
Databases
- MySQL -

Goals of the MySQL Course

- Basic understanding of the Databases
- Create a simple Database
- Perform different queries

Theory - Databases

A Quick Example

```
Andrews, Archie   - (949)345-2222
Cooper, Betty     - (212)246-9846
Flanders, Ned     - (415)987-3451
Jones, Jughead    - (415)888-3777
Lodge, Veronica   - (714)332-0981
Snow, Jon         - (949)621-1908
Stark, Ned        - (310)119-6501
```

Find Ned Flanders' Phone Number

Find People With First Name "Ned"

Find All Phone Numbers With Area Code 415

Find All People Who Have a 3-letter First Name

A Phone Book!

Theory - Databases

What Is A Database?

A structured set of computerized data with an accessible interface

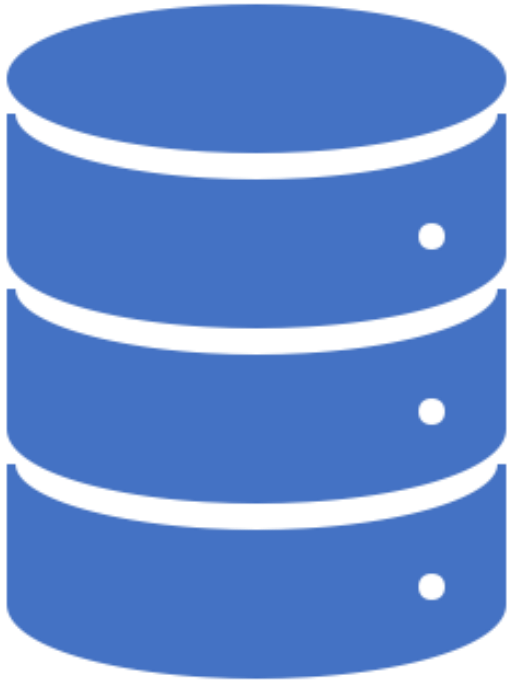
Theory - Relational DB

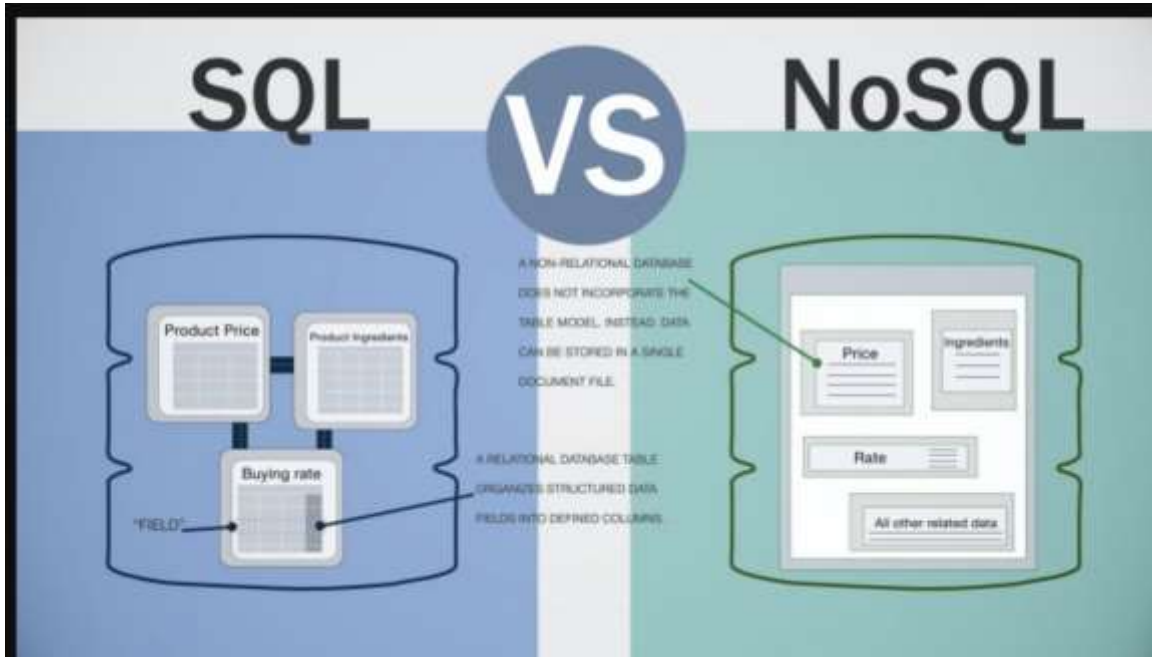
A database is an organized collection of data.

While one database can store multiple datasets, a table stores only one dataset.

Relational databases store data in multiple tables that are linked through different kinds of relationships.

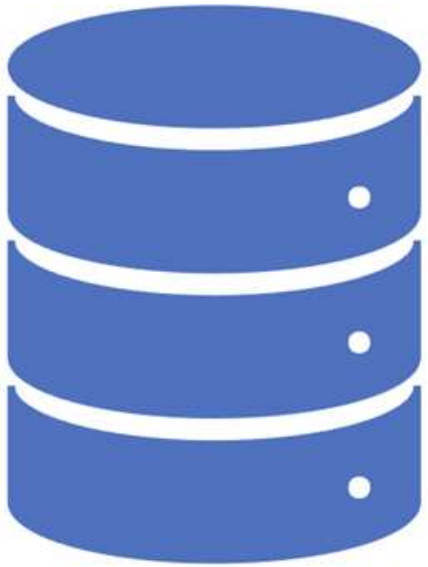
Theory - Relational DB vs non-relational





Theory - Relational DB vs non-relational

- **Relational databases** like MySQL, PostgreSQL and SQLite3 represent and store data in tables and rows. They're based on a branch of algebraic set theory known as **relational** algebra.
- Meanwhile, **non-relational databases** like MongoDB represent data in collections of JSON documents.



Theory - Relational DB

- **Bookstore database** - a bookstore must information related to books, customers, orders
  - Books information table: author, title, release year
  - Customers information table: first name, last name, phone number
  - Orders information table: the customer who ordered, books ordered, quantity, status
- **Fitness gym database** - a fitness gym must store information related to trainers, classes, subscriptions
  - Trainers information table: first name, last name, specialties
  - Classes information table: class name, description, schedule
  - Subscriptions information table: start date, end date, customer
- **Cinema database - a cinema database must store information related to the movies,**
- **schedules, tickets**
  - **Movies information table: movie title, description**
  - **Schedule information table: which movie, which room, start time, end time**
  - **Tickets: for which movie, adult/children ticket, seat number**

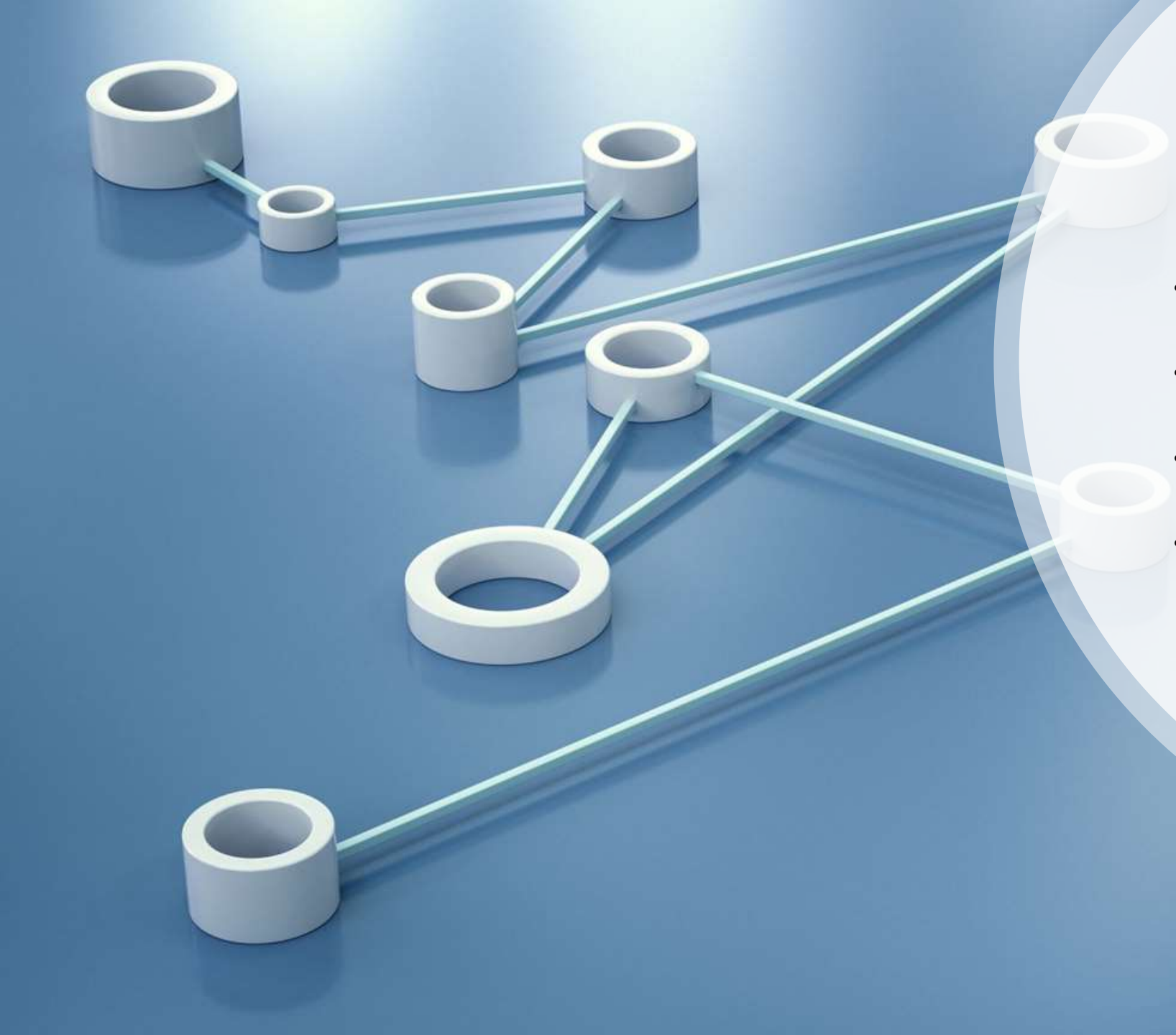
SQL

SQL - Structured Query Language

In order to interact with a relational databasewe can use the SQL (Structured Query Language) language.

SQL provides us with functionality to model, query, manipulate and control access to our databases.

Install MYSQL



For preparing the MySQL environment you need to follow the steps below

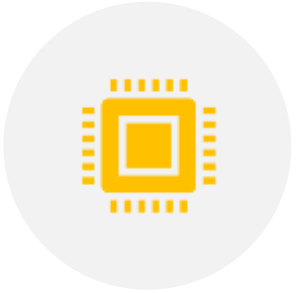
- Download MySQL Workbench
- Download MySQL Server MySQL Server
- Follow the installation steps in this document
- Note: Some installation require Microsoft Visual C++ installation. You download It here

SQL - subsets

DDL - data definition language - helps users define what kind of data they're going to store and how they're going to model this data

DML - data manipulation language - allows users to insert, update and delete data from the database

DQL - data query language - helps users retrieve information from the database



DCL - data control language - allows users to restrict and control access to the database

Drop DB

DROP	To drop a database: DROP DATABASE database_name;
DROP	Example: DROP DATABASE hello_world_db;
Remember	Remember to be careful with this command! Once you drop a database, it's gone!

Create DB

Start	Open the MySql client called Workbench
List	List available databases: show databases;
CREATE	The general command for creating a database: CREATE DATABASE <databaseName>;
CREATE	A specific example: CREATE DATABASE cinema;

Using the Database

Creating

- Creating a database does not select it for use; you must do that explicitly. To make menagerie the current database, use this statement:

USE

- USE <database name>;

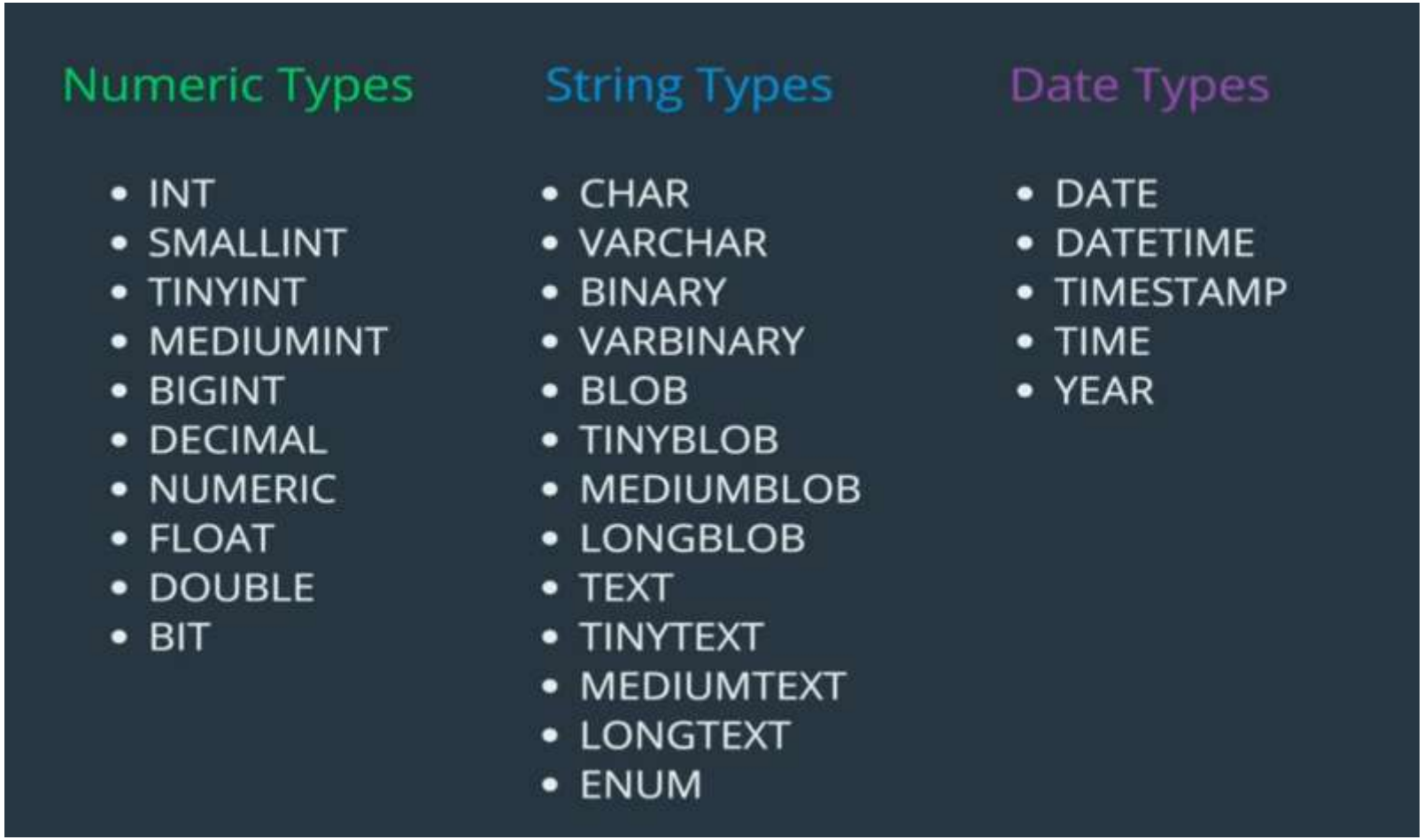
USE

- Example: USE dog_walking_app;

Tables

A relational DB is a bunch of tables

Tables hold the data (a collection of related data held in a structured format within a database)


Numeric Types
INT
SMALLINT
TINYINT
MEDIUMINT
BIGINT
DECIMAL
NUMERIC
FLOAT
DOUBLE
BIT
String Types
CHAR
VARCHAR
BINARY
VARBINARY
BLOB
TINYBLOB
MEDIUMBLOB
LONGBLOB
TEXT
TINYTEXT
MEDIUMTEXT
LONGTEXT
ENUM
Date Types
DATE
DATETIME
TIMESTAMP
TIME
YEAR

Datatypes

MySQL DATA TYPES

DATE TYPE	SPEC	DATA TYPE	SPEC
CHAR	String (0 - 255)	INT	Integer (-2147483648 to 214748-3647)
VARCHAR	String (0 - 255)	BIGINT	Integer (-9223372036854775808 to 9223372036854775807)
TINYTEXT	String (0 - 255)	FLOAT	Decimal (precise to 23 digits)
TEXT	String (0 - 65535)	DOUBLE	Decimal (24 to 53 digits)
BLOB	String (0 - 65535)	DECIMAL	"DOUBLE" stored as string
MEDIUMTEXT	String (0 - 16777215)	DATE	YYYY-MM-DD
MEDIUMBLOB	String (0 - 16777215)	DATETIME	YYYY-MM-DD HH:MM:SS
LONGTEXT	String (0 - 4294967295)	TIMESTAMP	YYYYMMDDHHMMSS
LONGBLOB	String (0 - 4294967295)	TIME	HH:MM:SS
TINYINT	Integer (-128 to 127)	ENUM	One of preset options
SMALLINT	Integer (-32768 to 32767)	SET	Selection of preset options
MEDIUMINT	Integer (-8388608 to 8388607)	BOOLEAN	TINYINT(1)

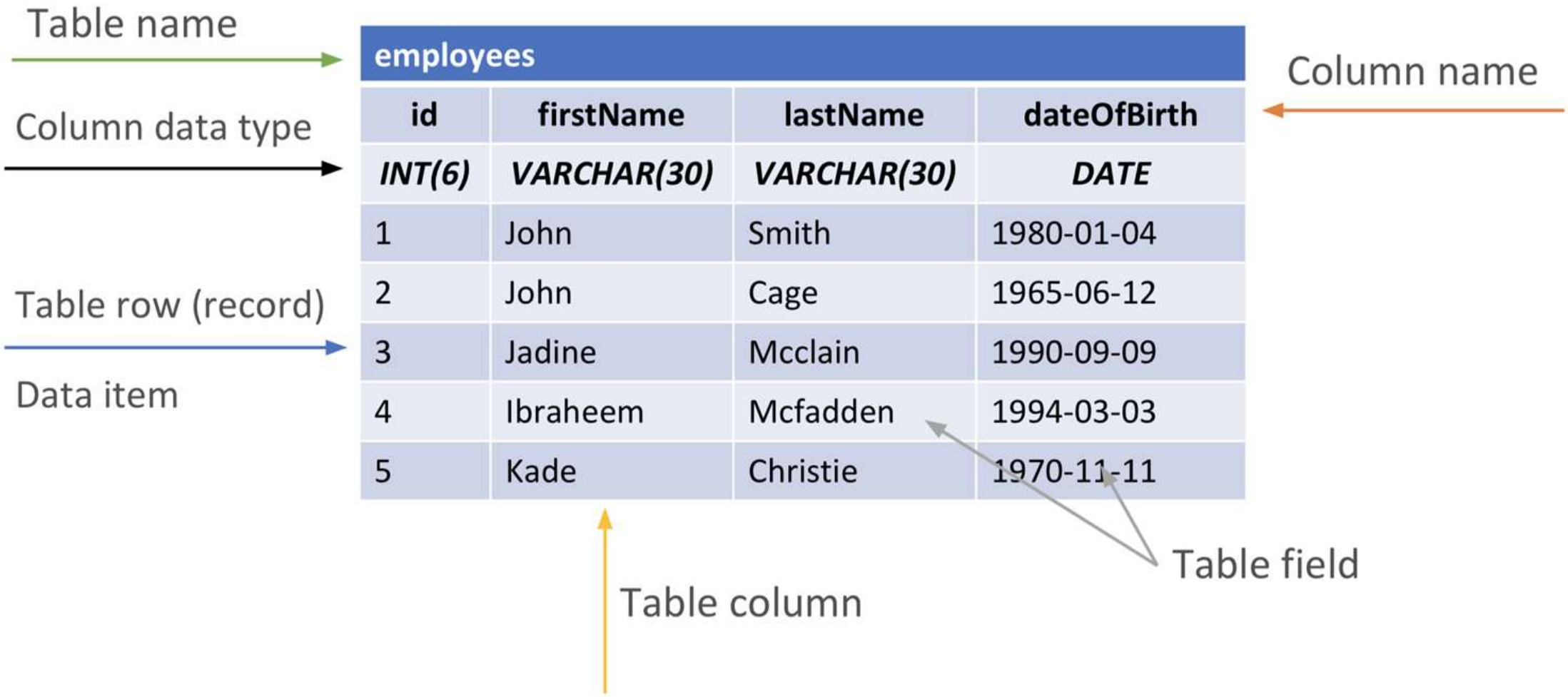


Table name
employees
Column name
Column data type
id
firstName
lastName
dateOfBirth
INT(6)
VARCHAR(30)
VARCHAR(30)
DATE
1
John
Smith
1980-01-04
2
John
Cage
1965-06-12
Table row (record)
3
Jadine
Mcclain
1990-09-09
Data item
4
Ibraheem
Mcfadden
1994-03-03
5
Kade
Christie
1970-11-11
Table field
Table column

DDL - Data Definition Language

The **CREATE DATABASE** instruction allows you to create a new database.

When creating a new database the only thing that you need to provide is the database name.

After a database has been created you can start defining the tables inside of it.

```
CREATE DATABASE database_name;
```

eg.

```
CREATE DATABASE petclinic;
```

The **CREATE TABLE** instruction allows you to create a new table in the database.

When creating a new table you need to provide the table name along with its column names, column definition and constraints.

The column definition refers to the column data type and properties.

```
CREATE TABLE table_name(
   col1_name column_1_definition,
   col2_name column_2_definition,
   ... ,
   [table_constraints]
);
```

eg.

```
CREATE TABLE owners(
   firstName VARCHAR(25) NOT NULL,
   lastName VARCHAR(25) NOT NULL
);
```

The **ALTER TABLE ADD** statement allows you to add one or more columns to a table

```
ALTER TABLE table_name
ADD new_column_name column_definition;
```

The **ALTER TABLE DROP** statement allows you to remove one or more columns from a table

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

The **ALTER TABLE MODIFY** statement allows you to update one or more columns

```
ALTER TABLE table_name
MODIFY column_name column_definition;
```

eg.

```
ALTER TABLE owners
ADD dateOfBirth DATE NOT NULL;
```

The **DROP TABLE** statement removes a table and its data permanently from the database.

In MySQL, you can also remove multiple tables using a single DROP TABLE statement, each table is separated by a comma.

```
DROP TABLE table_name [, table_name];
```

eg.

```
DROP TABLE owners;
```

DDL exercise

Create	Create a new database: humanResources
Create	Create a new table: employees, with the following columns: •employeeId - Integer •firstName - Varchar •lastName - Varchar •dateOfBirth - Date •postalAddress - Varchar
Alter	Alter table: employees and add the following columns: •phoneNumber - Varchar •email - Varchar •salary - Integer
Alter	Alter table: employees and remove the following columns: •postalAddress
Create	Create a new table: employeeAddresses •country - Varchar
Remove	Remove table: employeeAddresses

DDL exercise

Create	Create DB •create database humanResources;
Use	Use database •use humanResources;
Create	Create table •create table employees (employeeId INT, firstName VARCHAR(100), lastName VARCHAR(100), dateOfBirth DATE, postalAddress VARCHAR(200));
Alter	Alter - add •alter table employees add phoneNumber VARCHAR(100);
Alter	Alter - add •alter table employees add email VARCHAR(100);
Alter	Alter - add •alter table employees add salary INT; •desc employees;
Alter	Alter - drop •alter table employees drop column postalAddress;
Create	Create table •create table employeeAddresses (country VARCHAR(100));
Drop	Drop table •drop table employeeAddresses;

DML - Data Manipulation Language

The **INSERT INTO** statement allows you to insert one or more rows into a table.

First, specify the table name and a list of comma-separated column names inside parentheses after the INSERT INTO clause.

Then, put a comma-separated list of values of the corresponding columns inside the parentheses following the VALUES keyword.

```
INSERT INTO table_name(col1, col2, ... )
VALUES (val1, val2, ... ),
       (val1', val2', …'),
       (val1'', val2'', …'');
```

eg.

```
INSERT INTO owners
(firstName, lastName, dateOfBirth)
VALUES
('Jim', 'Jameson', '1980-01-01');
```

To delete data from a table, you can use the **DELETE FROM** statement.

```
DELETE FROM table_name
[WHERE condition];
```

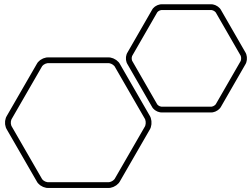
First, specify the table from which you delete data.

eg.

```
DELETE FROM owners;
```

Second, use a condition to specify which rows to delete in the WHERE clause. If the row matches the condition, it will be deleted. Notice that the WHERE clause is optional. If you omit it, the DELETE statement will delete all rows in the table.

Besides deleting data from a table, the DELETE statement returns the number of rows deleted.



DML exercise

01

Insert into table: employees a new entry:

- employeeId - 1
- firstName - John
- lastName - Johnson
- dateOfBirth - 1975-01-01
- phoneNumber - 0-800-800-314
- email - john@johnson.com
- salary - 1000

02

Update dateOfBirth of John Johnson to 1980-01-01

03

Delete everything from table: employees

04

Add two more entries in employees:

- 'John', 'Johnson', '1975-01-01', '0-800-800-888', 'john@johnson.com', 1000
- 'James', 'Jameson', '1985-02-02', '0-800-800-999', 'james@jameson.com', 2000

DML exercise

Insert	insert into employees (employeeId, firstName, lastName, dateOfBirth, phoneNumber, email, salary) values (1, 'John', 'Johnson', '1975-01-01', '0-800-800-314', 'john@johnson.com', 1000);
Update	update employees set dateOfBirth=1980-01-01; //fix the error
Select	select * from employees;
Delete	delete from employees;
Select	Do by yourself
Add other entries	insert into employees (employeeId, firstName, lastName, dateOfBirth, phoneNumber, email, salary) values (1, 'John', 'Johnson', '1975-01-01', '0-800-800-888', 'john@johnson.com', 1000), (2, 'James', 'Jameson', '1985-02-02', '0-800-800-999', 'james@jameson.com', 2000);
Select	Do by yourself

DQL - Data Query Language

The **SELECT** statement allows you to read data from one or more tables.

First you specify a list of columns or expressions that you want to show in the result.

Then you specify the table that you want to select from.

```
SELECT select_list
FROM table_name
[WHERE condition];
```

```
SELECT * FROM table_name;
- selects all columns from the table
```

```
SELECT col1, col2 FROM table_name;
- selects only col1 and col2 from table
```

eg.

```
SELECT * FROM owners;
```

The **WHERE** clause allows you to specify a search condition for the rows returned by a query.

By specifying a condition, the SELECT instruction will no longer return all of the results but only those that match the specified condition.

The search condition is a combination of one or more predicates using the logical operator AND, OR and NOT.

```
SELECT select_list
FROM table_name
WHERE condition;
```

The **WHERE** clause allows you to specify a number of comparison operators:

- a=b
- a>b, a>b, a<=b, a>=b
- a IN (value1, value2, value3)
- a IS NULL, a IS NOT NULL
- a!=b
- a BETWEEN b AND c
- a LIKE b

```
SELECT select_list
FROM table_name
WHERE condition;
```

eg.

```
SELECT *
FROM owners
WHERE lastName = 'Johnson';
```

The **WHERE** clause allows you to specify logical operators that can be used to combine multiple comparison operators:

- AND
- OR
- NOT

```
SELECT select_list
FROM table_name
WHERE condition;
```

eg.

```
SELECT *
FROM owners
WHERE lastName = 'Johnson' AND
firstName = 'James';
```

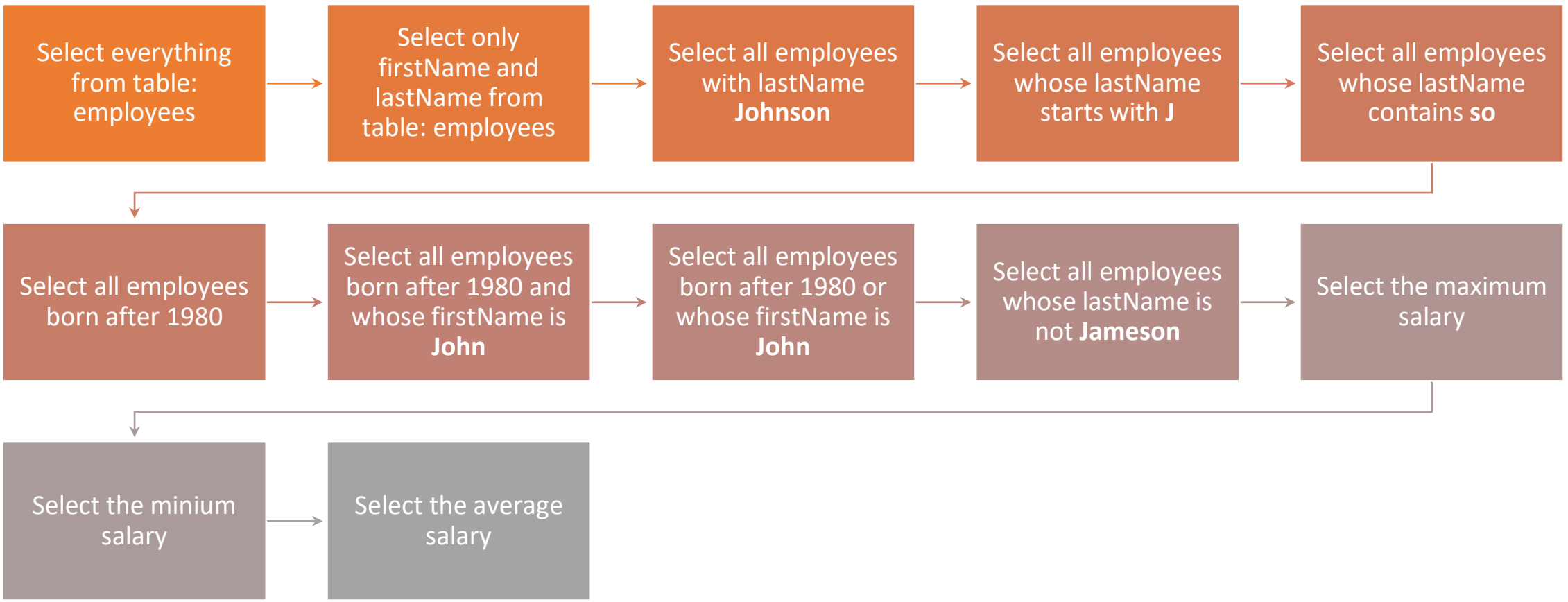
An aggregate function performs a calculation on multiple values and returns a single value:

- **AVG** - takes multiple numbers and returns the average value of the numbers
- **SUM** - returns the summation of all values
- **MAX** - returns the highest value
- **MIN** - returns the lowest value
- **COUNT** - returns the number of rows

```
SELECT AVG(col1)
FROM tableName;

SELECT SUM(col1)
FROM tableName;

SELECT MAX(col1)
FROM tableName;

SELECT MIN(col1)
FROM tableName;

SELECT COUNT(*)
FROM tableName;
```

eg. `SELECT COUNT(*) FROM owners;`

DQL exercise



DQL exercise

Select all	select * from employees;
Select 2 columns	select firstName, lastName from employees;
Select by lastName	select * from employees where lastName='Johnson';
Select start by	select * from employees where left (lastName, 1)='J';
Select like	select * from employees where lastName like '%so%';
Select after year	select * from employees where YEAR(dateOfBirth) > 1980;

Select after year and ..

- select * from employees where YEAR(dateOfBirth) > 1980 and firstName = 'John';

Select after year or ..

- select * from employees where YEAR(dateOfBirth) > 1980 or firstName = 'John';

Select where not

- select * from employees where lastName != 'John';

Select max

- select MAX(salary) from employees;

Select min

- select MIN(salary) from employees;

Select average

- select AVG(salary) from employees;

DQL exercise

Database Relationships

Table relationship - optional

THERE ARE SEVERAL TYPES OF DATABASE RELATIONSHIPS:

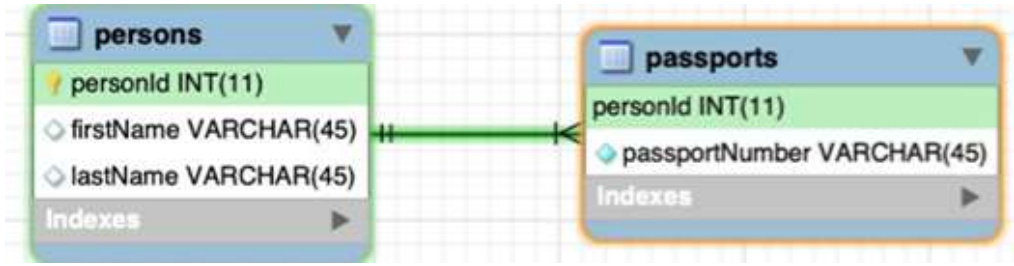
ONE TO ONE RELATIONSHIPS

ONE TO MANY AND MANY TO ONE RELATIONSHIPS

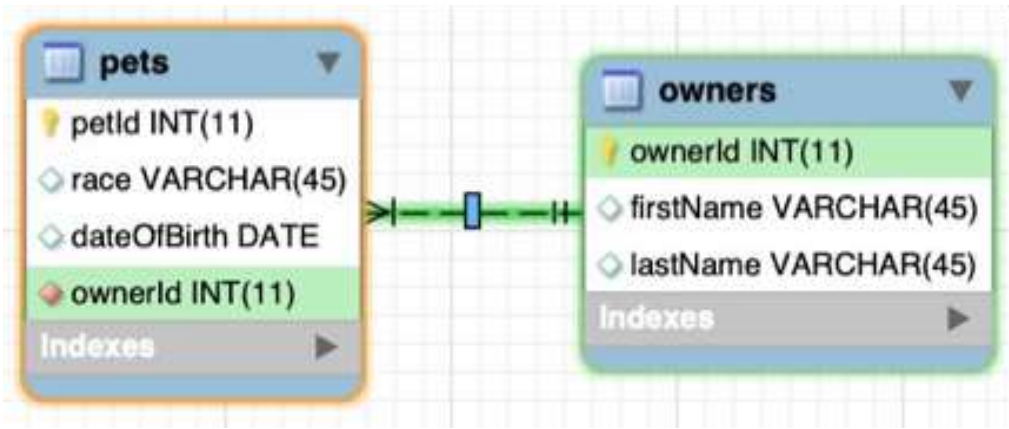
MANY TO MANY RELATIONSHIPS

SELF REFERENCING RELATIONSHIPS

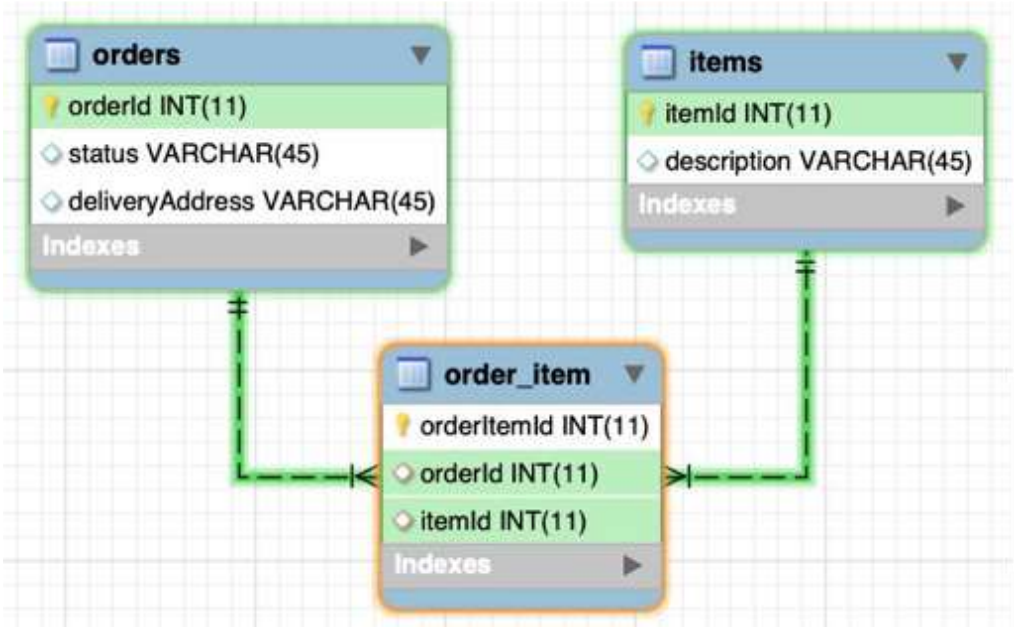
One-to-one -optional-



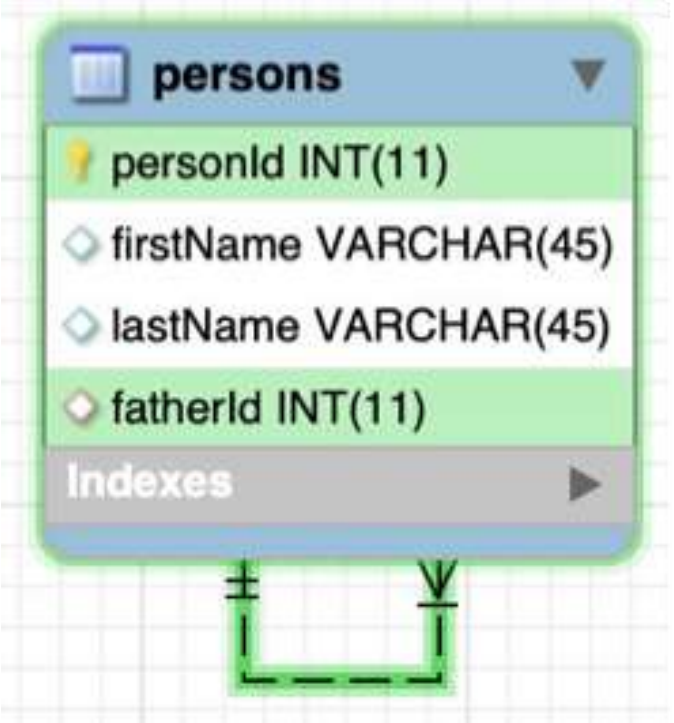
One-to-many -optional-



Many-to-many -optional-



Self-referencing
-optional-



Primary key - optional

- A **primary key** is a **column** or a s**et of columns** that **uniquely identifies** each **row** in the table.
- A **primary key must contain unique values**. If the primary key consists of multiple columns, the combination of values in these columns must be unique.
- A primary key column **cannot have NULL values**.
- A **table** can have one an **only one primary key**.
- Because MySQL works faster with integers, the **data type** of the **primary key** column **should be the integer** e.g., INT, BIGINT. And you should ensure sure that value ranges of the integer type for the primary key are sufficient for storing all possible rows that the table may have.
- A **primary key column** often **has the AUTO_INCREMENT attribute** that **automatically generates a sequential integer** whenever you insert a new row into the table.

Primary key - optional

The PRIMARY KEY constraint allows you to define a primary key of a table when you create or alter table.

Typically, you define the primary key for a table in the CREATE TABLE statement.

If a table, for some reasons, does not have a primary key, you can use the ALTER TABLE statement to add a primary key.

```
CREATE TABLE table_name (
    primary_key_column INT
AUTO_INCREMENT PRIMARY KEY NOT NULL,
    col2 col2_definition,
    col3 col3_definition,
     ...
);

ALTER TABLE table_name
ADD PRIMARY KEY(column_list);


eg.
ALTER TABLE owners
ADD PRIMARY KEY(ownerId);
```

Foreign key - optional

- **A foreign key** is a **column or group of columns** in **a table** that **links to** a column or group of columns in **another table**.
- The foreign key places constraints the related tables, so MySQL can maintain referential integrity.
- The **table containing the foreign key** is called the **child table**, and the referenced table is the **parent table**.
- Typically, the **foreign key columns** of the child table often **refer to** the **primary key columns** of the parent table
- A **table can have more than one foreign key** where each foreign key references to a primary key of the different parent tables.
  Once a foreign key constraint is in place, the **foreign key columns** from the **child table must have the corresponding row in the parent key columns** of the parent table or values in these foreign key column must be NULL

Foreign key - optional

First, specify the name of the foreign key constraint that you want to create. The foreign key name is optional and is generated automatically if you skip it.

Second, specify a list of comma-separated foreign key columns after the FOREIGN KEY keywords.

Third, specify the parent table followed by a list of comma-separated columns to which the foreign key columns reference.

Foreign key name is unique. As a best practice use: *fk_childTableName_parentTableName*

```
CREATE TABLE table_name(
    primaryKeyColumn INT AUTO_INCREMENT PRIMARY
KEY NOT NULL,
    col2 col2_definition,
    col3 col3_definition,
    CONSTRAINT [foreign_key_name] FOREIGN KEY
    (column_name, ... )
    REFERENCES parent_table(colunm_name, ... )
);
```

eg.

```
CREATE TABLE pets (
  petId INT NOT NULL AUTO_INCREMENT,
  race VARCHAR(45) NOT NULL,
  dateOfBirth DATE NOT NULL,
  ownerId INT NOT NULL,
  PRIMARY KEY (petId),
  CONSTRAINT fk_pets_owners
    FOREIGN KEY (ownerId)
    REFERENCES owners (ownerId));
```

Joins

Joins

SQL **Join statement** is used to **combine data or rows from two or more tables** based on a **common field between them**.

- CROSS JOIN
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN

Cross join

CROSS JOIN matches each row from the first table with each row of the second table.

If each table had 4 rows, we should be getting a result of 16 rows.

```
SELECT * FROM table1 JOIN table2

SELECT * FROM table1, table2
```

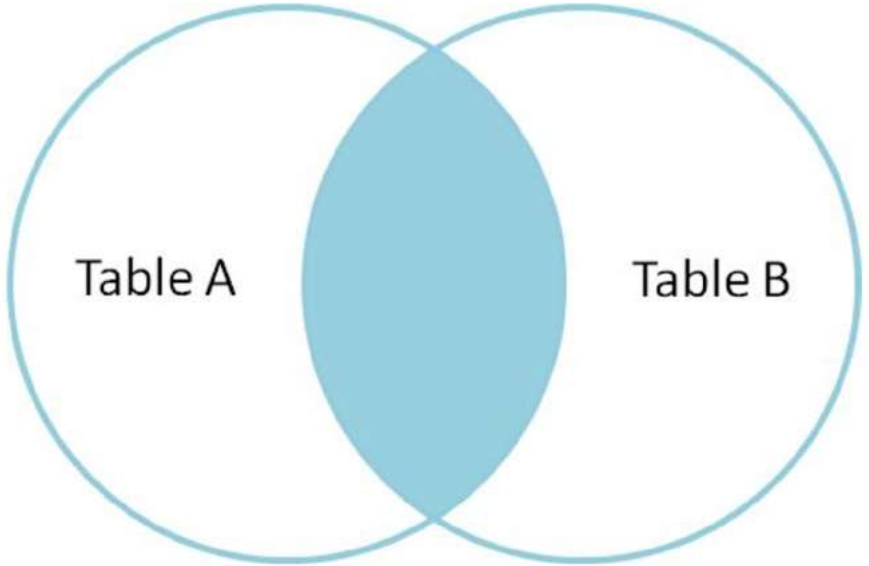
eg.

```
SELECT * FROM owners
JOIN pets
```

Inner Join

The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies.

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column = table2.column
```

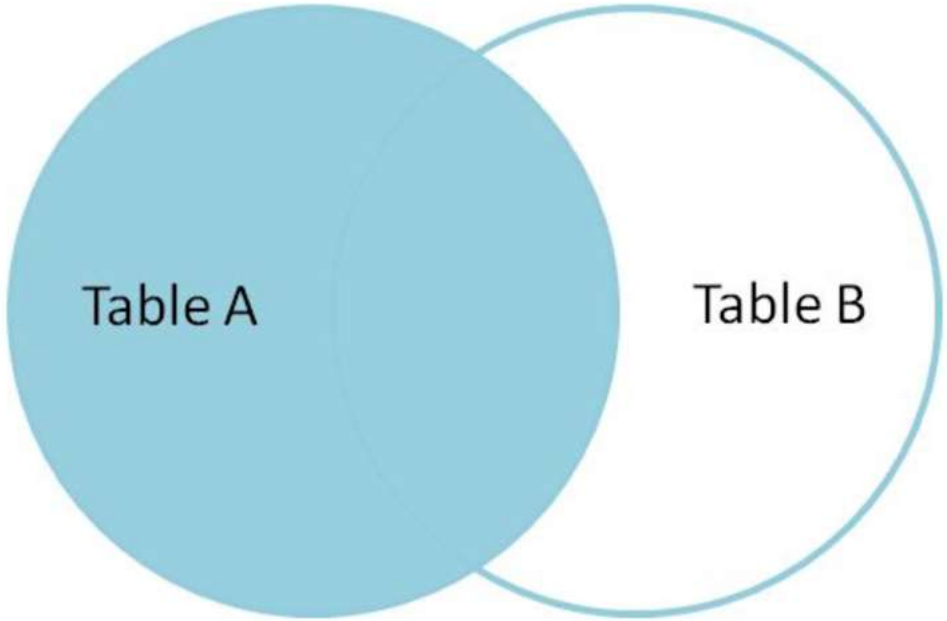


eg.

```
SELECT *
FROM pets
INNER JOIN owners
ON pets.ownerId = owners.ownerId
```

Left join

The LEFT JOIN returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. For the rows for which there is no matching row on right side, the result-set will contain null.



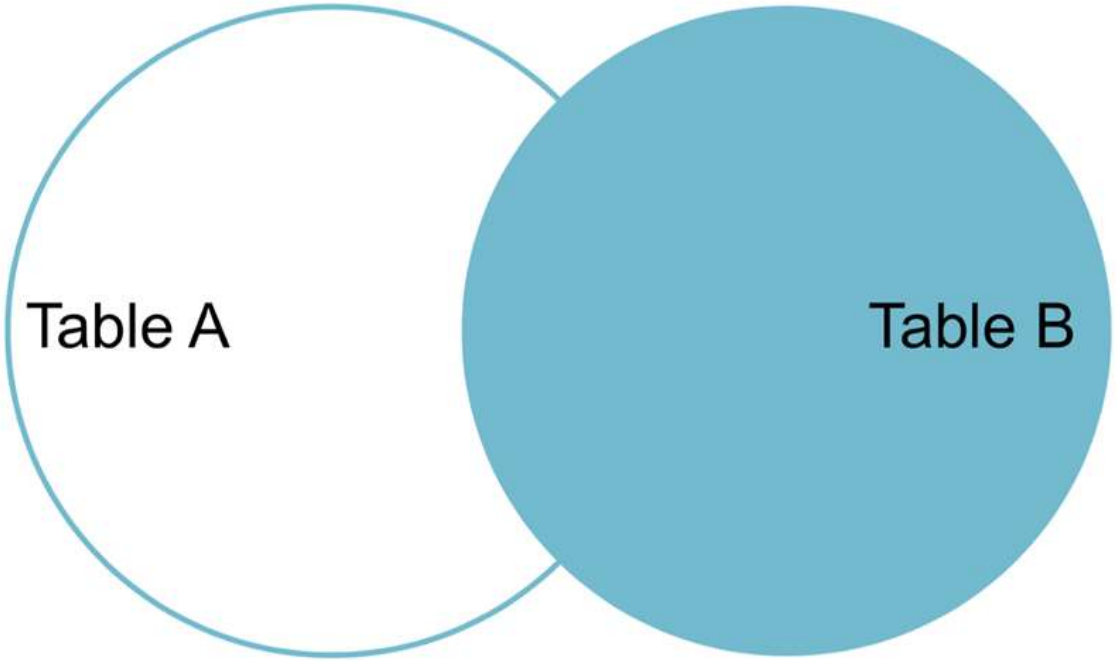

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column = table2.column
```

eg.

```
SELECT *
FROM pets
LEFT JOIN owners
ON pets.ownerId = owners.ownerId
```

Right join

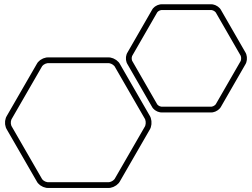
The RIGHT JOIN returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. For the rows for which there is no matching row on left side, the result-set will contain null.




```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column
```

eg.

```
SELECT *
FROM pets
RIGHT JOIN owners
ON pets.ownerId = owners.ownerId
```



Joins exercise

We will create two tables

We will insert some data on both of them

Apply all join available types

Joins exercise

Create 1st table	create table smoke_tests (id INT, name VARCHAR(100), steps VARCHAR (300));
Create 2nd table	create table regression_tests (id INT, name VARCHAR(100), steps VARCHAR (300));
Insert in 1st table	insert into smoke_tests(id, name, steps) values (1, 'tc1', '3 steps'), (2, 'tc2', '2 steps');
Insert in 2nd table	insert into regression_tests(id, name, steps) values (1, 'tc1', '3 steps'), (3, 'tc3', '4 steps’);
Cross join	select * from smoke_tests join regression_tests;
Inner join	select * from smoke_tests inner join regression_tests on smoke_tests.name= regression_tests.name;
Left join	select * from smoke_tests left join regression_tests on smoke_tests.name = regression_tests.name;
Right join	select * from smoke_tests right join regression_tests on smoke_tests.name = regression_tests.name;

SQL extras

- **ORDER BY** - is used to sort the result-set in ascending or descending order
a. SELECT column1, column2, ... FROM table_name ORDER BY column1 [ASC|DESC];
- **GROUP BY** statement groups rows that have the same values into summary rows, like "find the number of customers in each country"
    - SELECT column1, column2, ... FROM table_name GROUP BY column1;
    - SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country;
- **HAVING** clause was added to SQL because the WHERE keyword could not be used with aggregate functions
    - SELECT column1, column2, ... FROM table_name GROUP BY column1 HAVING condition;
    - SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country HAVING
    - COUNT(CustomerID) > 5;
- **Aliases** are used to give a table, or a column in a table, a temporary name
    - SELECT column1 *as newName*, column2, ... FROM table_name;
    - SELECT COUNT(CustomerID) *as Nr Of Cust*, Country FROM Customers GROUP BY Country;
- **LIMIT** is used to restrict the number of results retrieved from the database
    a. SELECT * FROM table_name LIMIT 5;

CRUD

- CREATE
- READ
- UPDATE
- DELETE

CRUD – CREATE TABLES

- DROP ALL TABLES
- CREATE A NEW ONE FOR TESTS
- > CREATE TABLE tests (id INT NOT NULL AUTO_INCREMENT,
  - -> name VARCHAR(30),
  - -> steps VARCHAR(500),
  - -> expected VARCHAR(100),
  - -> PRIMARY KEY(id));
- INSERT VALUES
  - > INSERT INTO tests(name, steps, expected)
  - VALUES('tc1', '3 steps', 'asdfas'), ('tc2', ' 5 ste[s', 'asda'), ('tc3', 'sdfas', 'fdsfasd');

CRUD – READ

- All:
- > SELECT * FROM tests;
- Select expression:
- > SELECT name FROM tests;
- > SELECT steps FROM tests;
- > SELECT expected FROM tests;
- > SELECT name, steps FROM tests;

CRUD – READ + condition

- Select by steps:
- SELECT * FROM tests WHERE expected=‘asda’;
- Select by name:
- SELECT * FROM tests WHERE name=‘tc1';
- Notice how it deals with case:
- SELECT name, expected FROM tests WHERE name=‘Tc1';

CRUD – Aliases

- > select name as n, steps as s, expected from tests;

CRUD – Update

- Alter existing data
- mysql> update tests set name='n' where expected = 'asda’;
- Fix it again;
- mysql> update tests set name='n' where name = ‘n’;
- Insert 2 new tests:
- mysql> INSERT into tests(name, steps, expected) VALUES('tc4', 'x', 'map visible'), ('tc5', 'y', 'erro
- r');
- Update tc5:
- mysql> UPDATE tests set steps='z' where name = 'tc5';

CRUD – DELETE

- **Insert a new test:** insert into tests(name, steps, expected) values('test', 't', 't’);
- **Delete test:** delete from tests; | delete from tests where name = 'tests’;
- **DROP vs DELETE:** DELETE still keep the table as empty but DROP will erase the table
- **Create a new table tests 2:** create table tests2(a VARCHAR(2), b INT);
- **Insert some values:** insert into tests2(a, b) values (‘2’, 2);
- **DELETE them:** delete from tests2;
- **Drop table:** drop table tests2;

CRUD Exercise - Create

- SELECT database();
- CREATE DATABASE shirts_db;
- use shirts_db;
- SELECT database();

```
CREATE TABLE shirts
(
        shirt_id INT NOT NULL AUTO_INCREMENT,
        article VARCHAR(100),
        color VARCHAR(100),
        shirt_size VARCHAR(100),
        last_worn INT,
        PRIMARY KEY(shirt_id)
);
```

CRUD Exercise - Read

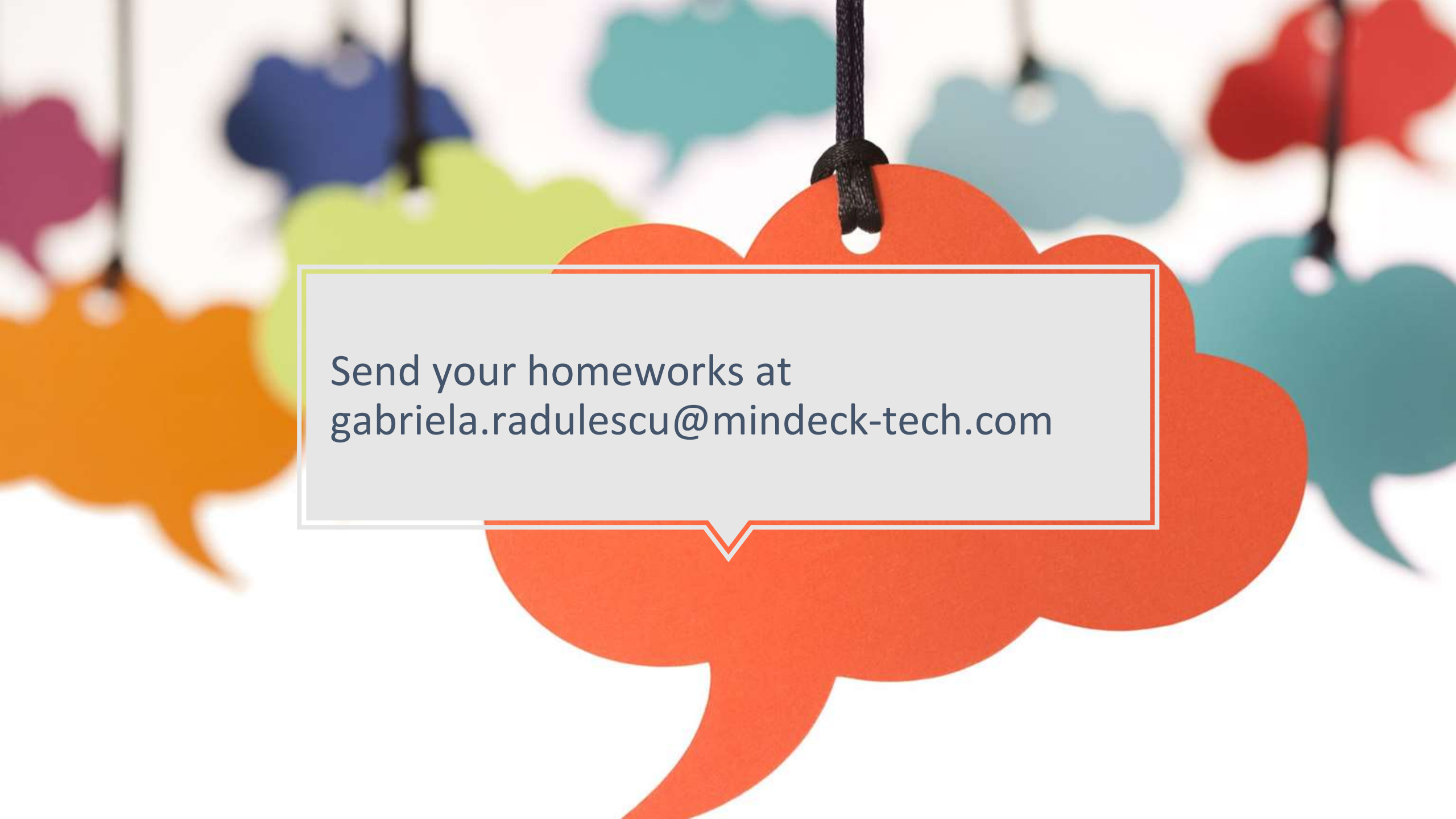
- SELECT article, color FROM shirts;
- SELECT * FROM shirts WHERE shirt_size='M';
- SELECT article, color, shirt_size, last_worn
  FROM shirts
  WHERE shirt_size='M';

CRUD Exercise - Update

- SELECT * FROM shirts WHERE article='polo shirt';
- UPDATE shirts SET shirt_size='L' WHERE article='polo shirt';
- SELECT * FROM shirts WHERE article='polo shirt';
- SELECT * FROM shirts;
- SELECT * FROM shirts WHERE last_worn=15;
- UPDATE shirts SET last_worn=0 WHERE last_worn=15;
- SELECT * FROM shirts WHERE last_worn=15;
- SELECT * FROM shirts WHERE last_worn=0;
- SELECT * FROM shirts WHERE color='white';
- UPDATE shirts SET color='off white', shirt_size='XS' WHERE color='white';
- SELECT * FROM shirts WHERE color='white';
- SELECT * FROM shirts WHERE color='off white';
- SELECT * FROM shirts;

CRUD Exercise - Delete

- SELECT * FROM shirts;
- SELECT * FROM shirts WHERE last_worn=200;
- DELETE FROM shirts WHERE last_worn=200;
- SELECT * FROM shirts WHERE article='tank top';
- DELETE FROM shirts WHERE article='tank top';
- SELECT * FROM shirts WHERE article='tank top';
- SELECT * FROM shirts;
- DELETE FROM shirts;
- SELECT * FROM shirts;
- DROP TABLE shirts;
- show tables;
- DESC shirts;


Send your homeworks at
gabriela.radulescu@mindeck-tech.com

Sometimes it is the people who no one imagined anything of who do the things that no one can imagine.