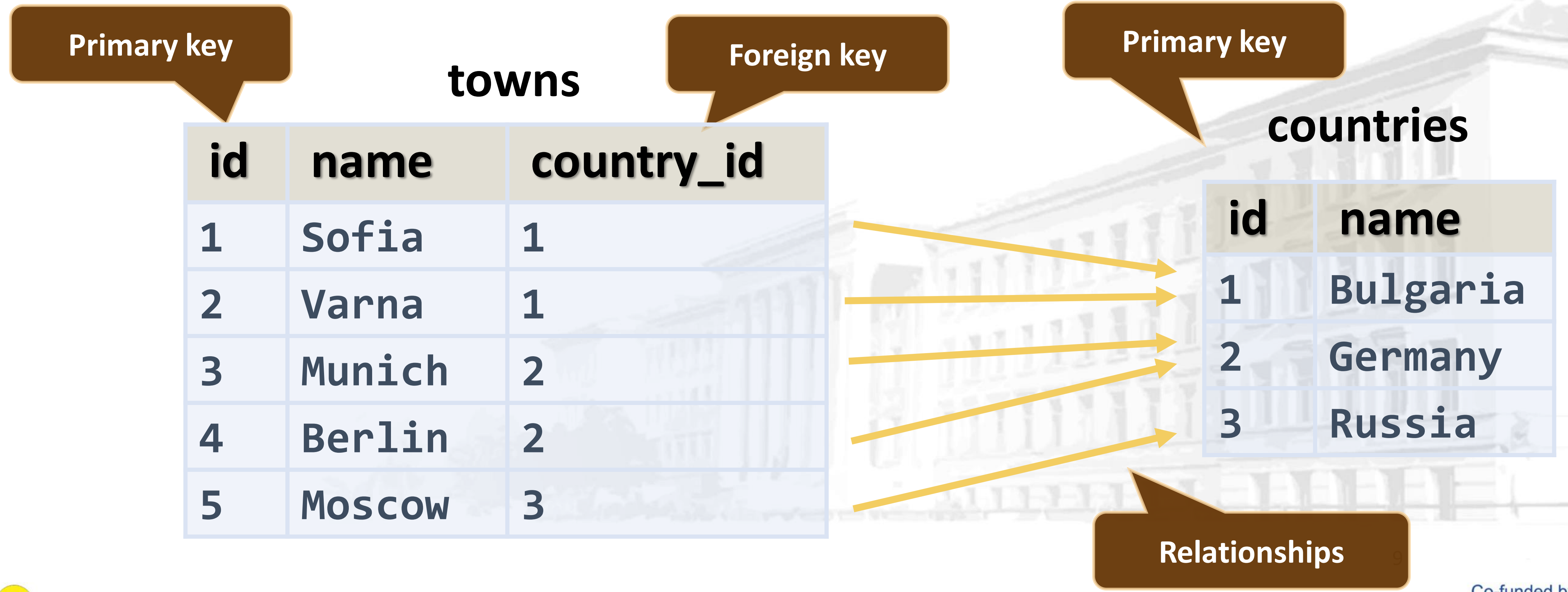
Table Relations

Relational Database Model in Action

Co-funded by the
Erasmus+ Programme
of the European Union

Relationships

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



Primary key

Foreign key

towns

id	name	country_id
1	Sofia	1
2	Varna	1
3	Munich	2
4	Berlin	2
5	Moscow	3

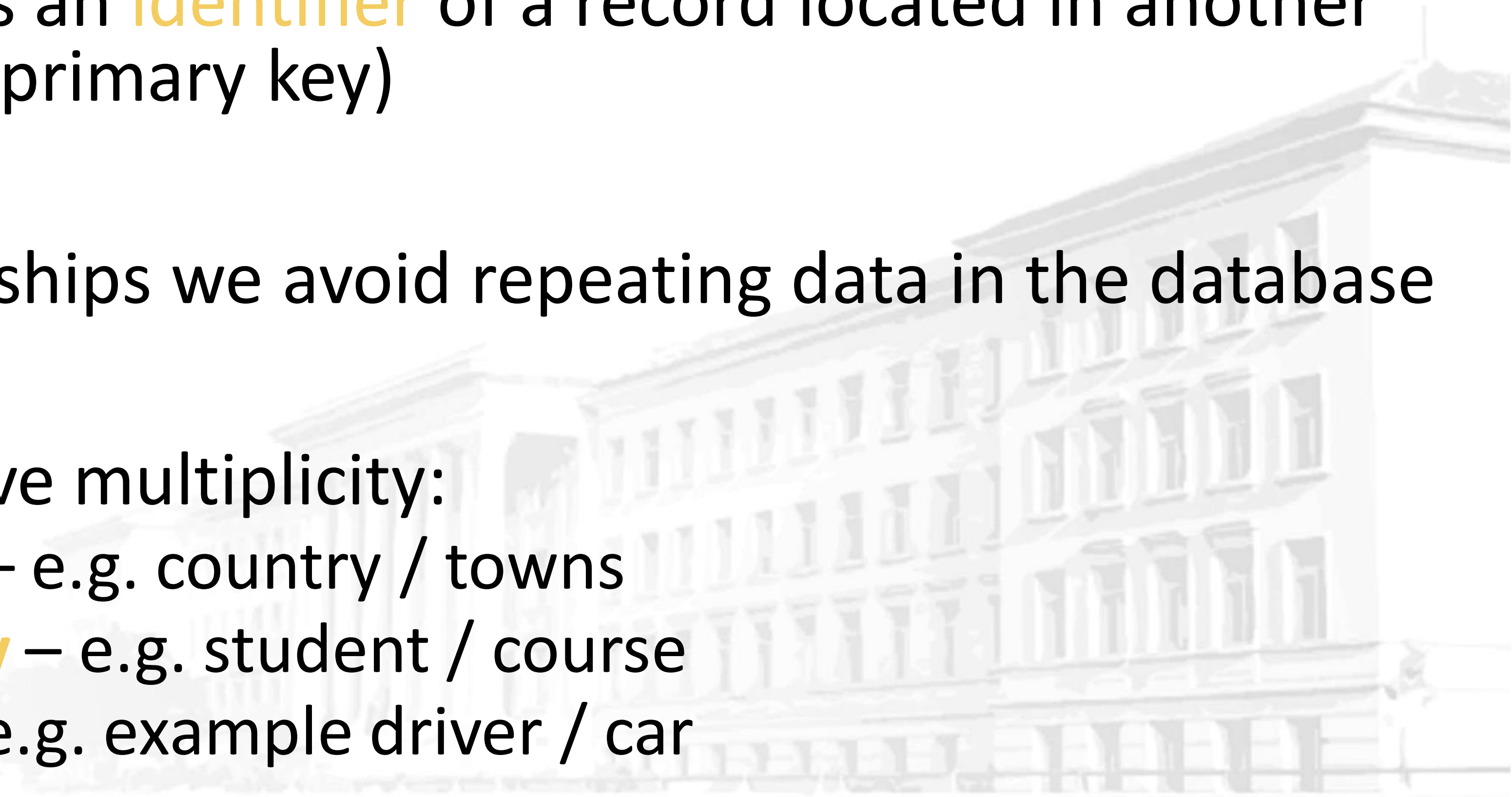
Primary key

countries

id	name
1	Bulgaria
2	Germany
3	Russia

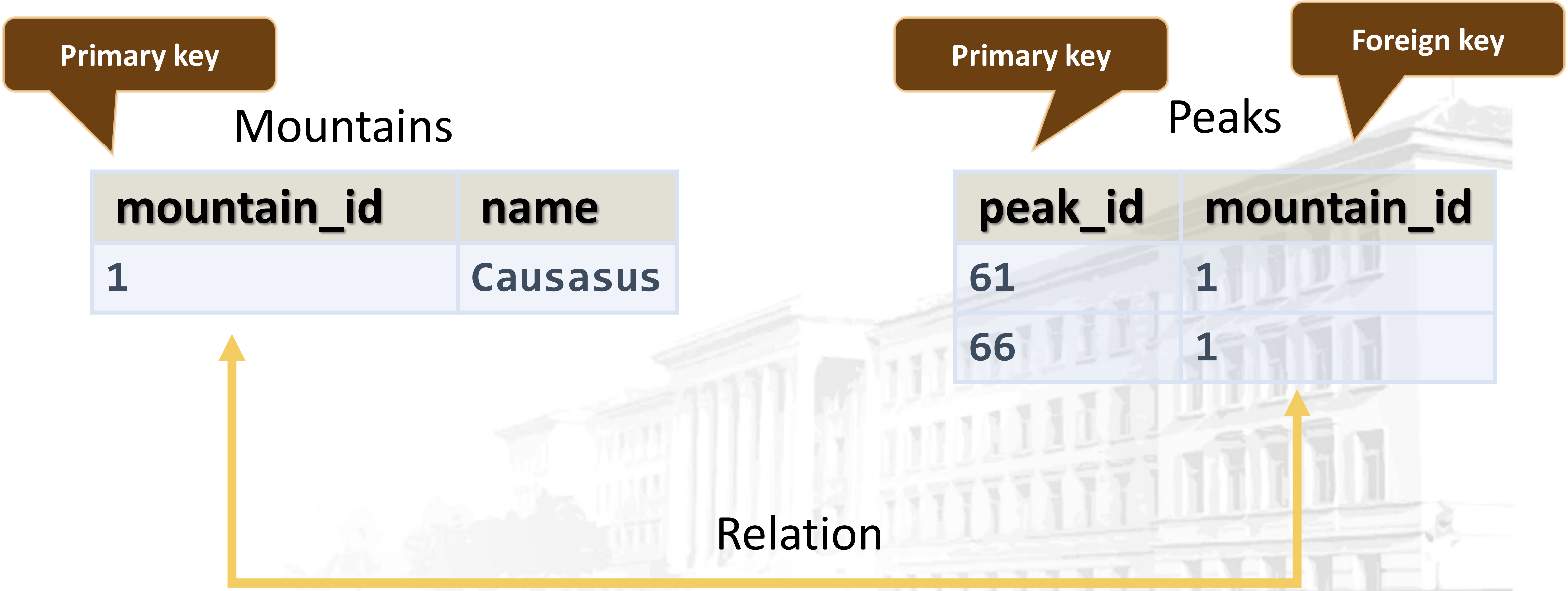
Relationships

Relationships



- 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. country / towns
  - **Many-to-many** – e.g. student / course
  - **One-to-one** – e.g. example driver / car

One-to-Many/Many-to-One



Primary key

Mountains

mountain_id	name
1	Causasus

Primary key

Foreign key

Peaks

peak_id	mountain_id
61	1
66	1

Relation

Setup

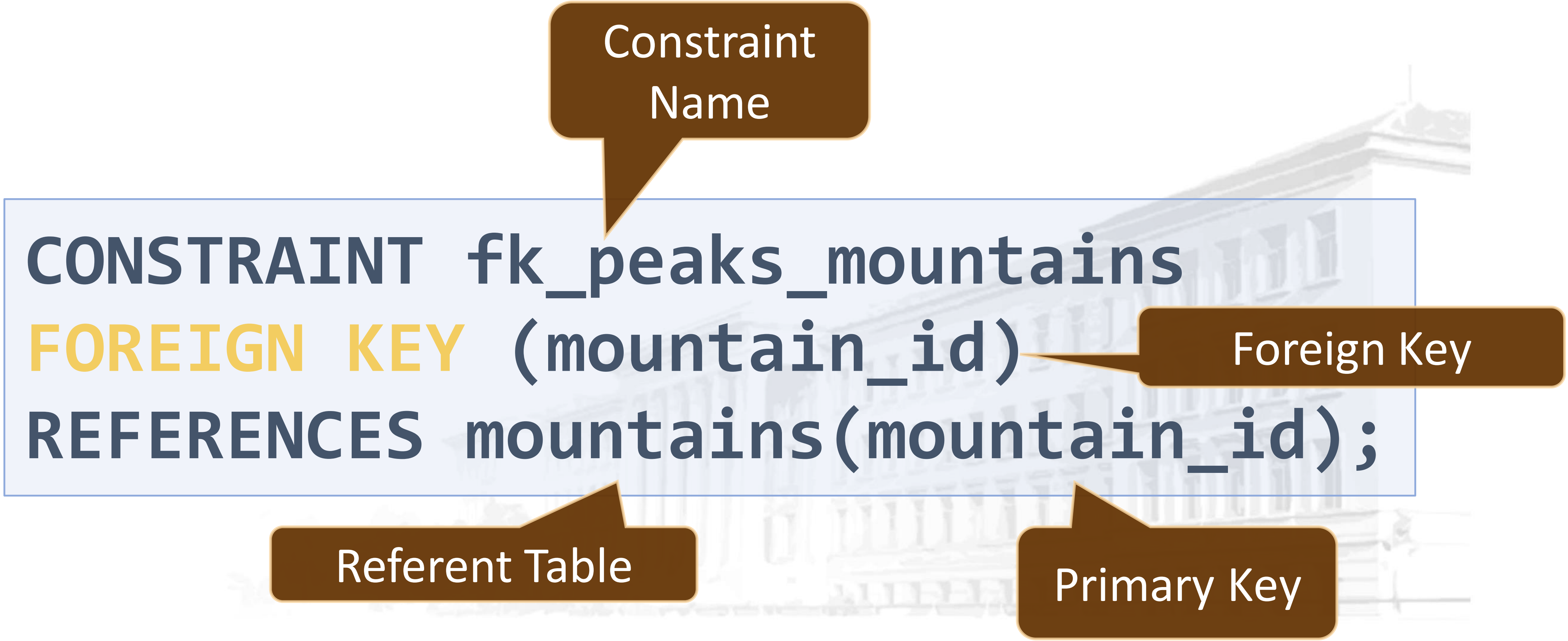
```
CREATE TABLE mountains(
  mountain_id INT PRIMARY KEY,
  mountain_name VARCHAR(50)
);
CREATE TABLE peaks(
  peak_id INT PRIMARY KEY,
  mountain_id INT,
  CONSTRAINT fk_peaks_mountains
  FOREIGN KEY (mountain_id)
  REFERENCES mountains(mountain_id)
);
```

Primary key

Table Peaks

Foreign Key

Foreign Key



```
CONSTRAINT fk_peaks_mountains
FOREIGN KEY (mountain_id)
REFERENCES mountains(mountain_id);
```

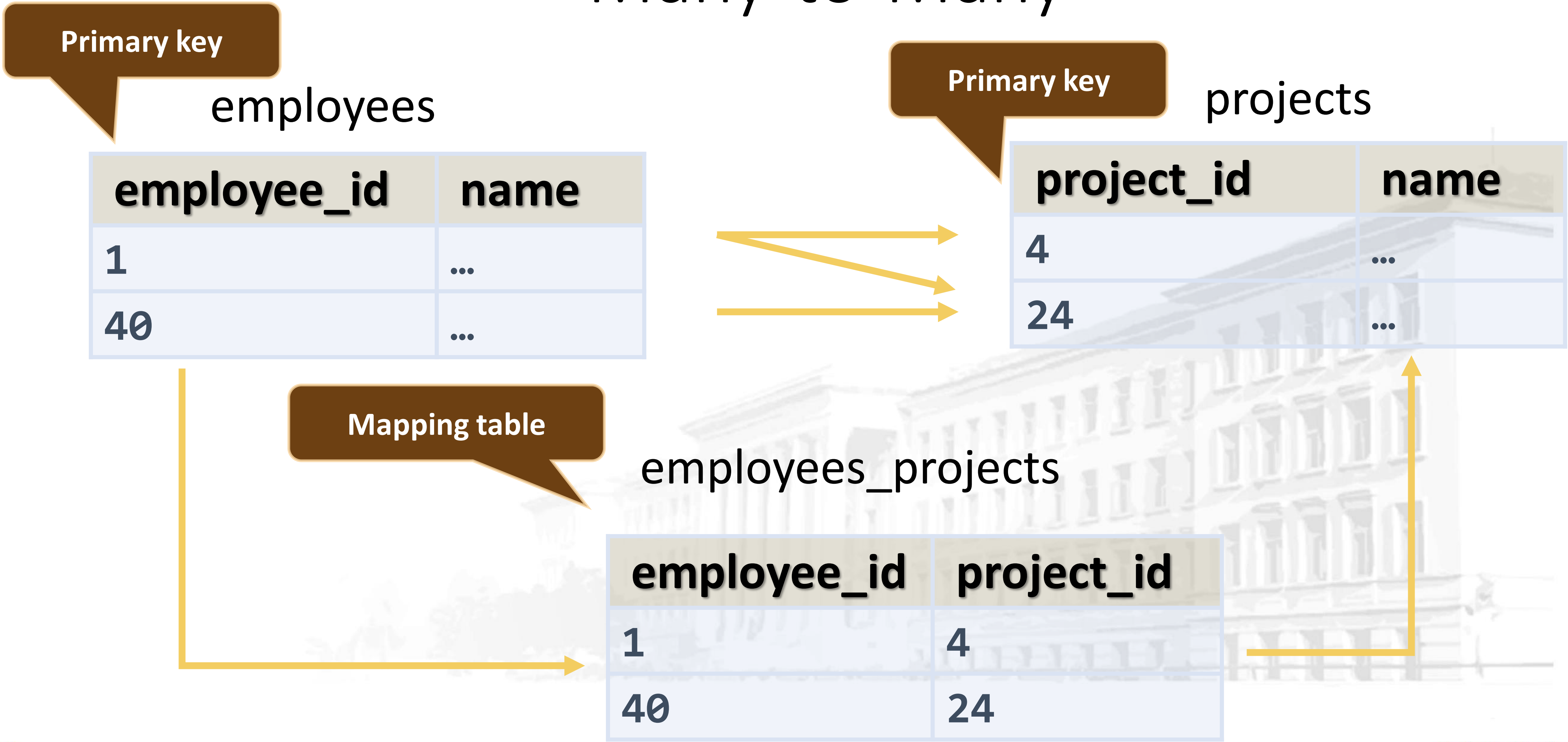
Constraint Name

Foreign Key

Referent Table

Primary Key

Many-to-Many



employees

employee_id	name
1	…
40	…

projects

project_id	name
4	…
24	…

employees_projects

employee_id	project_id
1	4
40	24

Setup

```
CREATE TABLE employees(
  employee_id INT PRIMARY KEY,
  employee_name VARCHAR(50)
);
```

Table Employees

```
CREATE TABLE projects(
  project_id INT PRIMARY KEY,
  project_name VARCHAR(50)
);
```

Table Projects

Setup

```
CREATE TABLE employees_projects(
  employee_id INT,
  project_id INT,
  CONSTRAINT pk_employees_projects
  PRIMARY KEY(employee_id, project_id),
  CONSTRAINT fk_employees_projects_employees
  FOREIGN KEY(employee_id)
  REFERENCES employees(employee_id),
  CONSTRAINT fk_employees_projects_projects
  FOREIGN KEY(project_id)
  REFERENCES projects(project_id)
);
```

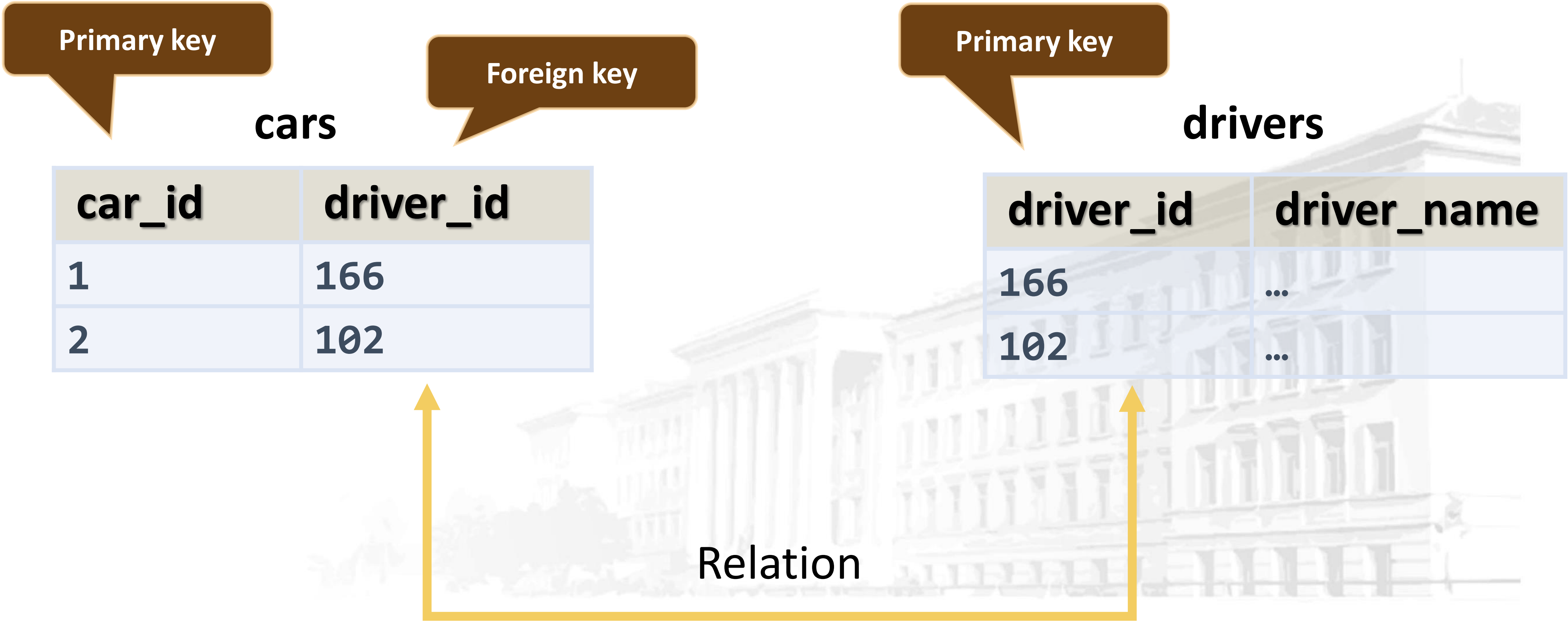
Mapping Table

Primary Key

Foreign Key

Foreign Key

One-to-One



Primary key

Foreign key

cars

car_id	driver_id
1	166
2	102

Primary key

drivers

driver_id	driver_name
166	…
102	…

Relation

Setup

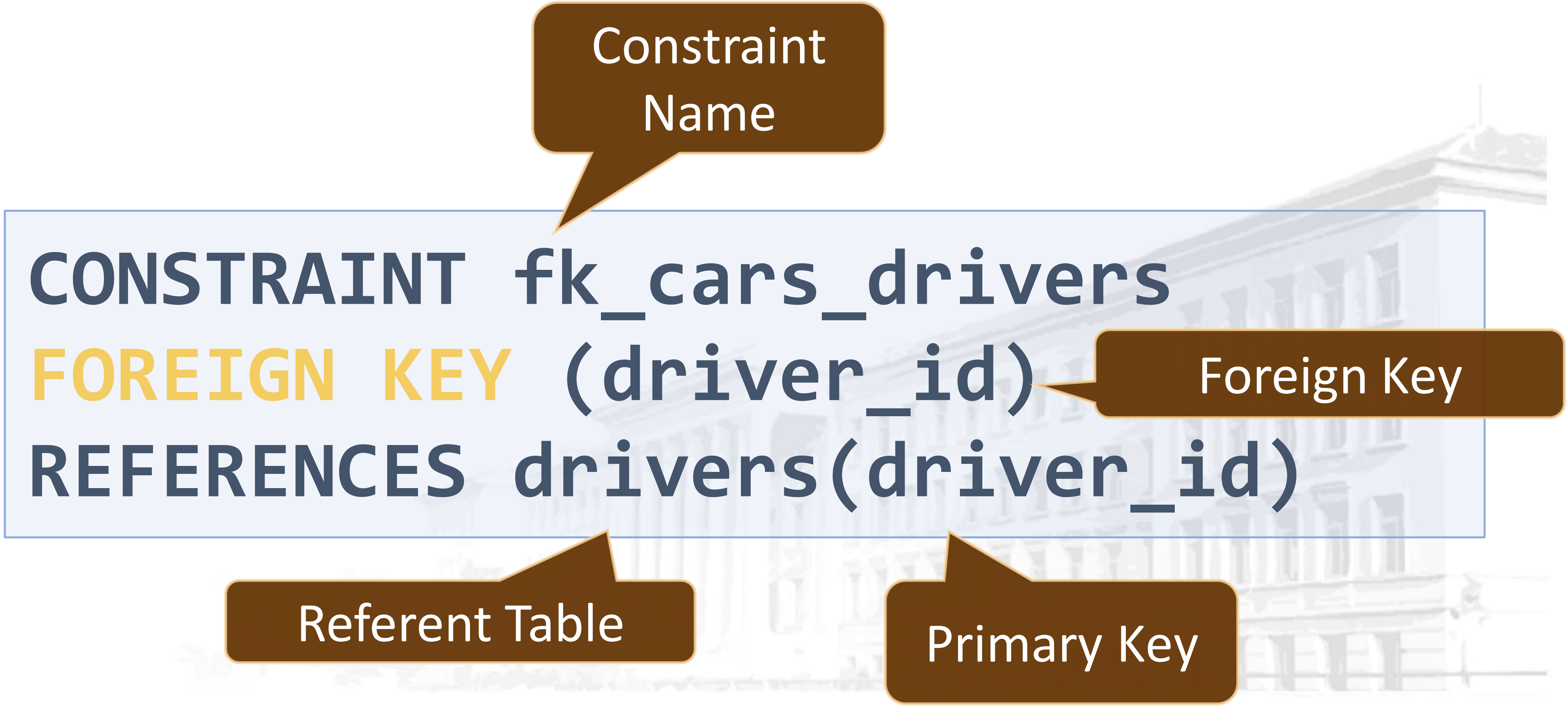
```
CREATE TABLE drivers(
  driver_id INT PRIMARY KEY,
  driver_name VARCHAR(50)
);

CREATE TABLE cars(
  car_id INT PRIMARY KEY,
  driver_id INT UNIQUE,
  CONSTRAINT fk_cars_drivers
  FOREIGN KEY (driver_id)
  REFERENCES drivers(driver_id)
);
```

Primary key

One driver per car

Foreign Key

Foreign Key



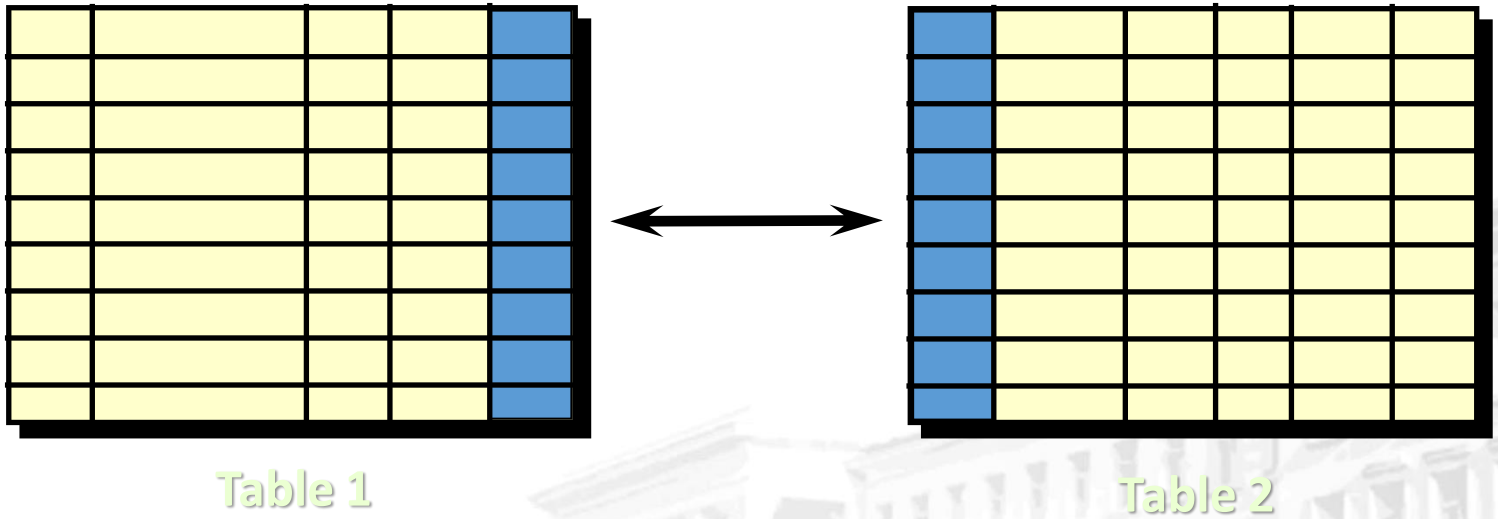
```
CONSTRAINT fk_cars_drivers
FOREIGN KEY (driver_id)
REFERENCES drivers(driver_id)
```

Constraint Name

Foreign Key

Referent Table

Primary Key



Retrieving Related Data

Using Simple JOIN statements

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:

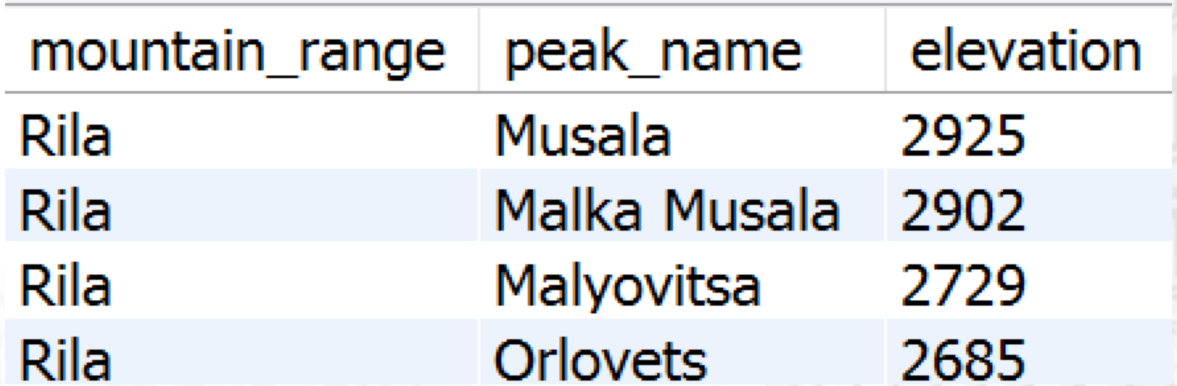
Select from Tables

```
SELECT * FROM table_a
  JOIN table_b ON
    table_b.common_column = table_a.common_column
```

Join Condition

Problem: Peaks in Rila

- Report all peaks for "Rila" mountain.
  - Report includes mountain's name, peak's name and also peak's elevation
  - Peaks should be sorted by elevation descending
  - Use database "Geography".



mountain_range	peak_name	elevation
Rila	Musala	2925
Rila	Malka Musala	2902
Rila	Malyovitsa	2729
Rila	Orlovets	2685

Solution: Peaks in Rila

Cross Table Selection

```
SELECT m.mountain_range, p.peak_name, p.elevation
  FROM peaks AS p
  JOIN mountains AS m ON m.id = p.mountain_id
  WHERE m.mountain_range = 'Rila'
  ORDER BY p.elevation DESC;
```

Join Condition

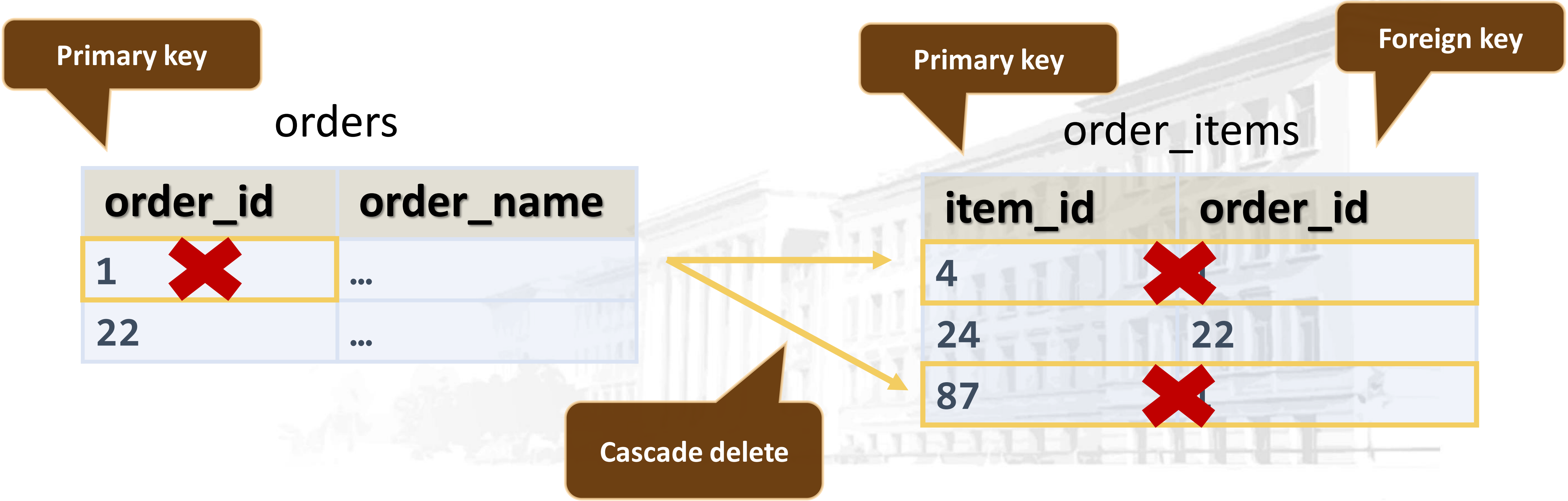
Sort

Cascade Operations

Cascade Delete/Update

Definition

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



Primary key

orders

order_id	order_name
1	…
22	…

Primary key

Foreign key

order_items

item_id	order_id
4	1
24	22
87	1

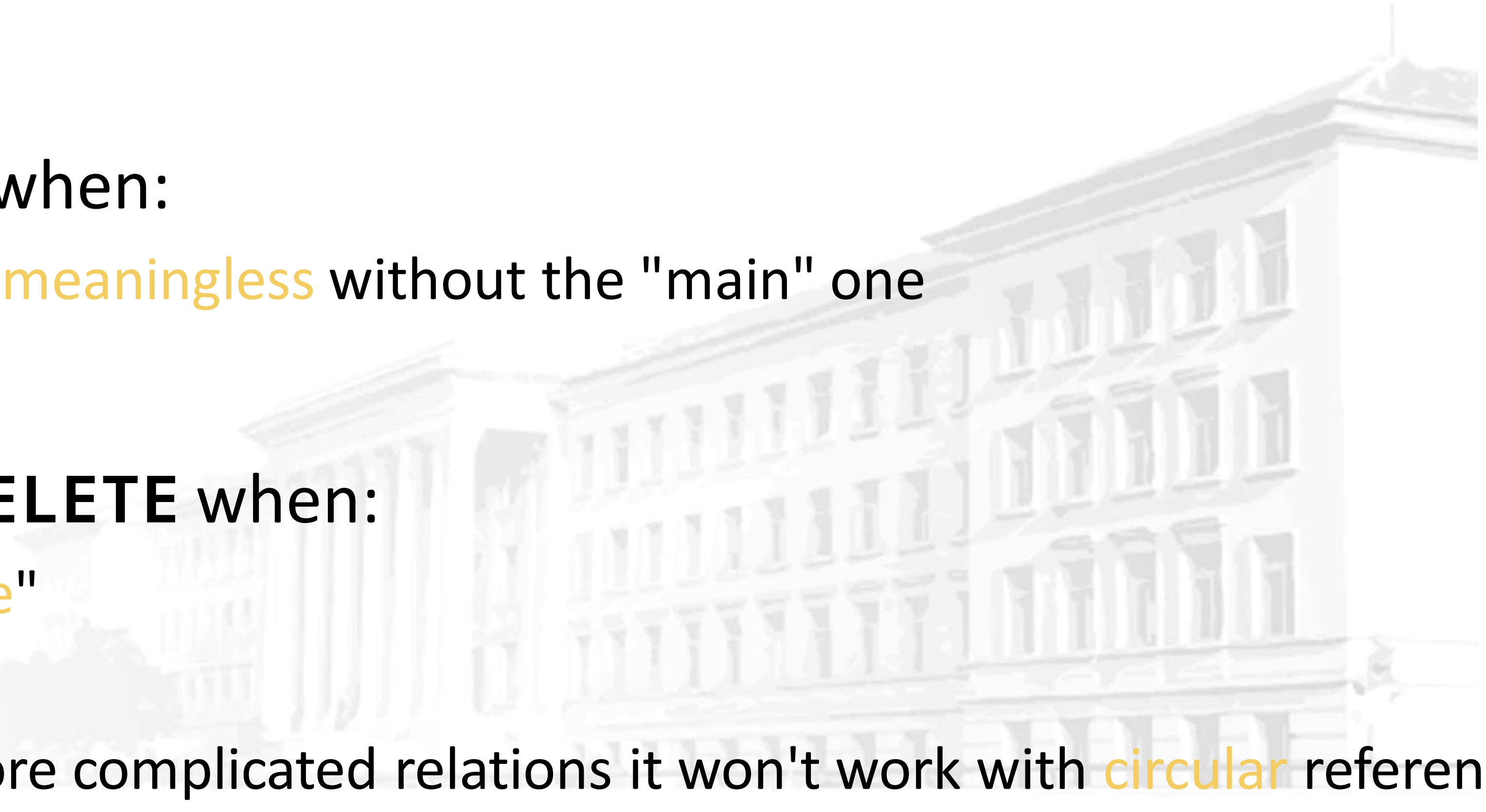
Cascade delete

CASCADE DELETE

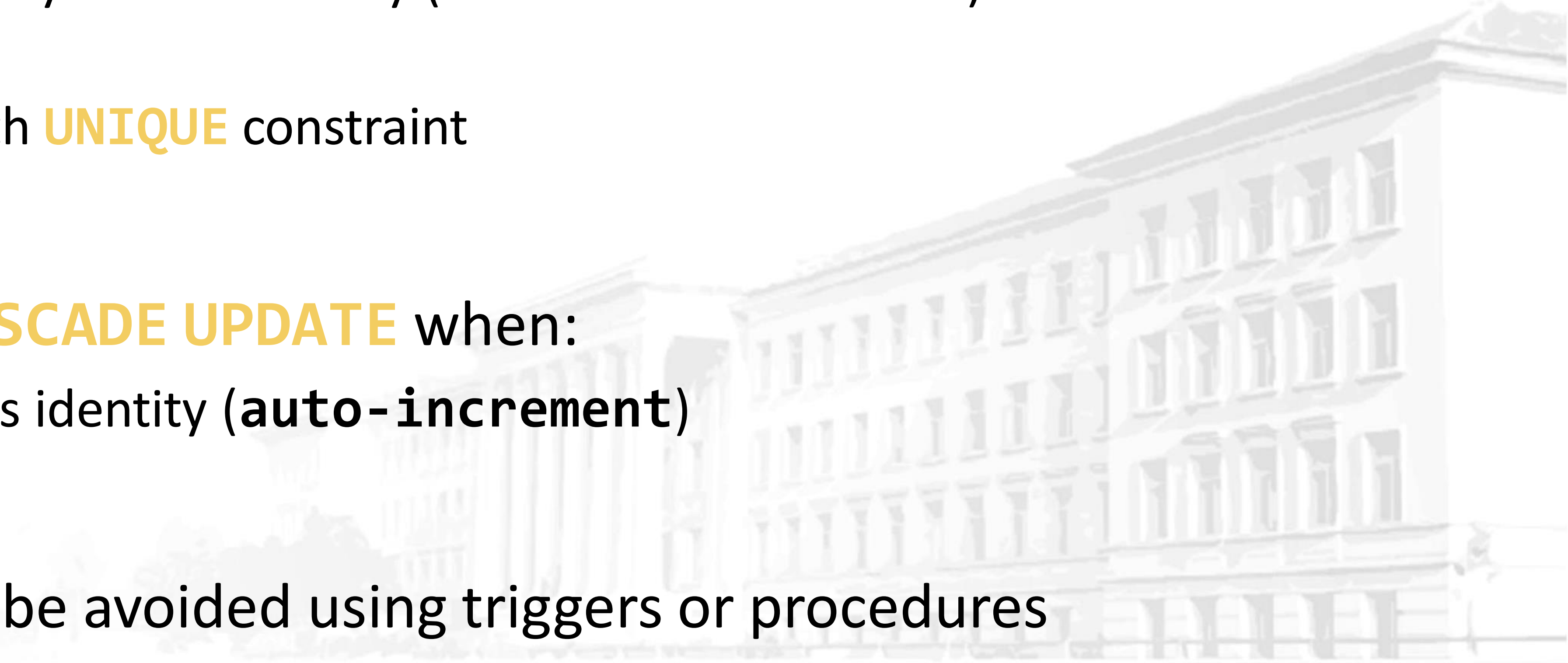
- **CASCADE** can be either DELETE or UPDATE.

- 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



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

Foreign Key Delete Cascade

```
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
);
```

Table Drivers

Table Cars

Foreign Key

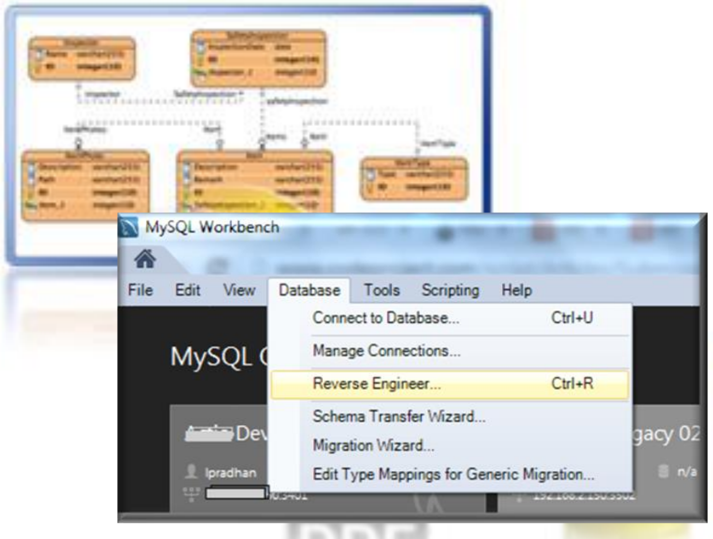
Foreign Key Update Cascade

```
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
);
```

Table Drivers

Table Cars

Foreign Key


MySQL Workbench
File
Edit
View
Database
Tools
Scripting
Help
Connect to Database... Ctrl+U
Manage Connections...
Reverse Engineer... Ctrl+R
Schema Transfer Wizard...
Migration Wizard...
Edit Type Mappings for Generic Migration...
MySQL C
Dev
lpradhan
gacy 02
n/a

E/R Diagrams

Entity / Relationship Diagrams

Relational Schema

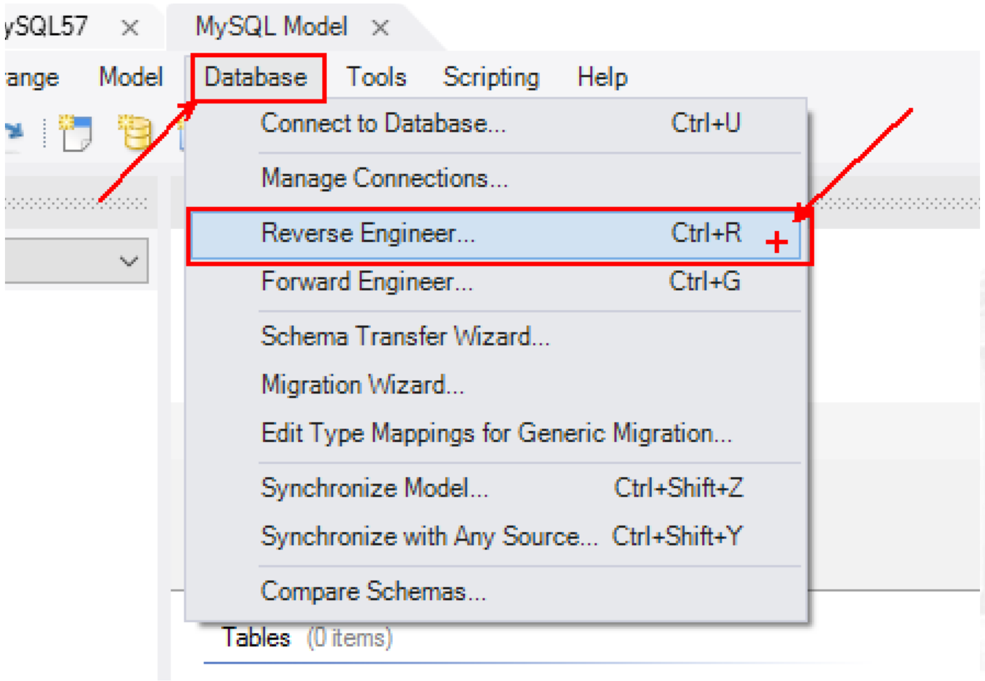
- Relational schema of a DB is the collection of:
  - The schemas of all tables
  - Relationships between the tables
  - Any other database objects (e.g. constraints)

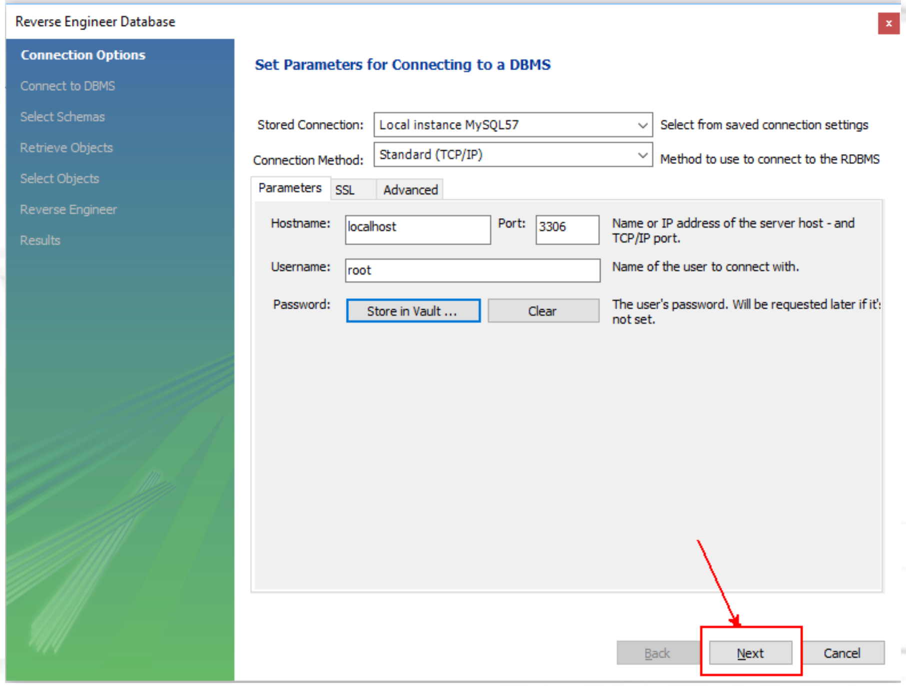
- The relational schema describes the structure of the database
  - Doesn't contain data, but metadata

- Relational schemas are graphically displayed in Entity / Relationship diagrams (E/R Diagrams)

E/R Diagram

- Click on "Database" then select "Reverse Engineer"


ySQL57
MySQL Model
ange
Model
Database
Tools
Scripting
Help
Connect to Database... Ctrl+U
Manage Connections...
Reverse Engineer... Ctrl+R
Forward Engineer... Ctrl+G
Schema Transfer Wizard...
Migration Wizard...
Edit Type Mappings for Generic Migration...
Synchronize Model... Ctrl+Shift+Z
Synchronize with Any Source... Ctrl+Shift+Y
Compare Schemas...
Tables (0 items)



Set Parameters for Connecting to a DBMS

Stored Connection: Local instance MySQL57 — Select from saved connection settings

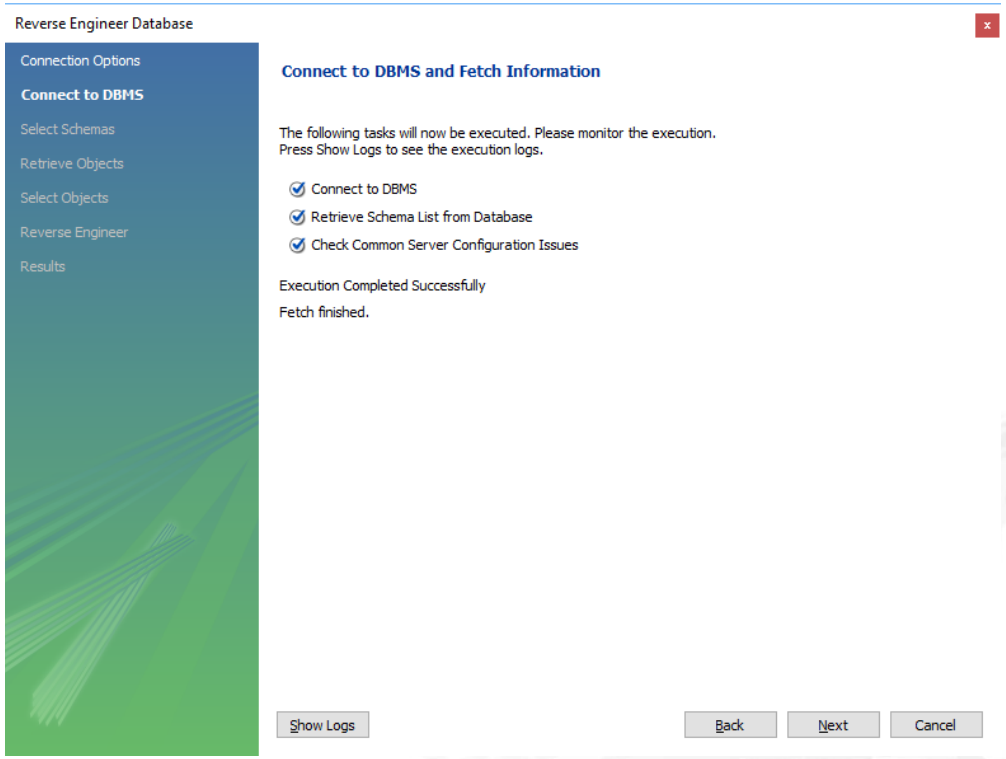
Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS

Hostname: localhost Port: 3306 — Name or IP address of the server host - and TCP/IP port.

Username: root — Name of the user to connect with.

Password: — The user's password. Will be requested later if it's not set.

E/R Diagram



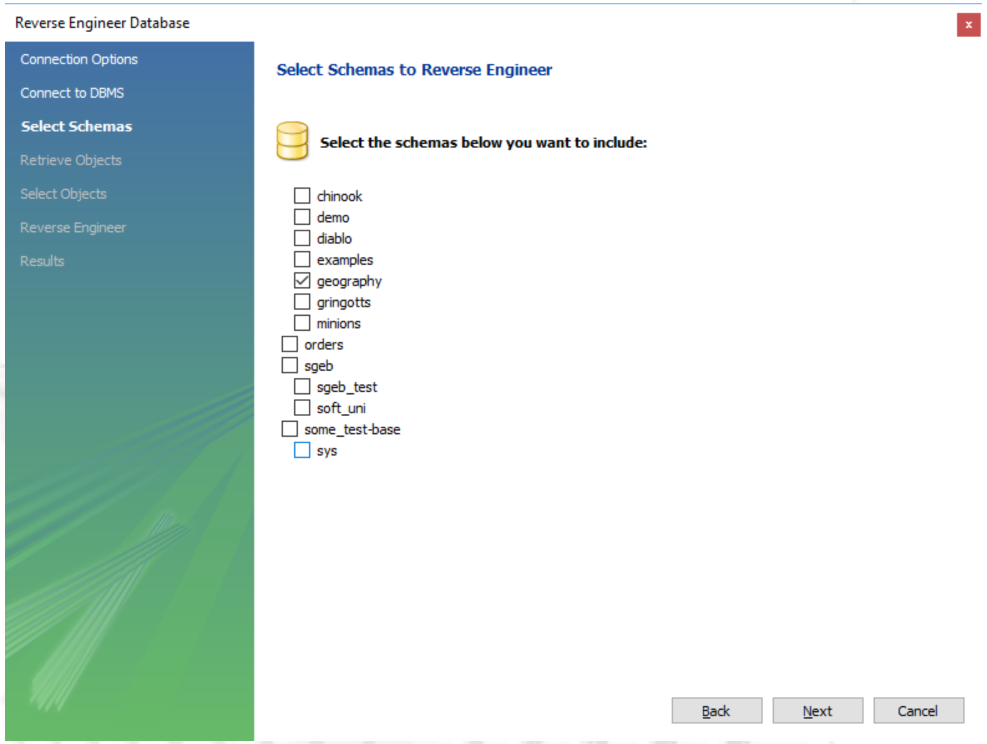
Connect to DBMS and Fetch Information

The following tasks will now be executed. Please monitor the execution.
Press Show Logs to see the execution logs.

- ☑ Connect to DBMS
- ☑ Retrieve Schema List from Database
- ☑ Check Common Server Configuration Issues

Execution Completed Successfully

Fetch finished.

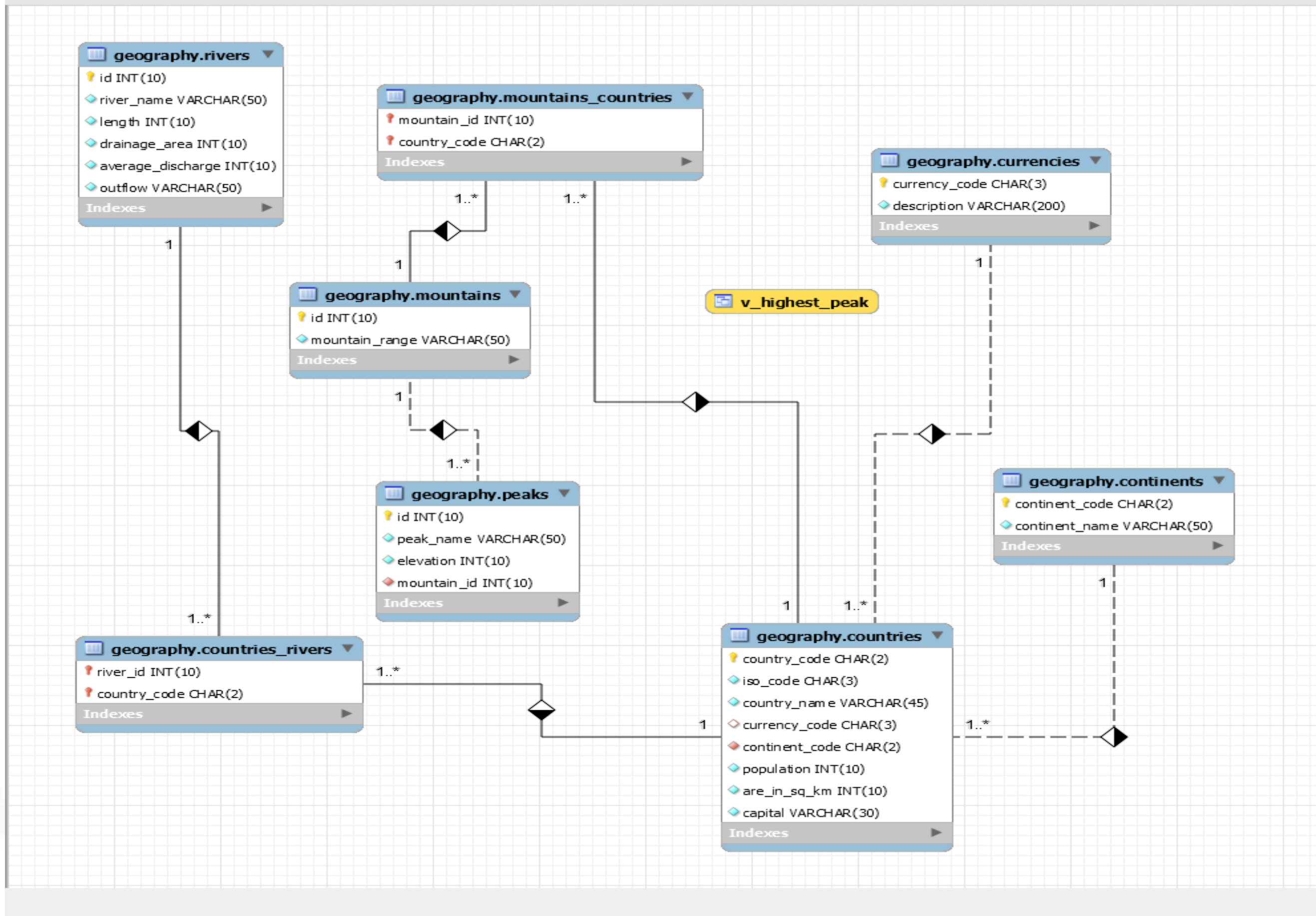


Select Schemas to Reverse Engineer

Select the schemas below you want to include:

- □ chinook
- □ demo
- □ diablo
- □ examples
- ☑ geography
- □ gringotts
- □ minions
- □ orders
- □ sgeb
- □ sgeb_test
- □ soft_uni
- □ some_test-base
- □ sys

E/R Diagram



geography.rivers
- id INT(10)
- river_name VARCHAR(50)
- length INT(10)
- drainage_area INT(10)
- average_discharge INT(10)
- outflow VARCHAR(50)
- Indexes

geography.mountains_countries
- mountain_id INT(10)
- country_code CHAR(2)
- Indexes

geography.currencies
- currency_code CHAR(3)
- description VARCHAR(200)
- Indexes

geography.mountains
- id INT(10)
- mountain_range VARCHAR(50)
- Indexes

v_highest_peak

geography.peaks
- id INT(10)
- peak_name VARCHAR(50)
- elevation INT(10)
- mountain_id INT(10)
- Indexes

geography.continents
- continent_code CHAR(2)
- continent_name VARCHAR(50)
- Indexes

geography.countries_rivers
- river_id INT(10)
- country_code CHAR(2)
- Indexes

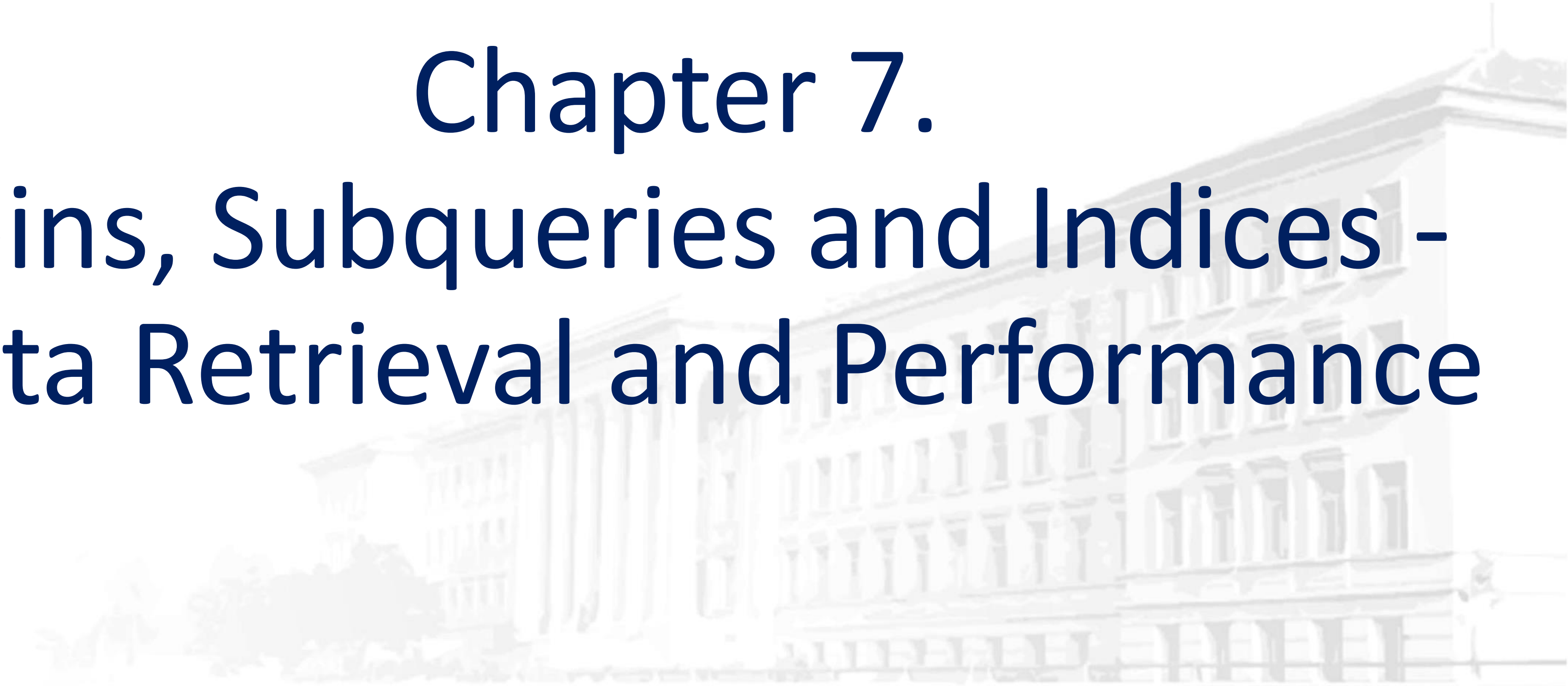
geography.countries
- country_code CHAR(2)
- iso_code CHAR(3)
- country_name VARCHAR(45)
- currency_code CHAR(3)
- continent_code CHAR(2)
- population INT(10)
- are_in_sq_km INT(10)
- capital VARCHAR(30)
- Indexes

Summary

- We design databases by specification entities and their characteristics

- Two types of relations:
  - One-to-many
  - Many-to-many

- We visualize relations via E/R diagrams



Chapter 7. Joins, Subqueries and Indices - Data Retrieval and Performance

JOINS

Gathering Data From Multiple Tables

Data from Multiple Tables

- Sometimes you need data from several tables:

Employees

employee_name	department_id
Edward	3
John	NULL

Departments

department_id	department_name
3	Sales
4	Marketing
5	Purchasing

employee_name	department_id	department_name
Edward	3	Sales

Cartesian Product

- This will produce Cartesian product:

```
SELECT last_name, name AS department_name
FROM employees, departments;
```

- The result:

last_name	department_name
Gilbert	Engineering
Brown	Engineering
…	…
Gilbert	Sales
Brown	Sales

Cartesian Product

- 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

- Formed when:
    - A join condition is omitted
    - A join condition is invalid

- To avoid, always include a valid `JOIN` condition

JOINS

- JOINS – used to collect data from two or more tables
- Types:

INNER JOIN

LEFT JOIN

RIGHT JOIN

OUTER (UNION) JOIN

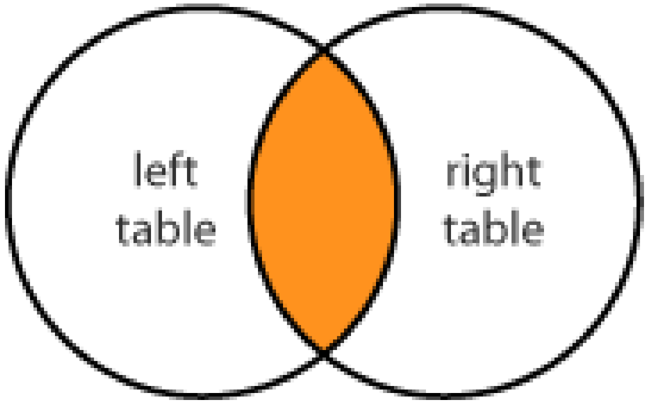
CROSS JOIN

Tables

id	name	course_id
1	Alice	1
2	Michael	1
3	Caroline	2
4	David	5
5	Emma	NULL

id	name
1	HTML5
2	CSS3
3	JavaScript
4	PHP
5	MySQL

INNER JOIN



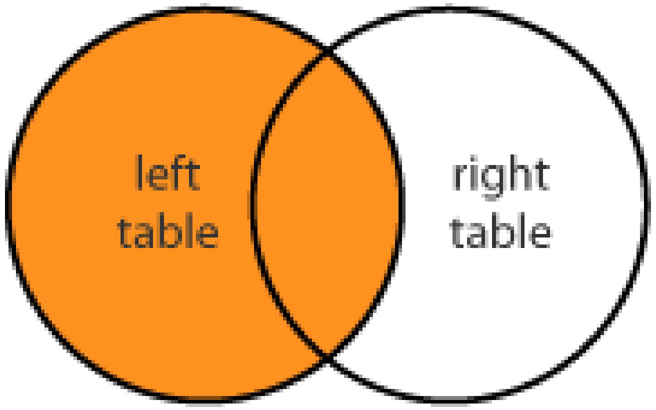
- Produces a set of records which match in both tables

```sql
SELECT students.name, courses.name
FROM students
INNER JOIN courses
ON students.course_id = courses.id
```

Join Conditions

students_name	courses_name
Alice	HTML5
Michael	HTML5
Caroline	CSS3
David	MySQL

LEFT JOIN



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

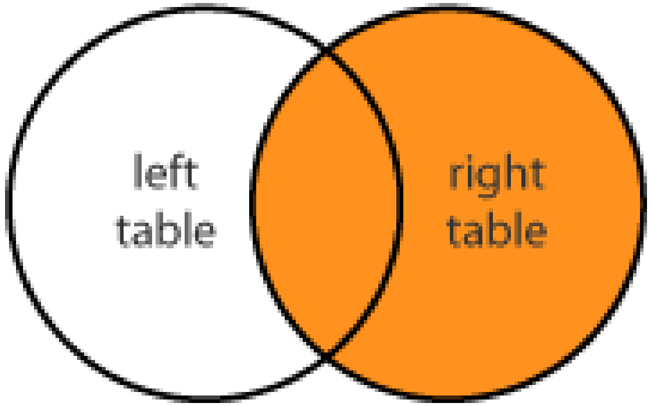
```
SELECT students.name, courses.name
FROM students
LEFT JOIN courses
ON students.course_id = courses.id
```

Join Conditions

students_name	courses_name
Alice	HTML5
Michael	HTML5
Caroline	CSS3
David	MySQL
Emma	NULL

RIGHT JOIN

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

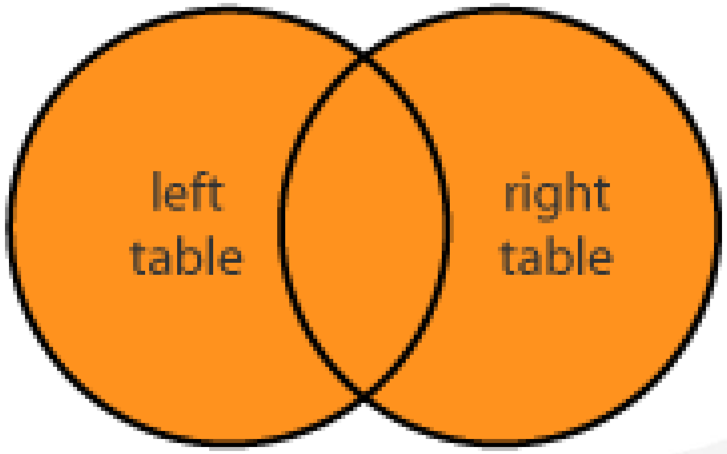


```
SELECT students.name, courses.name
FROM students
RIGHT JOIN courses
ON students.course_id = courses.id
```

Join Conditions

students_name	courses_name
Alice	HTML5
Michael	HTML5
Caroline	CSS3
NULL	JavaScript
NULL	PHP
David	MySQL

OUTER (FULL JOIN)



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

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
```

students_name	courses_name
Alice	HTML5
Michael	HTML5
Caroline	CSS3
David	MySQL
Emma	NULL
NULL	JavaScript
NULL	PHP

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

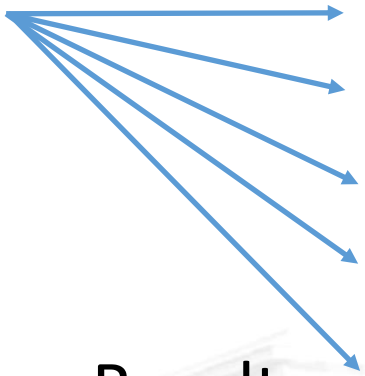
```
SELECT * FROM courses AS c
 CROSS JOIN students AS s;
```

No Join Conditions

Cross Join

Courses

id	name
1	HTML5
2	CSS3
3	JavaScript
4	PHP
5	MySQL



Students

id	name	course_id
1	Alice	1
2	Michael	1
3	Caroline	2
4	David	5
5	Emma	NULL

Result

course_id	course_name	student_id	student_name
1	HTML5	1	Alice
1	HTML5	2	Michael
1	HTML5	3	Caroline
…	…	…	…

Join Overview

employee_name	department_id
Sally	13
John	10
Michael	22
Bob	11
Robin	7
Jessica	15

department_id	department_name
7	Executive
8	Sales
10	Marketing
12	HR
18	Accounting
22	Engineering

Relation

Join Overview: INNER JOIN

employee_name	department_id
Sally	13
John	10
Michael	22
Bob	11
Robin	7
Jessica	15

department_id	department_name
7	Executive
8	Sales
10	Marketing
12	HR
18	Accounting
22	Engineering

Join Overview: LEFT JOIN

employee_name	department_id
Sally	13
John	10
Michael	22
Bob	11
Robin	7
Jessica	15

department_id	department_name
7	Executive
8	Sales
10	Marketing
12	HR
15	Shipping And Receiving
18	Accounting
22	Engineering
NULL	NULL

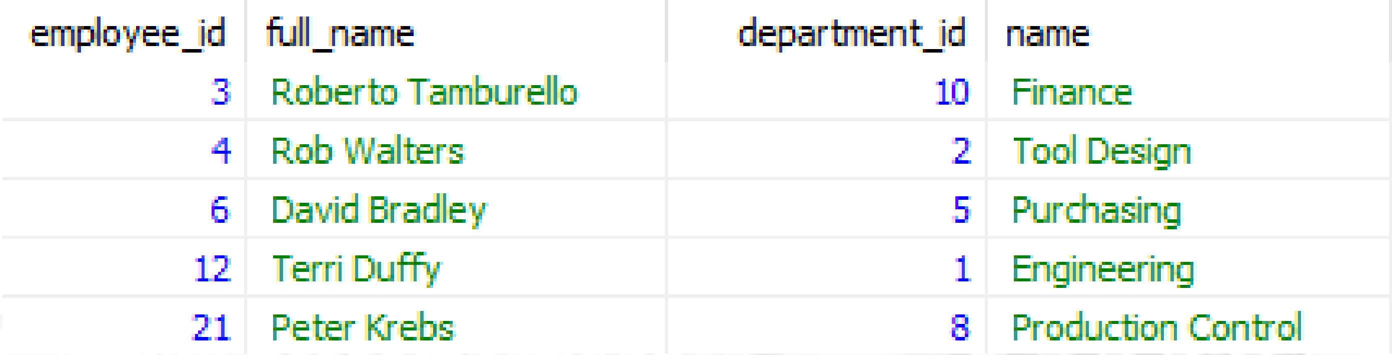
Join Overview: RIGHT JOIN

employee_name	department_id
Sally	13
John	10
Michael	22
Bob	11
Robin	7
Jessica	15

department_id	department_name
7	Executive
8	Sales
10	Marketing
12	HR
18	Accounting
22	Engineering

Problem: Managers

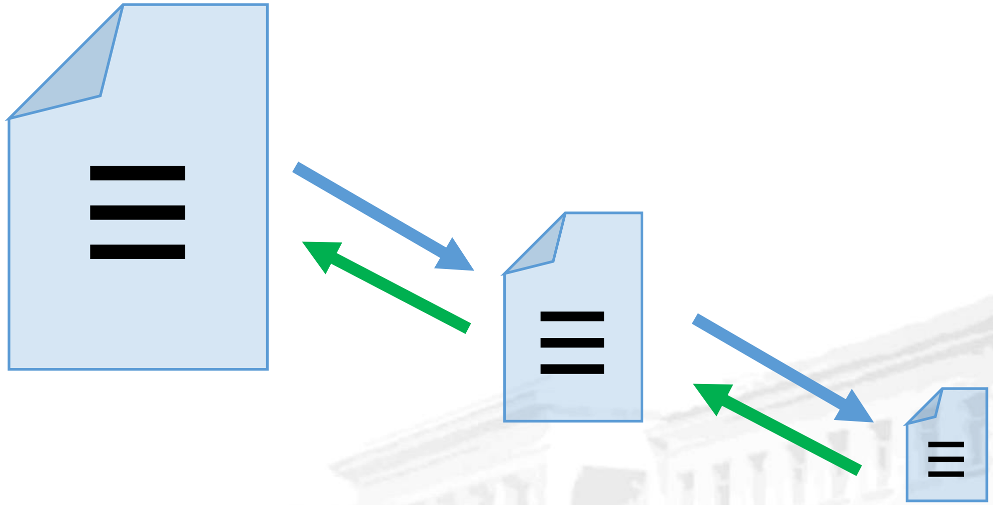
- Get information about the first 5 managers in the “uni_ruse" database
  - **`id`**
  - **`full_name`**
  - **`department_id`**
  - **`department_name`**



employee_id	full_name	department_id	name
3	Roberto Tamburello	10	Finance
4	Rob Walters	2	Tool Design
6	David Bradley	5	Purchasing
12	Terri Duffy	1	Engineering
21	Peter Krebs	8	Production Control

Solution: Managers

```
SELECT e.employee_id, CONCAT(first_name, " ",
last_name) AS `full_name`, d.department_id, d.name
FROM employees AS e
RIGHT JOIN departments AS d
ON d.manager_id = e.employee_id
ORDER BY e.employee_id LIMIT 5;
```



Subqueries

Query Manipulation on Multiple Levels

Subqueries

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

```
SELECT * FROM students
WHERE course_id = 1;
```

id	name	course_id
1	Alice	1
2	Michael	1

Subquery

Problem: Higher Salary

- Count the number of employees who receive salary, higher than the average
  - Use "uni_ruse" database

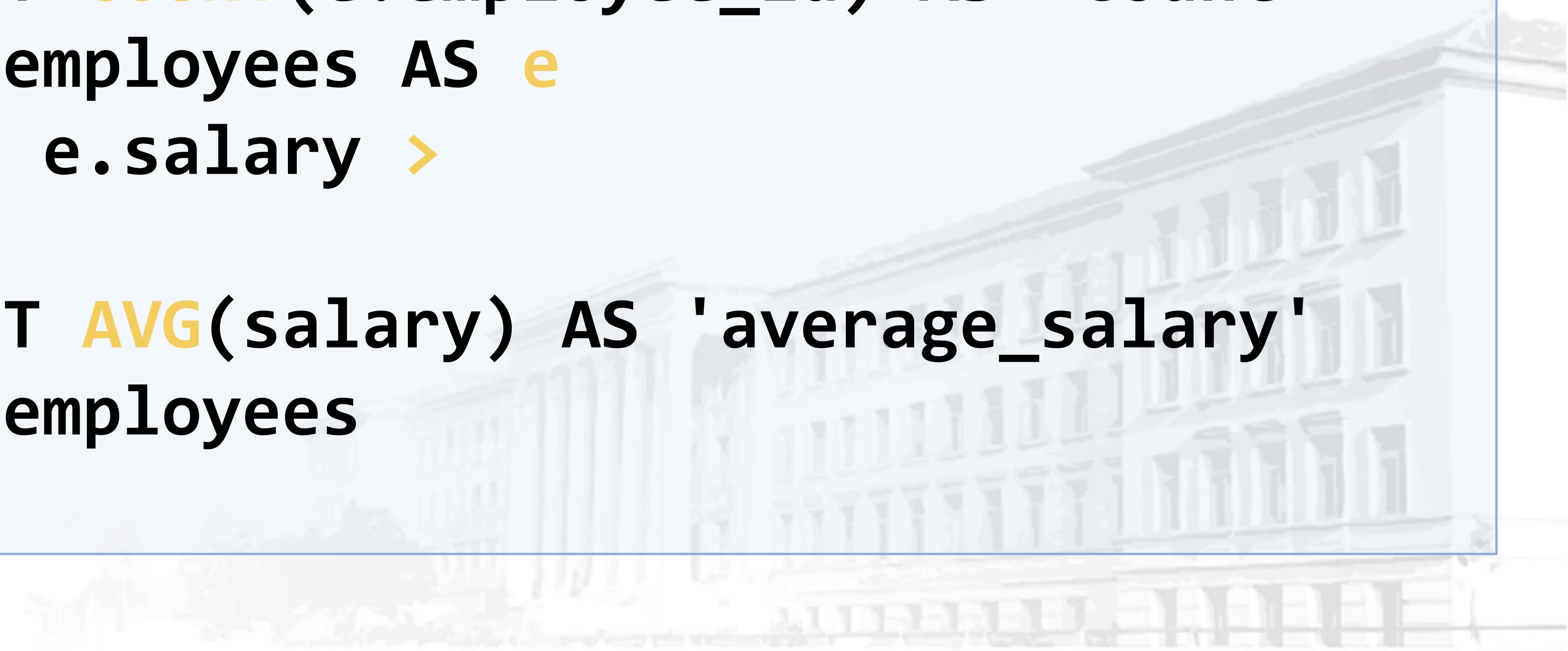
employee_id	first_name	last_name	…
216	Mike	Seamans	…
178	Barbara	Moreland	…
…	…	…	…

count
88

Table "employees"

Solution: Higher Salary

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



Indices

Clustered and Non-Clustered Indices

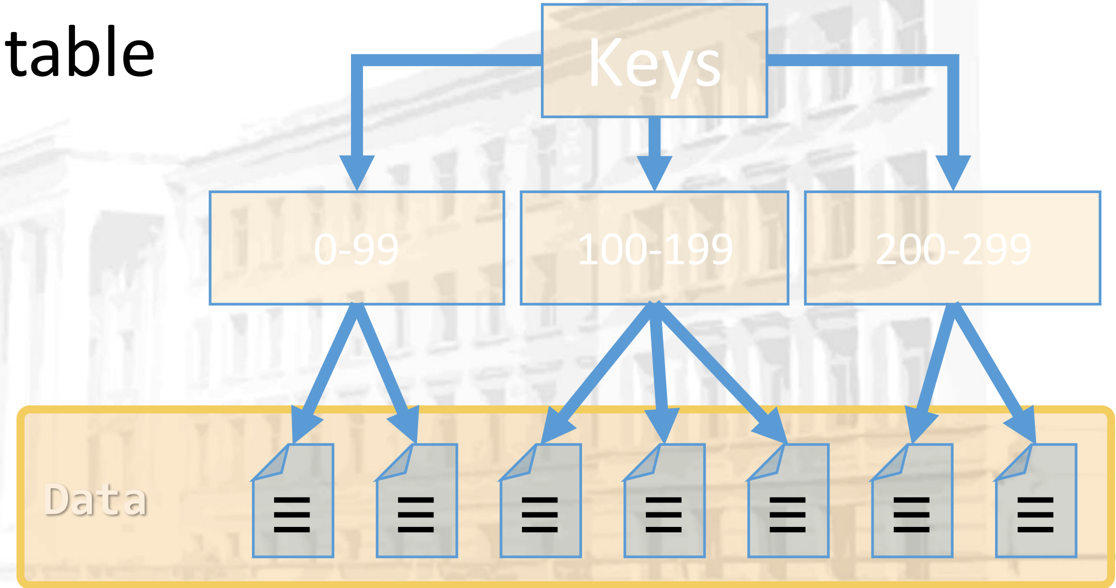
Indices

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

Clustered Indices

- Clustered index determine the order of data
  - Very useful for fast execution of WHERE, ORDER BY and GROUP BY clauses

- Maximum 1 clustered index per table
  - If a table has no clustered index, its data rows are stored in an unordered structure (heap)

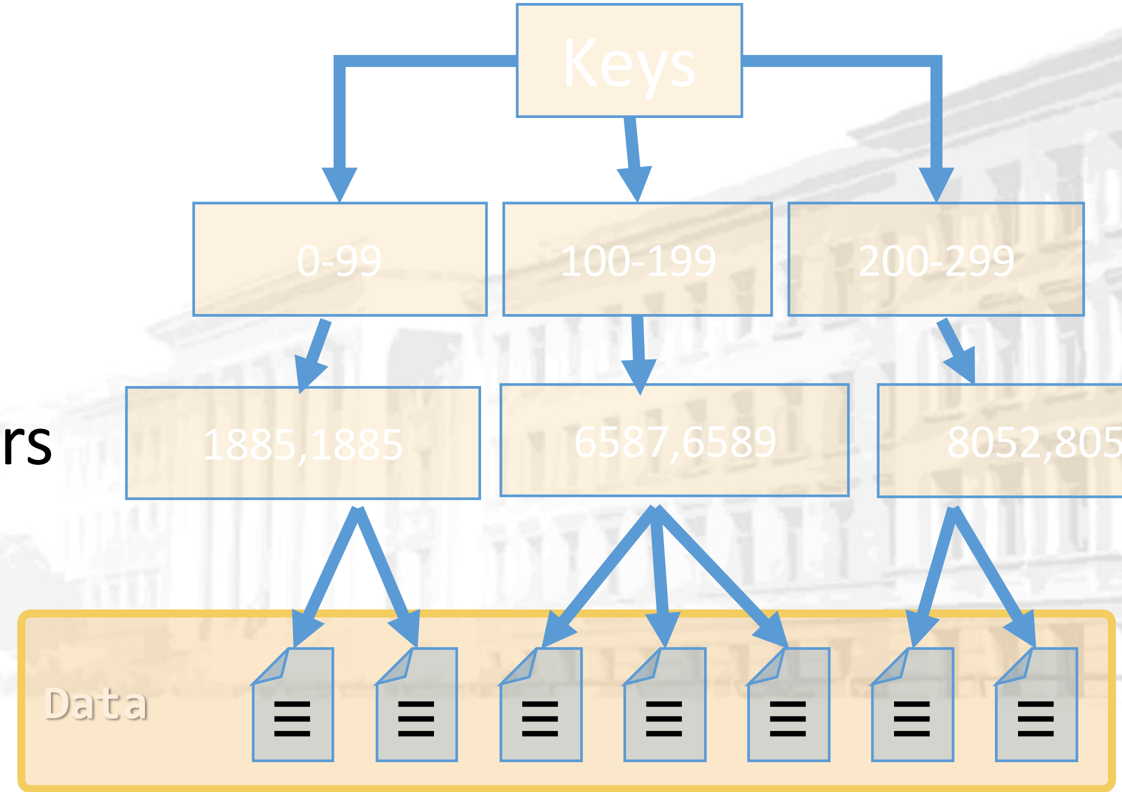


Non-Clustered Indices

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

- Maintained in a separate structure in the DB

Pointers



Indices Syntax

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

Table Name

Columns

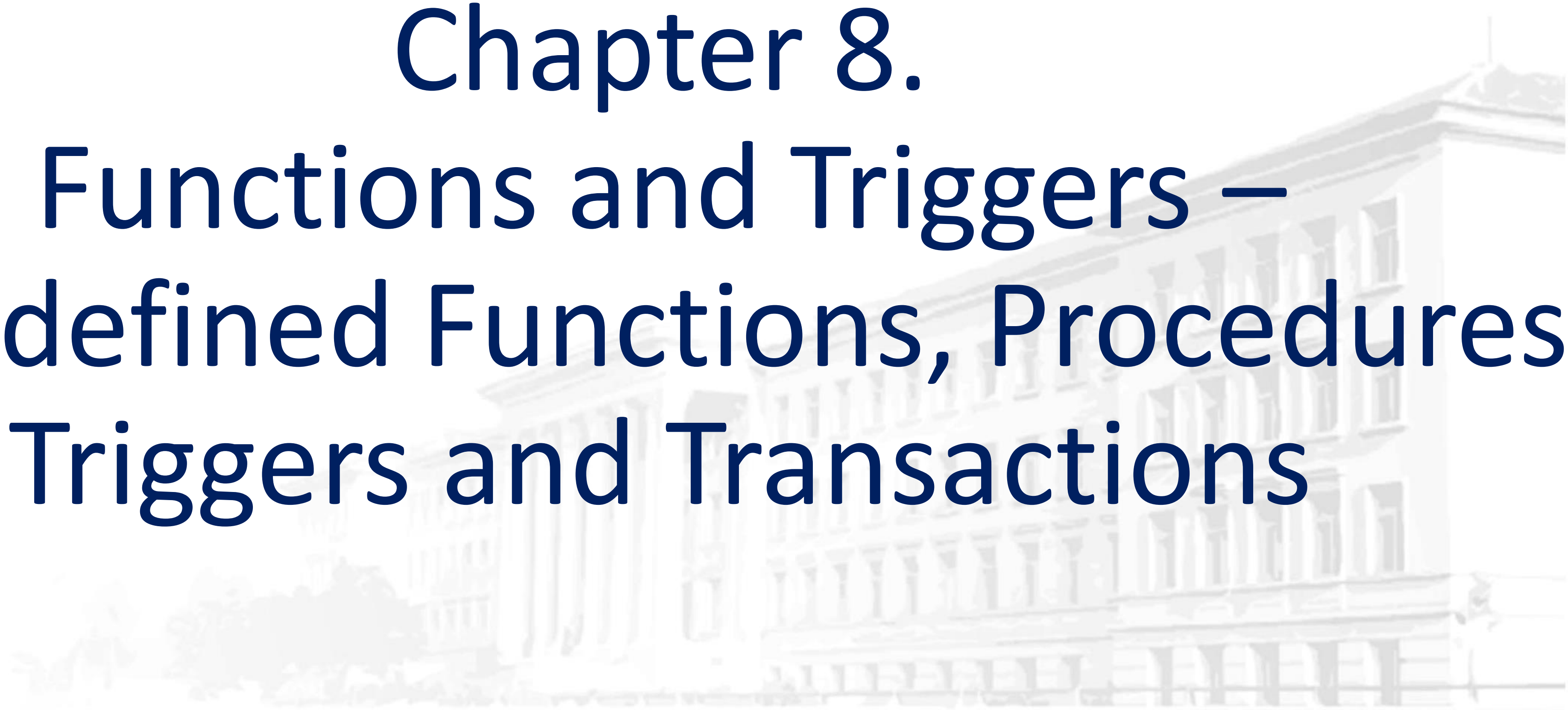
Summary

- Joins

```
SELECT * FROM employees AS e
  JOIN departments AS d ON
d.department_id = e.department_id
```

- Subqueries are used to nest queries

- Indices improve SQL search performance if used properly

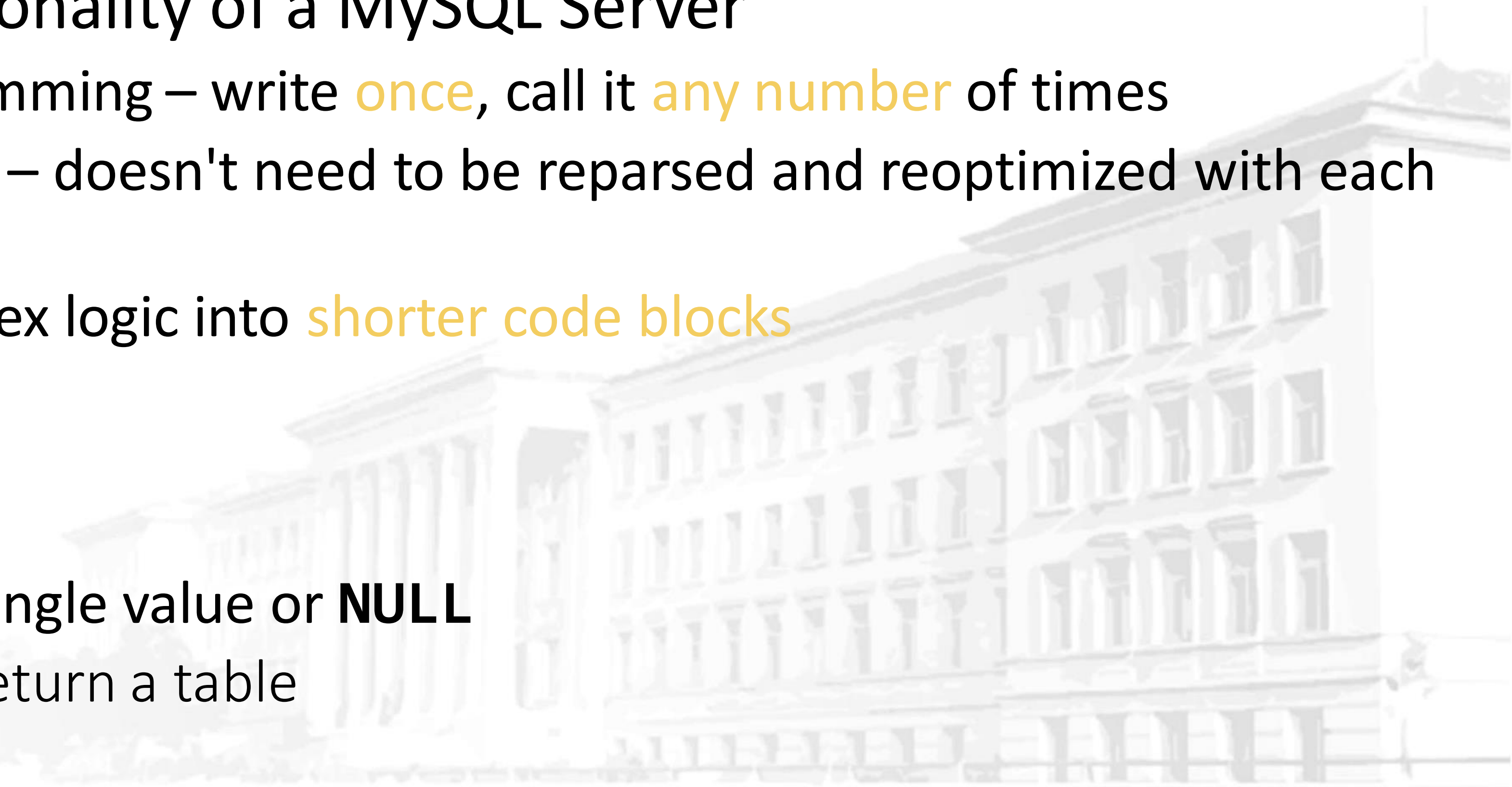


Chapter 8.
Functions and Triggers –
User-defined Functions, Procedures, Triggers and Transactions

User-Defined Functions

Encapsulating custom logic

User-Defined Functions



- Extend the functionality of a MySQL Server
  - Modular 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

Problem: Count Employees by Town

- Write a function `ufn_count_employees_by_town(town_name)` that:
  - Accepts town name as parameter
  - Returns the count of employees in the database who live in that town

Solution: Count Employees by Town

Function Name

```
CREATE FUNCTION ufn_count_employees_by_town(town_name VARCHAR(20))
RETURNS DOUBLE
BEGIN
        DECLARE e_count DOUBLE;
        SET e_count := (SELECT COUNT(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;
END
```

Function Logic

Result: Count Employees by Town

- Examples of expected output:

Function Call

Employees count

```sql
SELECT ufn_count_employees_by_town('Sofia');
```

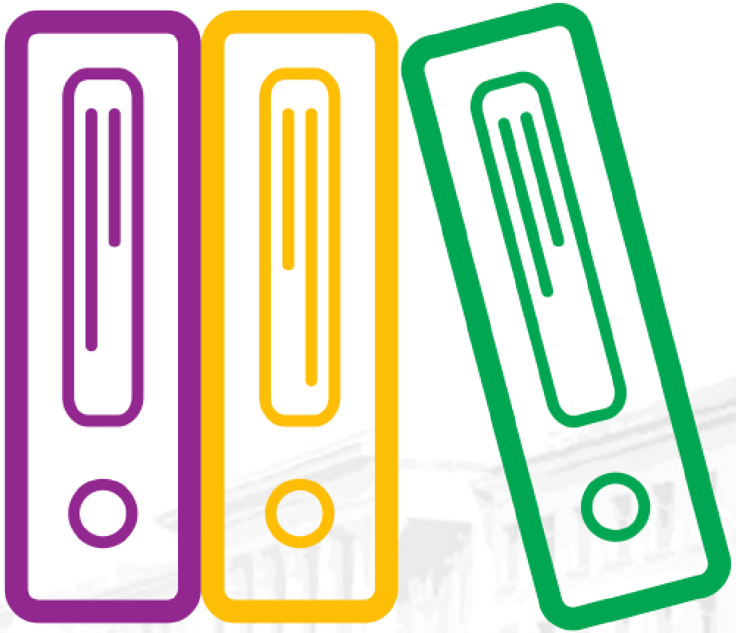
→ 3

```sql
SELECT ufn_count_employees_by_town('Berlin');
```

→ 1

```sql
SELECT ufn_count_employees_by_town(NULL);
```

→ 0



Stored Procedures

Sets of queries stored on DB Server

Stored Procedures

- 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 your database server
  - Separate data access routines from the business logic

- Accessed by programs using different platforms and API's

Creating Stored Procedures

- CREATE PROCEDURE
- Example:

Procedure Name

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

Procedure Logic

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,…)
```

Parameterized Stored Procedures – Example

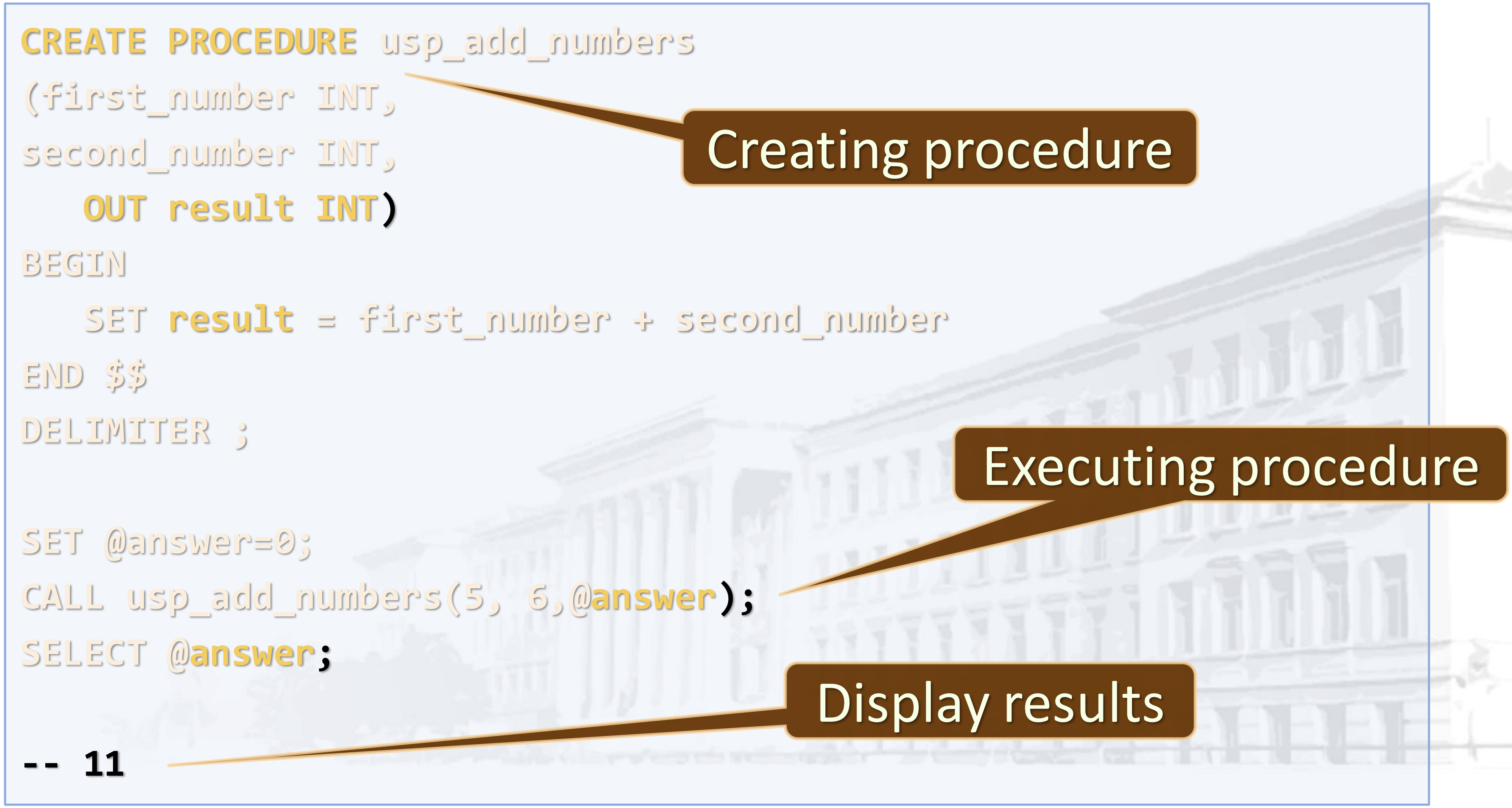
```
DELIMITER $$
CREATE PROCEDURE usp_select_employees_by_seniority(min_years_at_work INT)
BEGIN
  SELECT first_name, last_name, hire_date,
    ROUND(DATEDIFF(NOW(),DATE(hire_date)) / 365.25,0) AS 'years'
  FROM employees
  WHERE ROUND(DATEDIFF(NOW(),DATE(hire_date)) / 365.25,0) > min_years_at_work
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

CALL usp_select_employees_by_seniority(15);
```

Procedure Name

Procedure Logic

Usage

Returning Values



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

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

-- 11
```

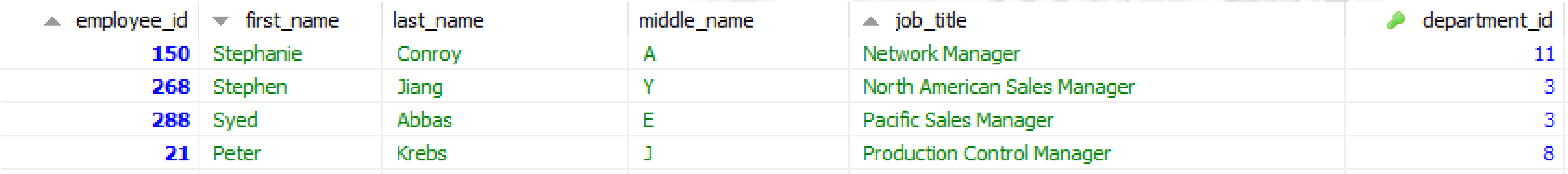
Creating procedure

Executing procedure

Display results

Problem: Employees Promotion

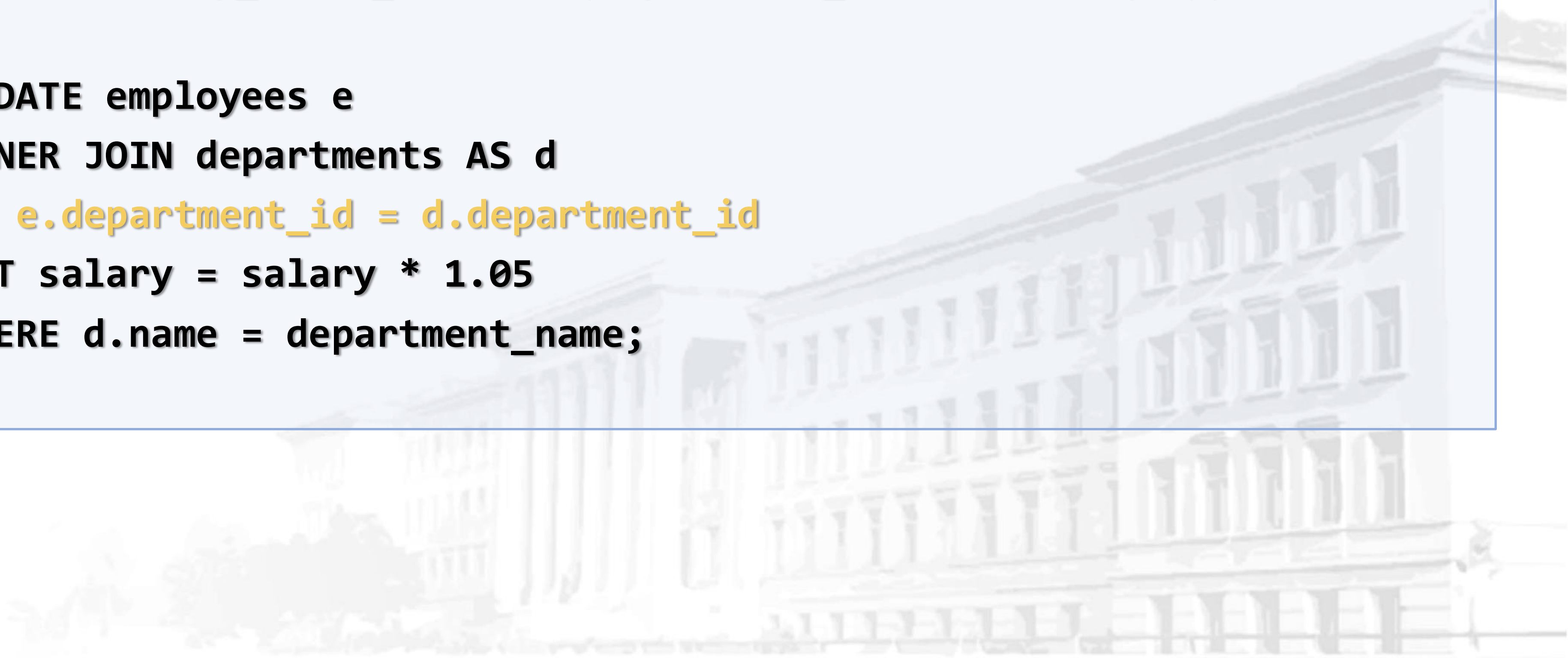
- Write a stored procedure that raises employees salaries by department name (as parameter) by 5%
  - Use uni_ruse database



employee_id	first_name	last_name	middle_name	job_title	department_id
150	Stephanie	Conroy	A	Network Manager	11
268	Stephen	Jiang	Y	North American Sales Manager	3
288	Syed	Abbas	E	Pacific Sales Manager	3
21	Peter	Krebs	J	Production Control Manager	8

Solution: Employees Promotion

```
CREATE PROCEDURE usp_raise_salaries(department_name varchar(50))
BEGIN
       UPDATE employees e
       INNER JOIN departments AS d
       ON e.department_id = d.department_id
       SET salary = salary * 1.05
       WHERE d.name = department_name;
END
```



Result: Employees Promotion

- Procedure result for 'Sales' department:

```
CALL usp_raise_salaries('Sales');
```

Data before procedure call:

employee_id	salary
268	48 100.00
273	72 100.00
…	…

Data after procedure call:

employee_id	salary
268	50 505.00
273	75 705.00
…	…

What is a Transaction?

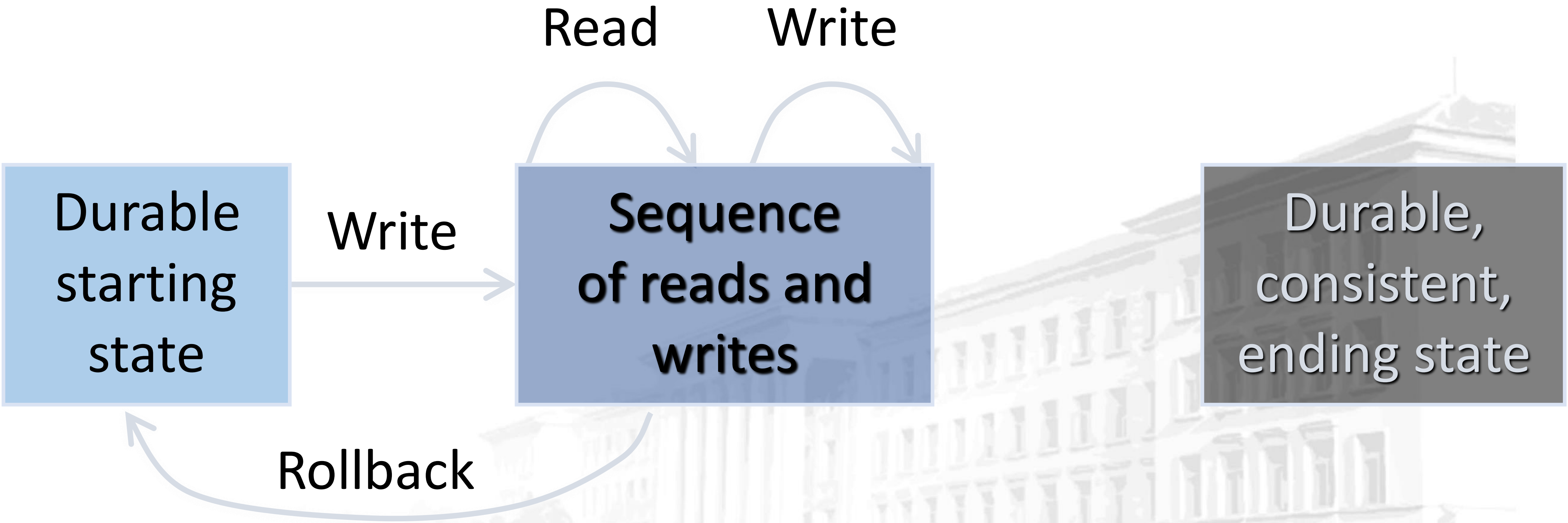
Executing operations as a whole

Transactions

- Transaction is a sequence of actions (database operations) executed as a whole
  - Either all of them complete successfully or none of the them

- Example of transaction
  - A bank transfer from one account into another (withdrawal + deposit)
    - If either the withdrawal or the deposit fails the whole operation is cancelled

Transactions: Lifecycle (Rollback)



Read

Write

Durable starting state

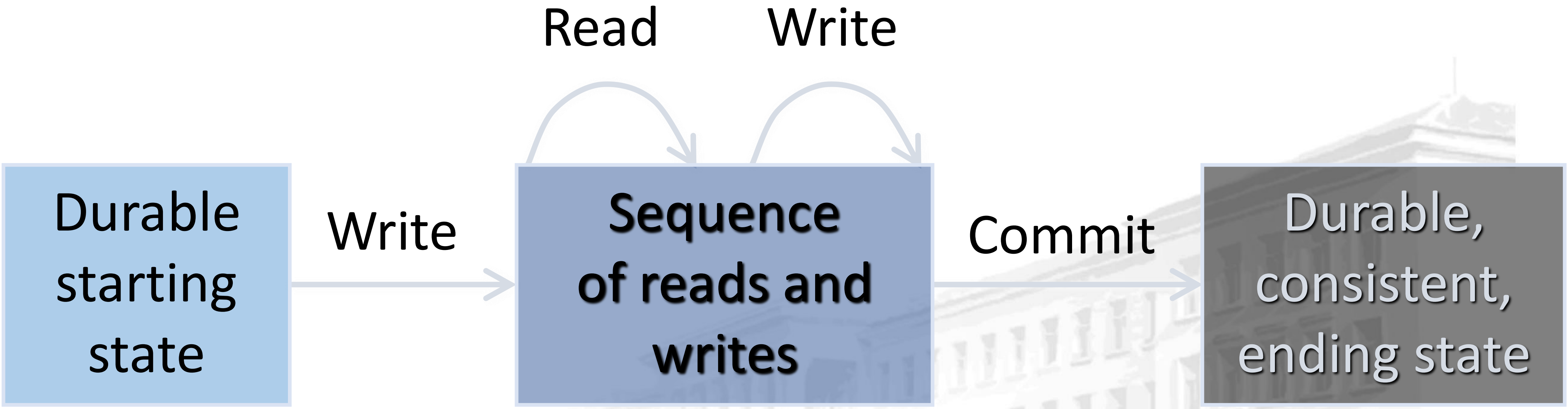
Write

Sequence of reads and writes

Durable, consistent, ending state

Rollback

Transactions: Lifecycle (Commit)



Read

Write

Durable starting state

Write

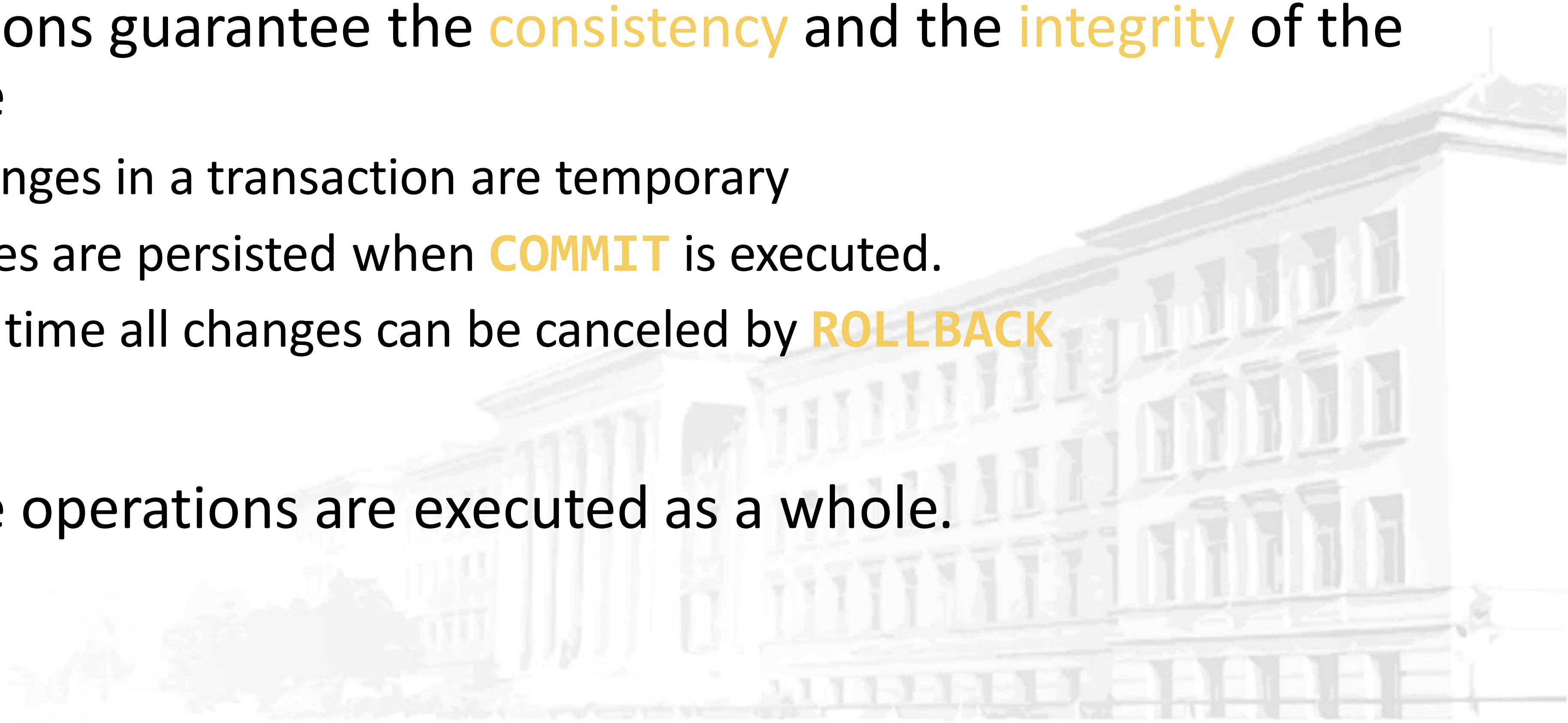
Sequence of reads and writes

Commit

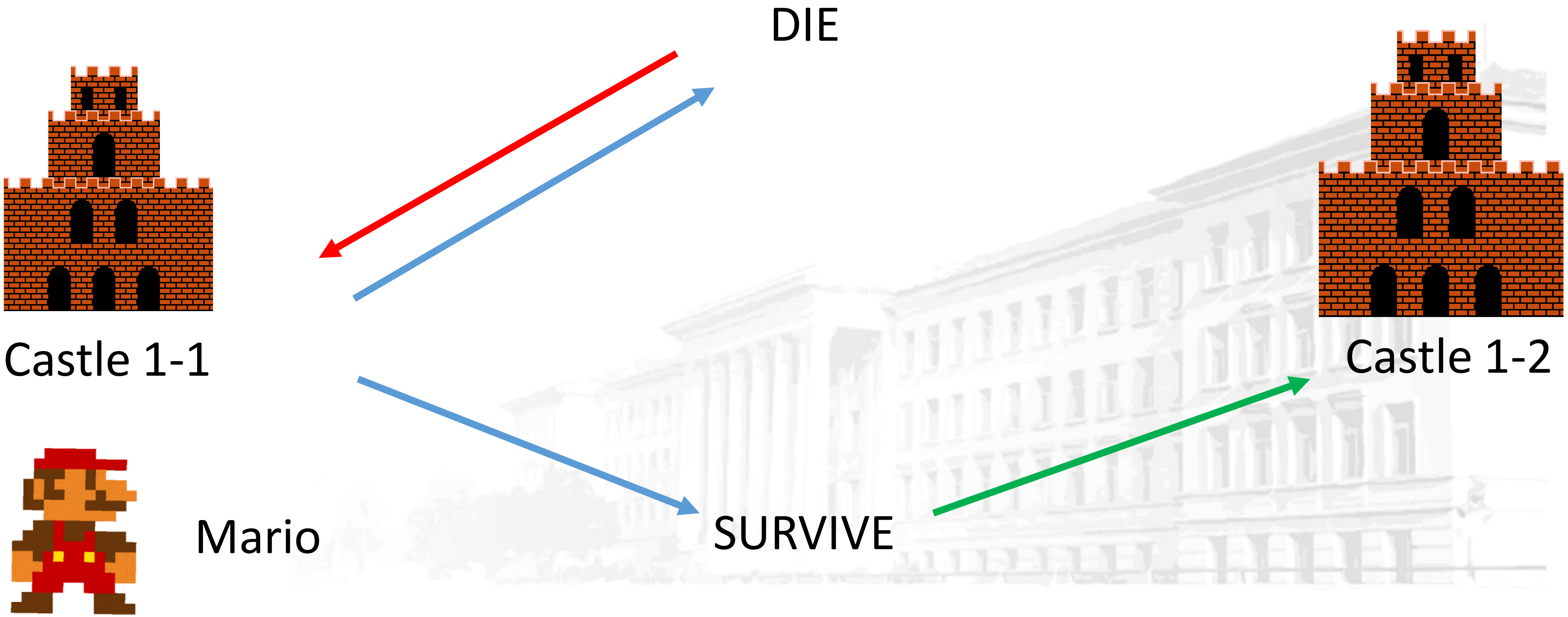
Durable, consistent, ending state

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



Checkpoints in games



DIE

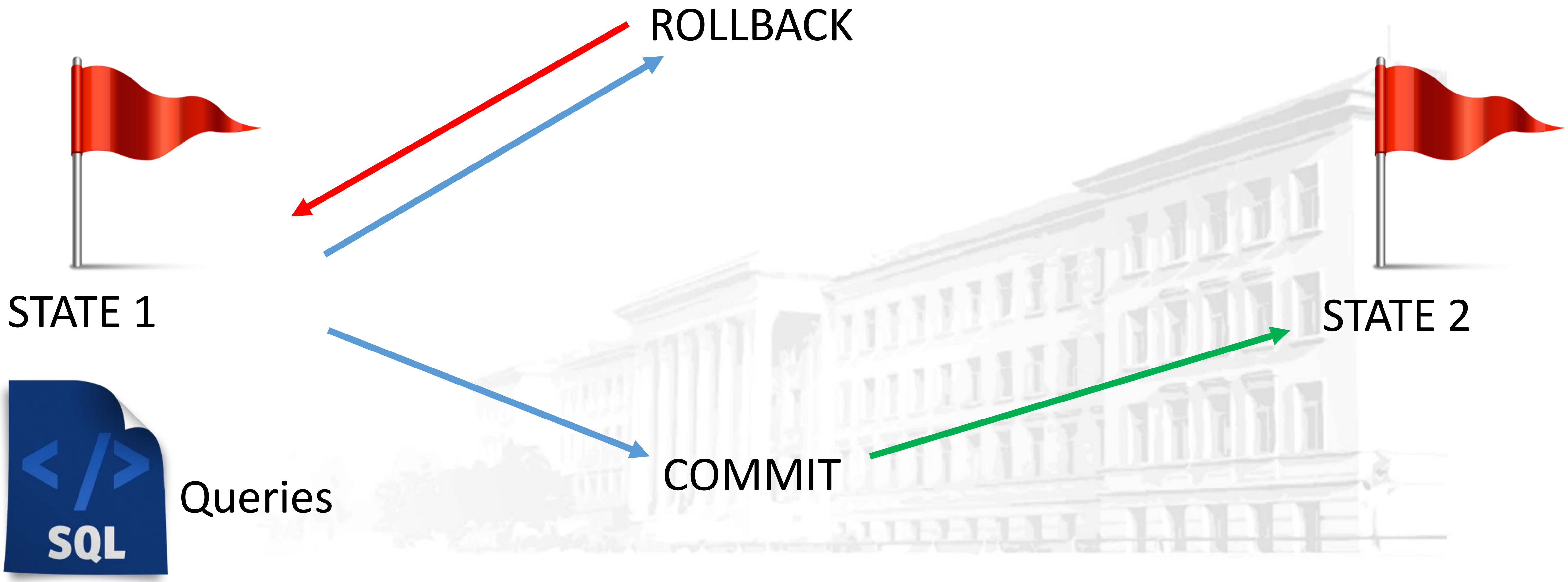
Castle 1-1

Castle 1-2

Mario

SURVIVE

What are Transactions?



ROLLBACK

STATE 1

STATE 2

COMMIT

Queries

Problem: Employees Promotion By ID

- Write a transaction that raises an employee's salary by id only if the employee exists in the database
  - If not, no changes should be made
  - Use uni_ruse database

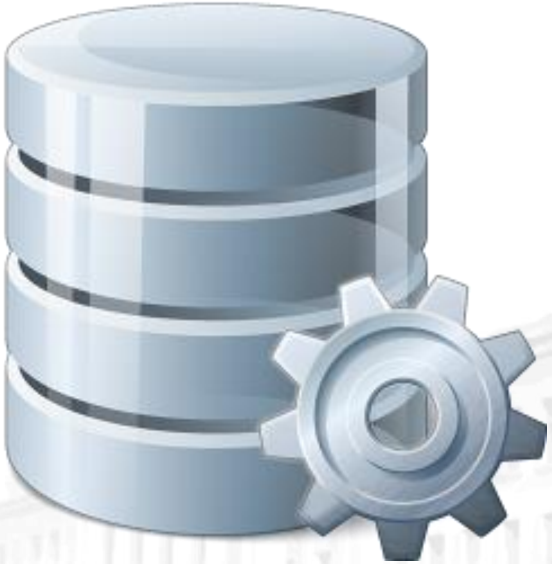
Solution: Employees Promotion

```
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
```

Transactions Properties

- Modern DBMS servers have built-in transaction support
  - Implement "ACID" transactions
  - E.g. Oracle, MySQL, MS SQL Server, …

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



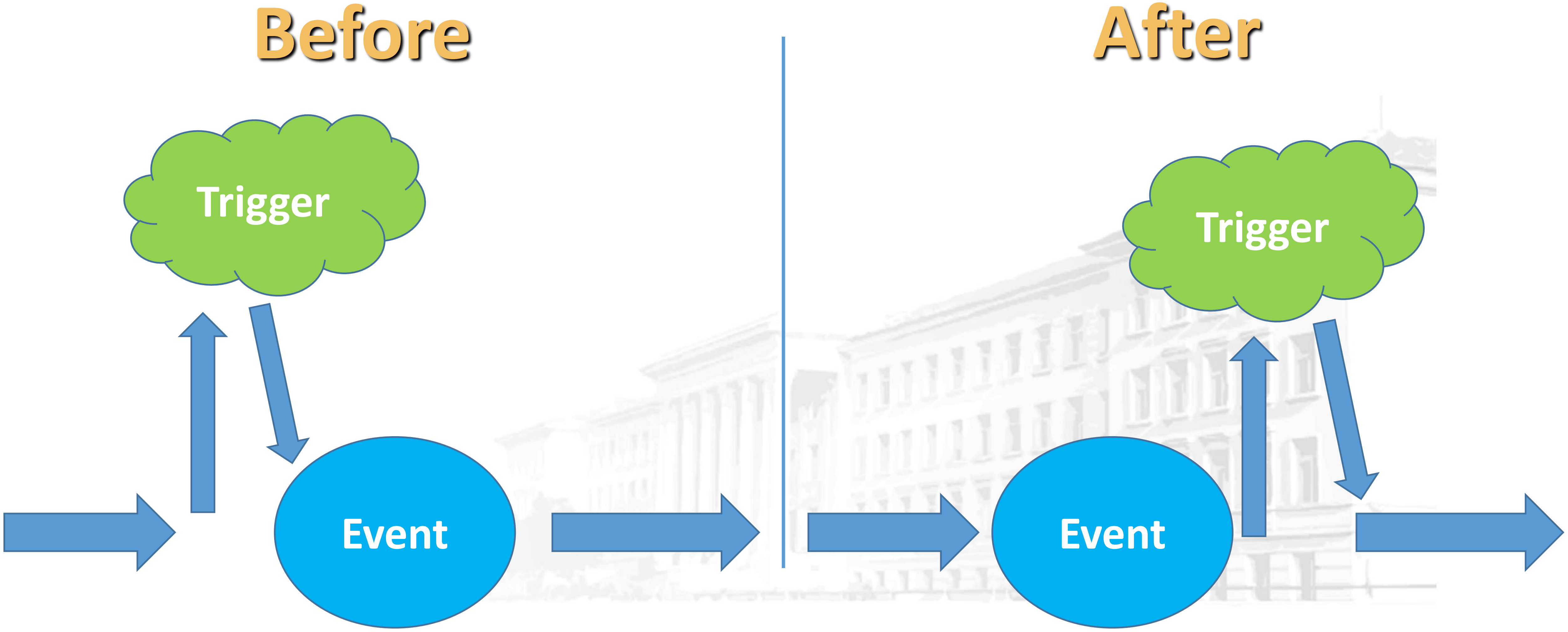
Triggers

Maintaining the integrity of the data

What Are Triggers?

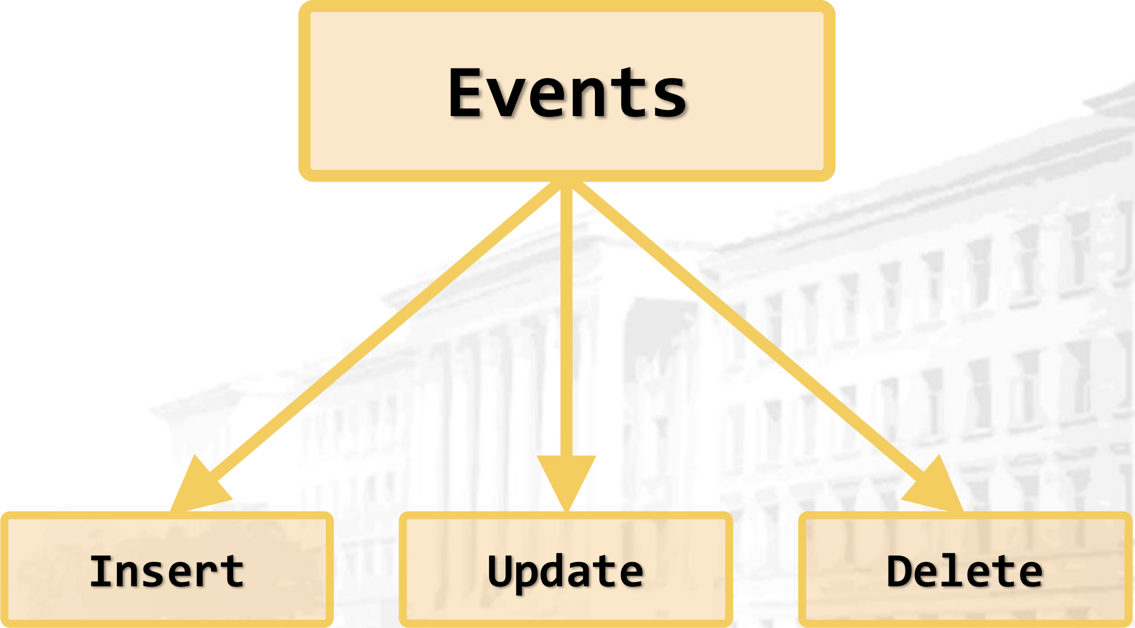
- Triggers - small programs in the database itself, activated by database events application layer
    - UPDATE, DELETE or INSERT queries
    - Called in case of specific event
- We do not call triggers explicitly
    - Triggers are attached to a table

MySQL Types of Triggers



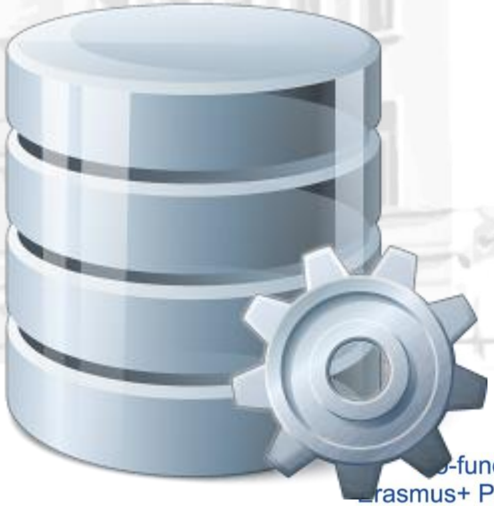
Events

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



Problem: Triggered

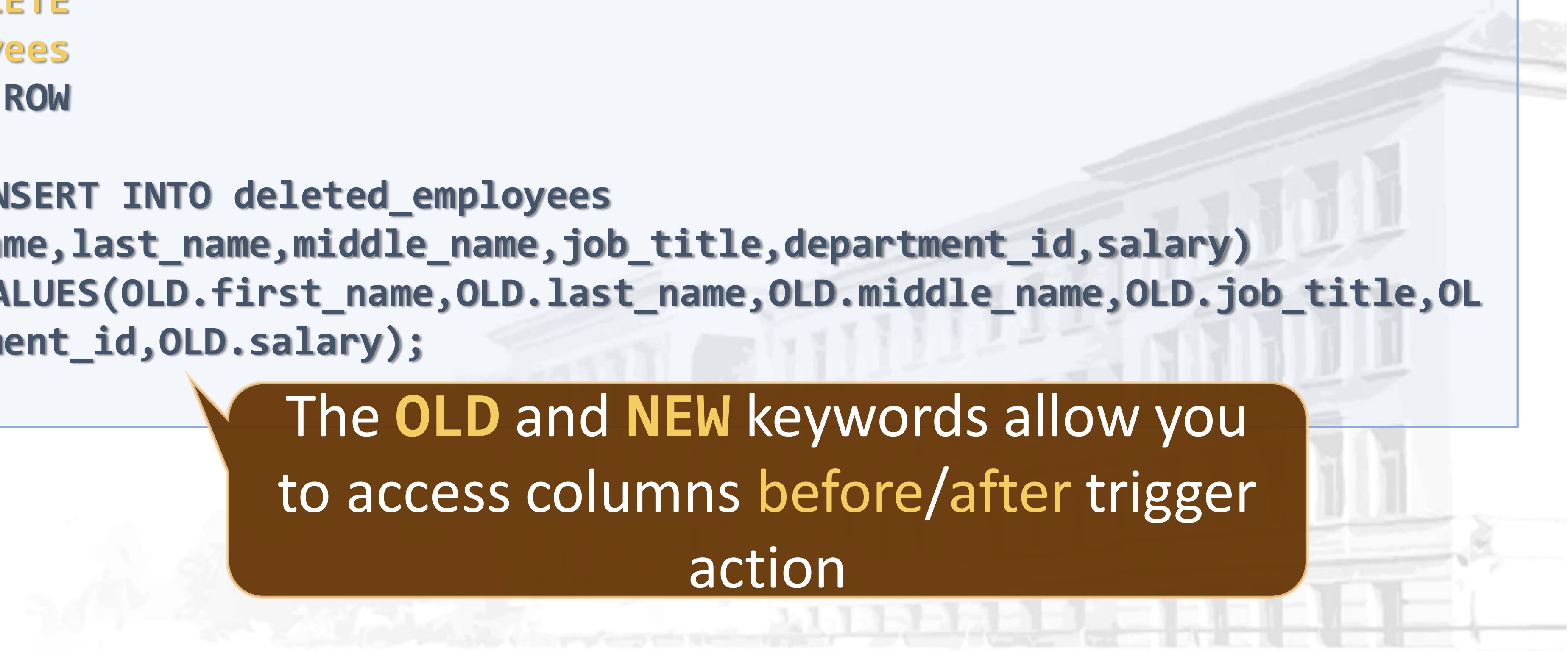
- Create a table deleted_employees with fields:
  - employee_id – primary key
  - first_name, last_name, middle_name, job_title, deparment_id, salary
- Add a trigger to employees table that logs deleted employees into the deleted_employees table
  - Use uni_ruse database



Solution: Triggered

```sql
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
);
```

Solution: Triggered



```
CREATE TRIGGER tr_deleted_employees
AFTER DELETE
ON employees
FOR EACH ROW
BEGIN
	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,OL
D.department_id,OLD.salary);
END;
```

The **OLD** and **NEW** keywords allow you to access columns before/after trigger action

Result: Triggered

- Trigger action result on **DELETE**:
  - NOTE: Remove foreign key checks before trying to delete employees
    - DO NOT submit foreign key restriction changes in the Judge System

```
DELETE FROM employees WHERE employee_id IN (1);
```

Data in deleted_employees table:

employee_id	first_name	last_name	…
1	Guy	Gilbert	…

Summary

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

REFERENCES

1. Robin Dewson, Beginning SQL Server for Developers, 4th Edition, Apress Publishing, pp. 705, ISBN: 1484202813, 2014;

2. Walter Shields, SQL QuickStart Guide: The Simplified Beginner's Guide to Managing, Analyzing, and Manipulating Data With SQL (QuickStart Guides™ - Technology), ClydeBank Media LLC, pp. 242, ISBN: 1945051752, 2019;

3. Vinicius Grippa, Sergey Kuzmichev, Learning MySQL: Get a Handle on Your Data, 2nd Edition, O'Reilly Media, pp. 629, ISBN: 1492085928, 2021;

4. Joel Murach, Murach's MySQL, 3rd Edition, Mike Murach & Associates Publishing, pp. 628, ISBN: 1943872368, 2019;

5. Rick Silva, MySQL Crash Course: A Hands-on Introduction to Database Development, No Starch Press, pp. 352, ISBN: 1718503008, 2023;

6. Sveta Smirnova, Alkin Tezuysal, MySQL Cookbook: Solutions for Database Developers and Administrators, 4th Edition, O'Reilly Media, pp. 971, ISBN: 1492093165, 2022;

7. Thomas Pettit, Scott Cosentino, The MySQL Workshop: A practical guide to working with data and managing databases with MySQL, Packt Publishing, pp. 726, ISBN: 1839214902, 2022;

8. Adam Aspin, Querying MySQL: Make your MySQL database analytics accessible with SQL operations, data extraction, and custom queries, BPB Publications, pp. 672, ISBN: 9355512678, 2022;

9. Rick Silva, MySQL Crash Course: A Hands-on Introduction to Database Development, No Starch Press, pp. 323, ISBN: 1718503008, 2023.