

# SQL – MySQL for Data Analytics and Business Intelligence

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows on one side. The chairs are arranged in several rows, facing towards the front of the room where a presentation screen or podium would be.

# SQL Theory

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of black office chairs facing towards the front. The front wall features several large, dark rectangular panels, likely projection screens or television monitors. The room has a high ceiling with a grid of recessed lighting fixtures. Large windows along the side walls provide a view of an outdoor area with trees and possibly a parking lot.

# Data Definition Language (DDL)

# Data Definition Language

- SQL's syntax

# Data Definition Language

- **SQL's syntax**

comprises several types of statements that allow you to perform various commands and operations

# Data Definition Language

- **SQL's syntax**

comprises several types of statements that allow you to perform various commands and operations

- **Data Definition Language (DDL)**

# Data Definition Language

- **SQL's syntax**

comprises several types of statements that allow you to perform various commands and operations

- **Data Definition Language (DDL)**

- a syntax
- a set of statements that allow the user to define or modify data structures and objects, such as tables

# Data Definition Language

- **SQL's syntax**

comprises several types of statements that allow you to perform various commands and operations

- **Data Definition Language (DDL)**

- a syntax
- a set of statements that allow the user to define or modify data structures and objects, such as tables

- **the CREATE statement**

# Data Definition Language

- **SQL's syntax**

comprises several types of statements that allow you to perform various commands and operations

- **Data Definition Language (DDL)**

- a syntax
- a set of statements that allow the user to define or modify data structures and objects, such as tables

- **the CREATE statement**

used for creating entire databases and database objects as tables

# Data Definition Language

- the CREATE statement

used for creating entire databases and database objects as tables



SQL

```
CREATE object_type object_name;
```

# Data Definition Language

- the CREATE statement

used for creating entire databases and database objects as tables



CREATE object\_type object\_name;

SQL

CREATE TABLE object\_name (column\_name data\_type);

# Data Definition Language



SQL

```
CREATE TABLE object_name (column_name data_type);
```

# Data Definition Language



SQL

```
CREATE TABLE object_name (column_name data_type);
```

```
CREATE TABLE sales (purchase_number INT);
```

# Data Definition Language



SQL

```
CREATE TABLE object_name (column_name data_type);
```

```
CREATE TABLE sales (purchase_number INT);
```

sales
purchase_number

# Data Definition Language



SQL

```
CREATE TABLE sales (purchase_number INT);
```

sales
purchase_number

- the table name can coincide with the name assigned to the database

# Data Definition Language

- the ALTER statement

# Data Definition Language

- the ALTER statement  
used when altering existing objects

# Data Definition Language

- **the ALTER statement**

used when altering existing objects

- - ADD
  - REMOVE
  - RENAME

# Data Definition Language

**sales**

<b>purchase_number</b>

# Data Definition Language



SQL

```
ALTER TABLE sales  
ADD COLUMN date_of_purchase DATE;
```

sales	
	purchase_number

# Data Definition Language



SQL

```
ALTER TABLE sales  
ADD COLUMN date_of_purchase DATE;
```

sales

purchase_number	date_of_purchase

# Data Definition Language

- the DROP statement

# Data Definition Language

- the DROP statement  
used for deleting a database object

# Data Definition Language



SQL

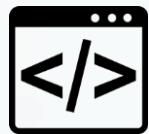
```
DROP object_type object_name;
```

customers

customer_id	first_name

# Data Definition Language

used for deleting a database object



SQL

```
DROP object_type object_name;
```

```
DROP TABLE customers;
```

customers

customer_id	first_name

# Data Definition Language

used for deleting a database object



SQL

```
DROP object_type object_name;
```

```
DROP TABLE customers;
```

customers

customer_id	first_name

# Data Definition Language

- the RENAME statement

# Data Definition Language

- the RENAME statement  
allows you to rename an object

# Data Definition Language



SQL

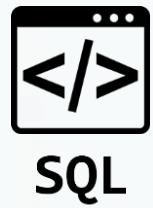
```
RENAME object_type object_name TO new_object_name;
```

customers

customer_id	first_name

# Data Definition Language

used for deleting a database object



```
RENAME object_type object_name TO new_object_name;
```

```
RENAME TABLE customers TO customer_data;
```

customers

customer_id	first_name

# Data Definition Language

used for deleting a database object



SQL

```
RENAME object_type object_name TO new_object_name;
```

```
RENAME TABLE customers TO customer_data;
```

customer_id	first_name

# Data Definition Language

used for deleting a database object



SQL

```
RENAME object_type object_name TO new_object_name;
```

```
RENAME TABLE customers TO customer_data;
```

customer\_data

customer_id	first_name

# Data Definition Language

- the TRUNCATE statement

# Data Definition Language

- the TRUNCATE statement

instead of deleting an entire table through DROP, we can also remove its data and continue to have the table as an object in the database

# Data Definition Language



SQL

```
TRUNCATE object_type object_name;
```

customers

customer_id	first_name

# Data Definition Language

used for deleting a database object



```
TRUNCATE object_type object_name;
```

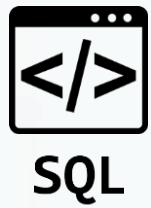
```
TRUNCATE TABLE customers;
```

customers

customer_id	first_name

# Data Definition Language

used for deleting a database object



```
TRUNCATE object_type object_name;
```

```
TRUNCATE TABLE customers;
```

customers

customer_id	first_name

# Data Definition Language

- Data Definition Language (DDL)

# Data Definition Language

- Data Definition Language (DDL)

- CREATE
- ALTER
- DROP
- RENAME
- TRUNCATE

**Next:**

Next:

Keywords

Next:

Keywords

Data Manipulation Language

A large, modern conference room with a long rectangular table in the center. Rows of black office chairs are arranged facing the table. The room has large windows on one side, providing a view of an outdoor area. The ceiling is white with a grid of recessed lights. The overall atmosphere is professional and spacious.

# SQL Keywords

# Keywords

- **Keywords:**

- ADD

# Keywords

- **Keywords:**

- ADD
- CREATE
- ALTER
- etc.

# Keywords

- **Keywords:**

- ADD
- CREATE
- ALTER
- etc.

- **KEYWORDS IN SQL CANNOT BE VARIABLE NAMES!**

# Keywords

- **Keywords:**

- ADD
- CREATE
- ALTER
- etc.

- **KEYWORDS IN SQL CANNOT BE VARIABLE NAMES!**

objects or databases cannot have names that coincide with SQL keywords

# Keywords

- CREATE, ALTER:

# Keywords

- CREATE, ALTER:



SQL

```
CREATE TABLE alter (purchase_number INT);
```

alter

purchase_number

# Data Definition Language

- ADD

# Data Definition Language

- ADD



SQL

```
ALTER TABLE sales  
ADD COLUMN date_of_purchase DATE;
```

**sales**

purchase_number	date_of_purchase

# Data Definition Language

- ADD, ALTER



SQL

```
ALTER TABLE sales  
ADD COLUMN date_of_purchase DATE;
```

**sales**

purchase_number	date_of_purchase

# Keywords

- Keywords = reserved words

# Keywords

- Keywords = reserved words  
they cannot be used when naming objects

A large, modern conference room is shown from a perspective looking down the length of the room. On the left, there are several rows of black office chairs facing towards the right. The room has a high ceiling with a grid of recessed lighting. Large windows along the right wall provide a view of an outdoor area. The overall atmosphere is professional and spacious.

# Data Manipulation Language (DML)

# Data Manipulation Language

- **Data Manipulation Language (DML)**

its statements allow us to manipulate the data in the tables of a database

- **the SELECT statement**

used to retrieve data from database objects, like tables

# Data Manipulation Language



SQL

```
SELECT * FROM sales;
```

sales
purchase_number

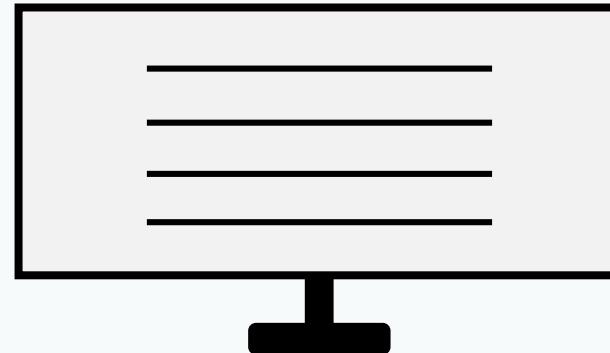
# Data Manipulation Language



SQL

```
SELECT * FROM sales;
```

purchase_number



# Data Manipulation Language



SQL

```
SELECT... FROM sales;
```

sales
purchase_number

# Data Manipulation Language



SQL

```
SELECT... FROM sales;
```

sales
purchase_number

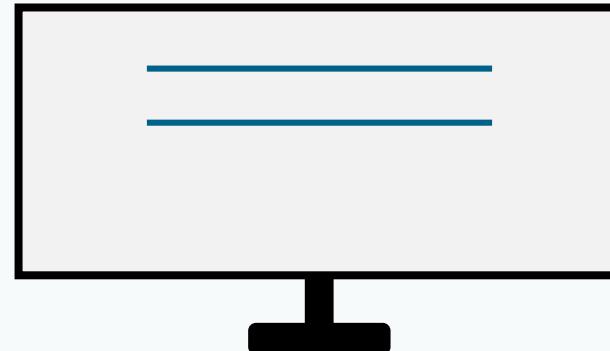
# Data Manipulation Language



SQL

`SELECT... FROM sales;`

sales
purchase_number



# Data Manipulation Language

- Why are we going to need just a piece of the table?

- imagine a table with 2 million rows of data
- it can be helpful if you could extract only a portion of the table that satisfies given criteria
- you should know how to use SELECT perfectly well

# Data Manipulation Language

- the INSERT statement  
used to insert data into tables
- INSERT INTO... VALUES...;

# Data Manipulation Language



SQL

```
INSERT INTO sales (purchase_number, date_of_purchase) VALUES  
(1, '2017-10-11');
```

sales

purchase_number	date_of_purchase

# Data Manipulation Language



SQL

```
INSERT INTO sales (purchase_number, date_of_purchase) VALUES  
(1, '2017-10-11');
```

sales

purchase_number	date_of_purchase
1	2017-10-11

# Data Manipulation Language



SQL

```
INSERT INTO sales VALUES  
(1, '2017-10-11');
```

sales

purchase_number	date_of_purchase
1	2017-10-11

# Data Manipulation Language



SQL

```
INSERT INTO sales (purchase_number, date_of_purchase) VALUES  
(1, '2017-10-11');
```

```
INSERT INTO sales VALUES  
(1, '2017-10-11');
```

# Data Manipulation Language



SQL

```
INSERT INTO sales (purchase_number, date_of_purchase) VALUES  
(2, '2017-10-27');
```

sales

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language

- the UPDATE statement

allows you to renew existing data of your tables

# Data Manipulation Language



SQL

**sales**

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language



SQL

```
UPDATE sales  
SET date_of_purchase = '2017-12-12'  
WHERE purchase_number = 1;
```

sales

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language



SQL

```
UPDATE sales  
SET date_of_purchase = '2017-12-12'  
WHERE purchase_number = 1;
```

sales

purchase_number	date_of_purchase
1	<b>2017-12-12</b>
2	<b>2017-10-27</b>

# Data Manipulation Language

- the DELETE statement

- functions similarly to the TRUNCATE statement

- TRUNCATE vs. DELETE

TRUNCATE allows us to remove all the records contained in a table

vs.

with DELETE, you can specify precisely what you would like to be removed

# Data Manipulation Language



SQL

```
DELETE FROM sales;
```

sales

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language



SQL

```
DELETE FROM sales;
```

```
TRUNCATE TABLE sales;
```

sales

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language



SQL

```
DELETE FROM sales;
```

```
TRUNCATE TABLE sales;
```

sales

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language



SQL

```
DELETE FROM sales  
WHERE  
    purchase_number = 1;
```

sales

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language



SQL

```
DELETE FROM sales  
WHERE  
    purchase_number = 1;
```

sales

purchase_number	date_of_purchase
1	2017-10-11
2	2017-10-27

# Data Manipulation Language

- **Data Manipulation Language (DML)**

- **SELECT... FROM...**
- **INSERT INTO... VALUES...**
- **UPDATE... SET... WHERE...**
- **DELETE FROM... WHERE...**

Next:

Data Control Language (DCL)

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of black office chairs with chrome bases, all facing towards the front of the room. The front wall is a light-colored paneling, and mounted on it are several flat-screen monitors. The ceiling is white with a grid of recessed lighting. The floor is a polished wood. The overall atmosphere is professional and spacious.

# Data Control Language (DCL)

# Data Control Language

- Data Control Language (DCL)

# Data Control Language

- Data Control Language (DCL)
- the GRANT and REVOKE statements

# Data Control Language

- Data Control Language (DCL)
- the GRANT and REVOKE statements  
allow us to manage the rights users have in a database

# Data Control Language



# Data Control Language



# Data Control Language



users

# Data Control Language

- The GRANT statement

# Data Control Language

- The GRANT statement  
gives (or grants) certain permissions to users

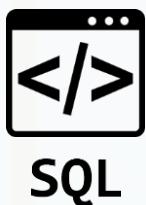
# Data Control Language

- The GRANT statement  
gives (or grants) certain permissions to users



# Data Control Language

- The GRANT statement  
gives (or grants) certain permissions to users



```
GRANT type_of_permission ON database_name.table_name TO  
'username'@'localhost'
```

# Data Control Language

- The GRANT statement  
gives (or grants) certain permissions to users
- one can grant a *specific* type of permission, like *complete* or *partial access*



SQL

```
GRANT type_of_permission ON database_name.table_name TO  
'username'@'localhost'
```

# Data Control Language

- these rights will be assigned to a person who has a *username* registered at the *local server* ('*localhost*': IP 127.0.0.1)



SQL

```
GRANT type_of_permission ON database_name.table_name TO  
'username'@'localhost'
```

# Data Control Language

- these rights will be assigned to a person who has a *username* registered at the *local server* ('*localhost*': IP 127.0.0.1)
- big companies and corporations don't use this type of server, and their databases lay on *external*, more powerful servers



SQL

```
GRANT type_of_permission ON database_name.table_name TO  
'username'@'localhost'
```

# Data Control Language

- Database administrators

# Data Control Language

- **Database administrators**

people who have *complete* rights to a database

# Data Control Language

- **Database administrators**

people who have *complete* rights to a database

- they can grant access to users and can revoke it

# Data Control Language

- **Database administrators**

people who have *complete* rights to a database

- they can grant access to users and can revoke it

- **the REVOKE clause**

# Data Control Language

- **Database administrators**

people who have *complete* rights to a database

- they can grant access to users and can revoke it

- **the REVOKE clause**

used to revoke permissions and privileges of database users

# Data Control Language

- **Database administrators**

people who have *complete* rights to a database

- they can grant access to users and can revoke it

- **the REVOKE clause**

used to revoke permissions and privileges of database users

- the exact opposite of GRANT

# Data Control Language

- the REVOKE clause

used to revoke permissions and privileges of database users



SQL

# Data Control Language

- the REVOKE clause

used to revoke permissions and privileges of database users



SQL

```
REVOKE type_of_permission ON database_name.table_name FROM  
'username'@'localhost'
```

# Data Control Language

Next:

# Data Control Language

Next:

Transaction Control Language (TCL)

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are rows of black office chairs facing towards the front. The front wall features several large, dark rectangular panels, likely projection screens or television monitors. The room has a polished wooden floor and a ceiling with a grid of recessed lighting fixtures. The overall atmosphere is professional and spacious.

# Transaction Control Language (DCL)

# Transaction Control Language

- Transaction Control Language (DCL)

# Transaction Control Language

- Transaction Control Language (DCL)

- not every change you make to a database is saved automatically

# Transaction Control Language

- Transaction Control Language (DCL)
  - not every change you make to a database is saved automatically
- the COMMIT statement

# Transaction Control Language

- Transaction Control Language (DCL)

- not every change you make to a database is saved automatically

- the COMMIT statement

- related to INSERT, DELETE, UPDATE

# Transaction Control Language

- Transaction Control Language (DCL)

- not every change you make to a database is saved automatically

- the COMMIT statement

- related to INSERT, DELETE, UPDATE
  - will save the changes you've made

# Transaction Control Language

- Transaction Control Language (DCL)

- not every change you make to a database is saved automatically

- the COMMIT statement

- related to INSERT, DELETE, UPDATE
  - will save the changes you've made
  - will let other users have access to the modified version of the database

# Transaction Control Language

## DB administrator

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator

- Change the last name of the 4<sup>th</sup> customer from ‘Winnfield’ to ‘Johnson’

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator

- Change the last name of the 4<sup>th</sup> customer from ‘Winnfield’ to ‘Johnson’

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine		<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator

- Change the last name of the 4<sup>th</sup> customer from ‘Winnfield’ to ‘Johnson’

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator



SQL

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator



SQL

```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4;
```

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator



SQL

```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4;
```

Customers					
customer_id	first_name	last_name		email_address	number_of_complaints
1	John	McKinley		<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0
2	Elizabeth	McFarlane		<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2
3	Kevin	Lawrence		<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1
4	Catherine			<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0

# Transaction Control Language

## DB administrator



SQL

```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4;
```

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator

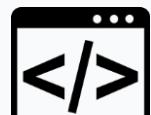
Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Problem: users

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator



SQL

```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4;
```

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator



```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4  
COMMIT;
```

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

## users

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

- the COMMIT statement

# Transaction Control Language

- the COMMIT statement

committed states can accrue

# Transaction Control Language

- **the COMMIT statement**

committed states can accrue

- **the ROLLBACK clause**

# Transaction Control Language

- **the COMMIT statement**

committed states can accrue

- **the ROLLBACK clause**

the clause that will let you make a step back

# Transaction Control Language

- **the COMMIT statement**

committed states can accrue

- **the ROLLBACK clause**

the clause that will let you make a step back

- allows you to undo any changes you have made but don't want to be saved permanently

# Transaction Control Language

## DB administrator



SQL

```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4  
COMMIT;
```

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator



SQL

```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4  
COMMIT;  
  
ROLLBACK;
```

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Johnson	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

## DB administrator



SQL

```
UPDATE customers  
SET last_name = 'Johnson'  
WHERE customer_id = 4  
COMMIT;
```

```
ROLLBACK;
```

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Transaction Control Language

- the COMMIT statement

# Transaction Control Language

- the COMMIT statement
  - saves the transaction in the database

# Transaction Control Language

- **the COMMIT statement**

- saves the transaction in the database
- changes cannot be undone

# Transaction Control Language

- the COMMIT statement
  - saves the transaction in the database
  - changes cannot be undone
- the ROLLBACK clause

# Transaction Control Language

- **the COMMIT statement**

- saves the transaction in the database
  - changes cannot be undone

- **the ROLLBACK clause**

- allows you to take a step back

# Transaction Control Language

- **the COMMIT statement**

- saves the transaction in the database
  - changes cannot be undone

- **the ROLLBACK clause**

- allows you to take a step back
  - the last change(s) made will not count

# Transaction Control Language

- **the COMMIT statement**

- saves the transaction in the database
- changes cannot be undone

- **the ROLLBACK clause**

- allows you to take a step back
- the last change(s) made will not count
- reverts to the last non-committed state

# SQL Syntax

- DDL - Data Definition Language
- DML - Data Manipulation Language
- DCL - Data Control Language
- TCL - Transaction Control Language

# SQL Syntax

- **DDL - Data Definition Language**

creation of data

- **DML - Data Manipulation Language**

manipulation of data

- **DCL - Data Control Language**

assignment and removal of permissions to use this data

- **TCL - Transaction Control Language**

saving and restoring changes to a database

Next:

Next:

More about Database Theory



# Basic Database Terminology



# Relational Schemas: Primary Key

# Relational Schemas: Primary Key

Sales			
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# Relational Schemas: Primary Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# Relational Schemas: Primary Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

Items						
item_code	item	unit_price_usd	company_id	company	headquarters	phone_number
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

# Relational Schemas: Primary Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Items						
item_code	item	unit_price_usd	company_id	company	headquarters	phone_number
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

Companies		
company	Headquarters_Phone_number	company_id
Company A	+1 (202) 555-0196	1
Company B	+1 (202) 555-0152	2
Company C	+1 (229) 853-9913	3
Company D	+1 (618) 369-7392	4

# Relational Schemas: Primary Key

- these tables have a tabular shape

# Relational Schemas: Primary Key

- these tables have a tabular form
- a relational schema can be applied to represent them

# Relational Schemas: Primary Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Items						
item_code	item	unit_price_usd	company_id	company	headquarters	phone_number
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

Companies		
company	Headquarters_Phone_number	company_id
Company A	+1 (202) 555-0196	1
Company B	+1 (202) 555-0152	2
Company C	+1 (229) 853-9913	3
Company D	+1 (618) 369-7392	4

# Relational Schemas: Primary Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

## sales per customer of our company

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Items					
item_code	item	unit_price_usd	company_id	company	headquarters_phone_number
A_1	Lamp	20	1	Company A	+1 (202) 555-0196
A_2	Desk	250	1	Company A	+1 (202) 555-0196
B_1	Lamp	30	2	Company B	+1 (202) 555-0152
B_2	Desk	350	2	Company B	+1 (202) 555-0152
C_1	Chair	150	3	Company C	+1 (229) 853-9913
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392

Companies		
company	Headquarters_Phone_number	company_id
Company A	+1 (202) 555-0196	1
Company B	+1 (202) 555-0152	2
Company C	+1 (229) 853-9913	3
Company D	+1 (618) 369-7392	4

# Relational Schemas: Primary Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

the dates of a few purchases may coincide

# Relational Schemas: Primary Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

the ID of a customer may appear a few times

# Relational Schemas: Primary Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

there is a possibility to see the same item code a few times

# Relational Schemas: Primary Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

all the numbers in this column will be different

# Relational Schemas: Primary Key

## primary key

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

# Relational Schemas: Primary Key

- Primary Key

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each table can have one and only one primary key

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each **table** can have one and only one primary key
- in one **table**, you cannot have 3 or 4 primary keys

# Relational Schemas: Primary Key

## Primary Key

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each table can have one and only one primary key
- in one table, you cannot have 3 or 4 primary keys

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# Relational Schemas: Primary Key

- **Primary Key**

may be composed of a set of columns

# Relational Schemas: Primary Key

- Primary Key

may be composed of a set of columns

e.g. “purchase\_number” + “date\_of\_purchase”

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	9/3/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

- Primary Key

may be composed of a set of columns

e.g. “purchase\_number” + “date\_of\_purchase”

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
1	9/3/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

- Primary Key

one-column primary key = all purchases will be recorded under a different number

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each **table** can have one and only one primary key
- in one **table**, you cannot have 3 or 4 primary keys

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each table can have one and only one primary key
- in one table, you cannot have 3 or 4 primary keys
- primary keys are the unique identifiers of a table

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each table can have one and only one primary key
- in one table, you cannot have 3 or 4 primary keys
- primary keys are the unique identifiers of a table
- cannot contain null values!

# Relational Schemas: Primary Key

- Primary Key

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

- Primary Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

# Relational Schemas: Primary Key

Sales

purchase number

date\_of\_purchase

customer\_id (FK)

item\_code (FK)

# Relational Schemas: Primary Key

Sales

purchase\_number

date\_of\_purchase

customer\_id (FK)

item\_code (FK)

Table name:

Sales

Primary key:

purchase\_number

Other fields:

date\_of\_purchase, customer\_id, item\_code

# Relational Schemas: Primary Key

Sales

<u>purchase number</u>
date_of_purchase
customer_id (FK)
item_code (FK)

=

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each table can have one and only one primary key
- in one table, you cannot have 3 or 4 primary keys
- primary keys are the unique identifiers of a table
- cannot contain null values!

# Relational Schemas: Primary Key

- **Primary Key**

a column (or a set of columns) whose value exists and is unique for every record in a table is called a **primary key**

- each table can have one and only one primary key
- in one table, you cannot have 3 or 4 primary keys
- primary keys are the unique identifiers of a table
- cannot contain null values!
- not all tables you work with will have a primary key

# Relational Schemas: Primary Key

## relational schema

Sales

purchase number

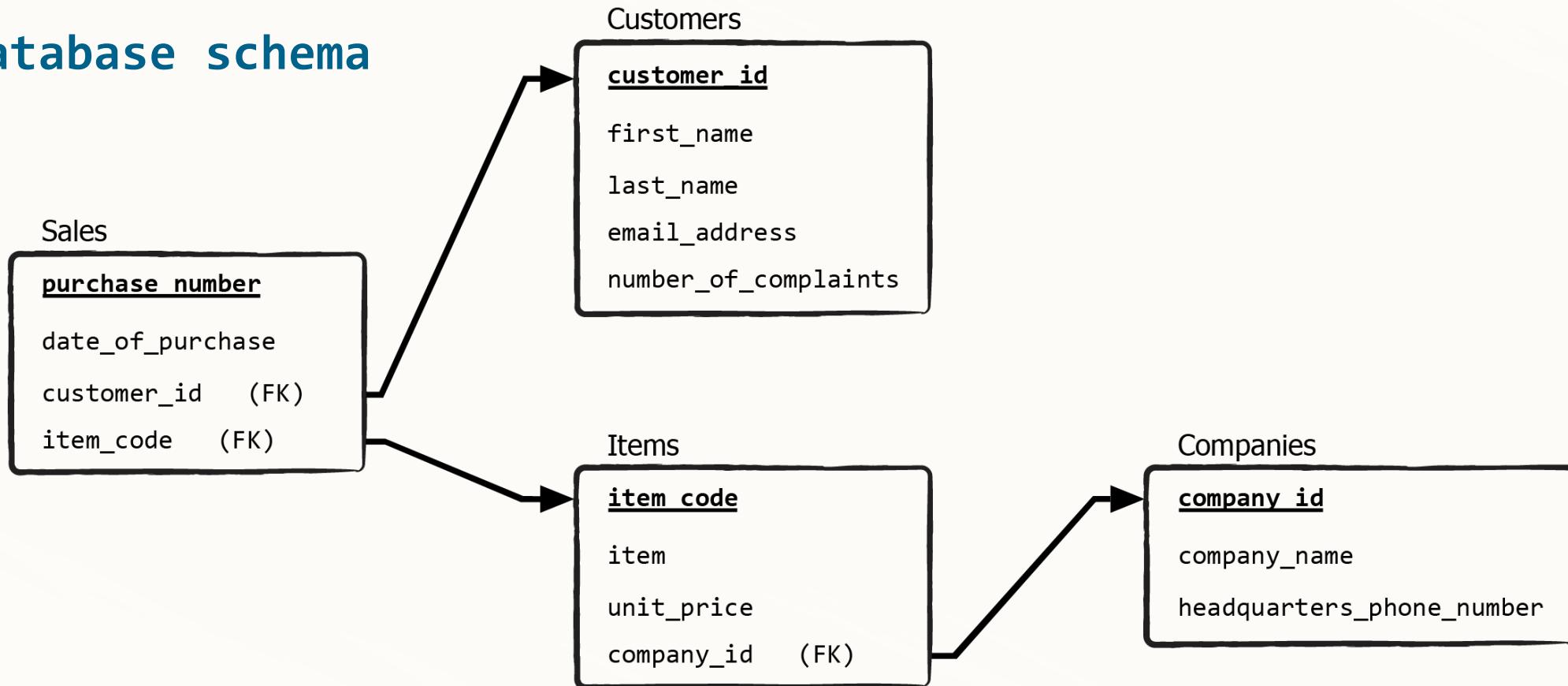
date\_of\_purchase

customer\_id (FK)

item\_code (FK)

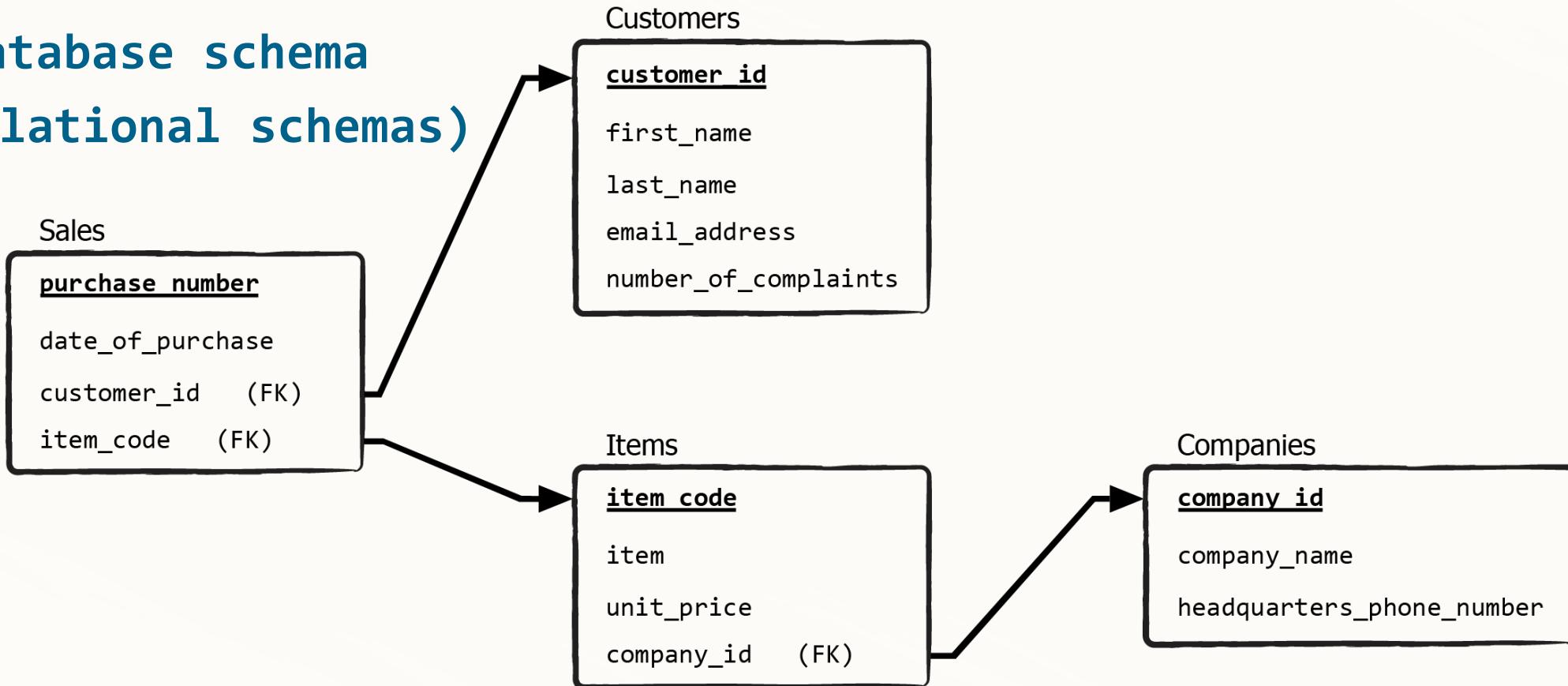
# Relational Schemas: Primary Key

## database schema



# Relational Schemas: Primary Key

database schema  
(relational schemas)





# Relational Schemas: Foreign Key

# Relational Schemas: Foreign Key

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales				
purchase_number	date_of_purchase	ID	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Relational Schemas: Foreign Key

# Relational Schemas: Foreign Key

Sales

**purchase number**

date\_of\_purchase

customer\_id

item\_code

Customers

**customer\_id**

first\_name

last\_name

email\_address

number\_of\_complaints

# Relational Schemas: Foreign Key

Sales

**purchase number**  
date\_of\_purchase  
customer\_id  
item\_code

Customers

**customer\_id**  
first\_name  
last\_name  
email\_address  
number\_of\_complaints

**always look for the foreign keys, as they show us where the relations are**

# Relational Schemas: Foreign Key

Sales

**purchase number**

date\_of\_purchase

customer\_id (FK)

item\_code

Customers

**customer\_id**

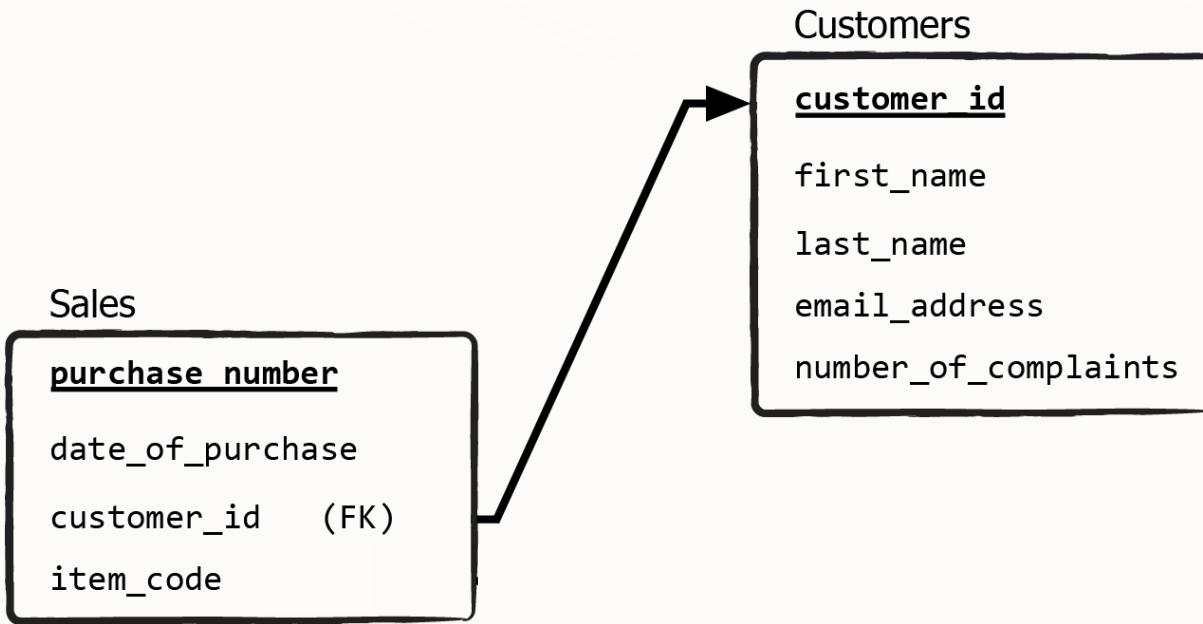
first\_name

last\_name

email\_address

number\_of\_complaints

# Relational Schemas: Foreign Key



# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017		A_1
10	8/11/2017	2	B_1

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

no repeating and missing values

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017		A_1
10	8/11/2017	2	B_1

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

no repeating and missing values  
(unique values only)

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017		A_1
10	8/11/2017	2	B_1

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

no repeating and missing values  
(unique values only)

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017		A_1
10	8/11/2017	2	B_1

repeating and missing values

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017		A_1
10	8/11/2017	2	B_1

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017		A_1
10	8/11/2017	2	B_1

ERROR

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Items						
item_code	item	unit_price_usd	company_id	company	headquarters_phone_number	
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Items						
item_code	item	unit_price_usd	company_id	company	headquarters_phone_number	
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Items						
item_code	item	unit_price_usd	company_id	company	headquarters_phone_number	
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Items						
item_code	item	unit_price_usd	company_id	company	headquarters_phone_number	
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

no repeating and missing values

# Relational Schemas: Foreign Key

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

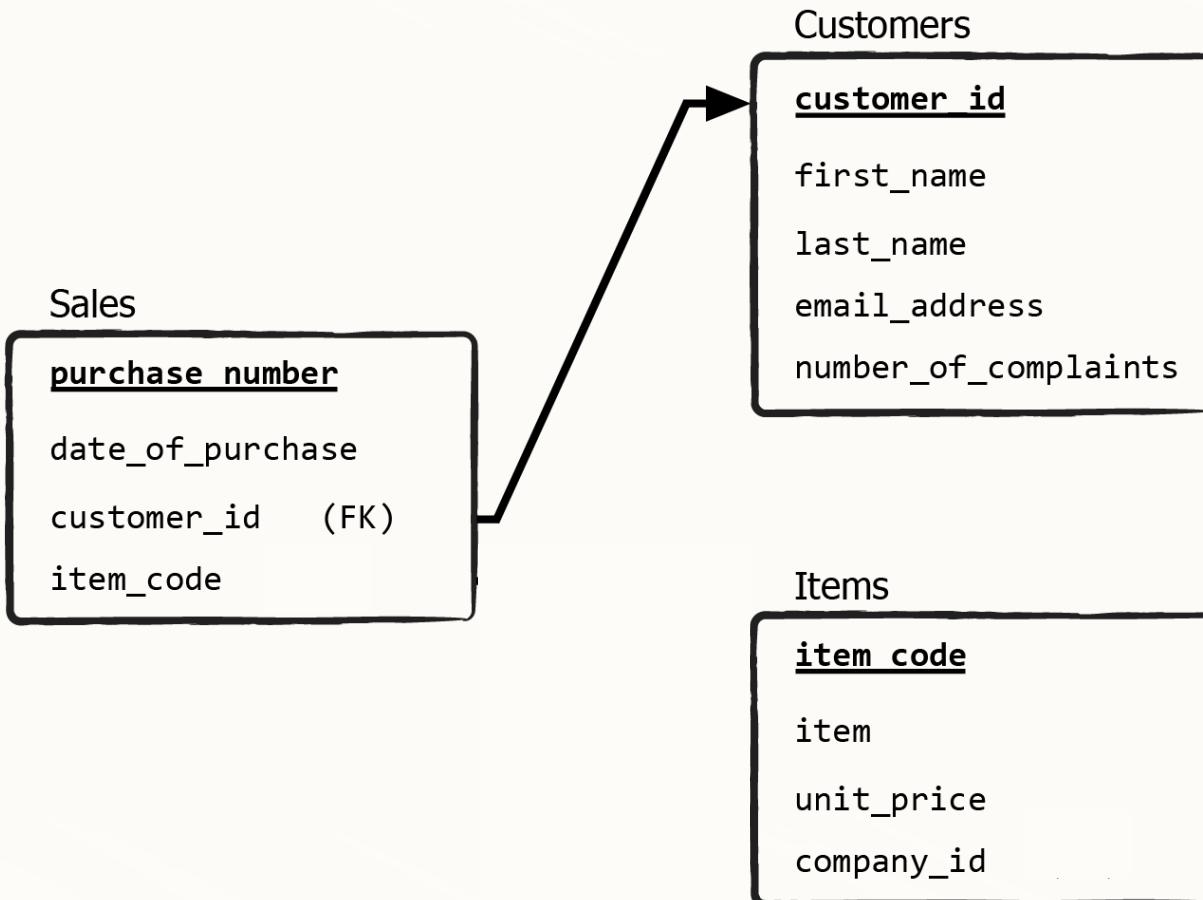
Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

repeating and missing values

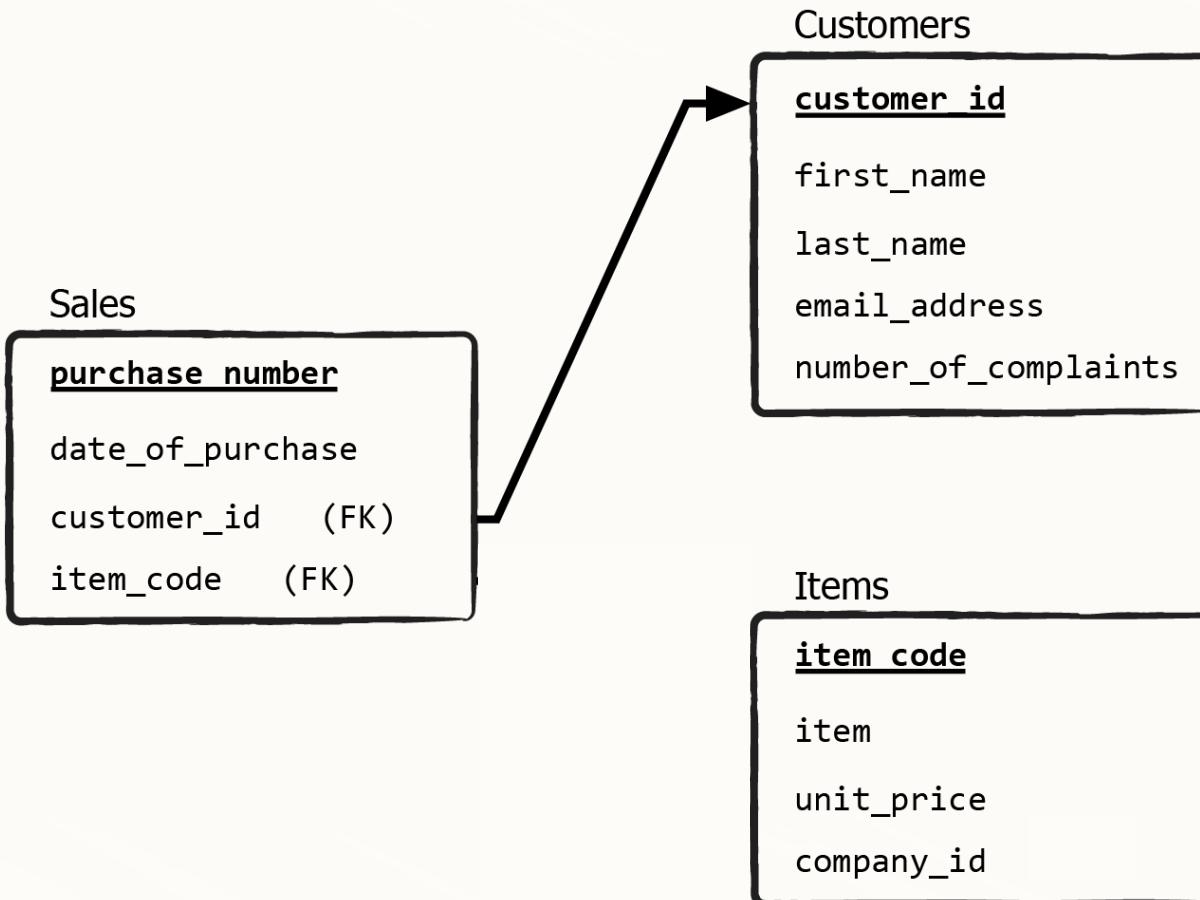
Items						
item_code	item	unit_price_usd	company_id	company	headquarters_phone_number	
A_1	Lamp	20	1	Company A	+1 (202) 555-0196	
A_2	Desk	250	1	Company A	+1 (202) 555-0196	
B_1	Lamp	30	2	Company B	+1 (202) 555-0152	
B_2	Desk	350	2	Company B	+1 (202) 555-0152	
C_1	Chair	150	3	Company C	+1 (229) 853-9913	
D_1	Loudspeakers	400	4	Company D	+1 (618) 369-7392	

no repeating and missing values

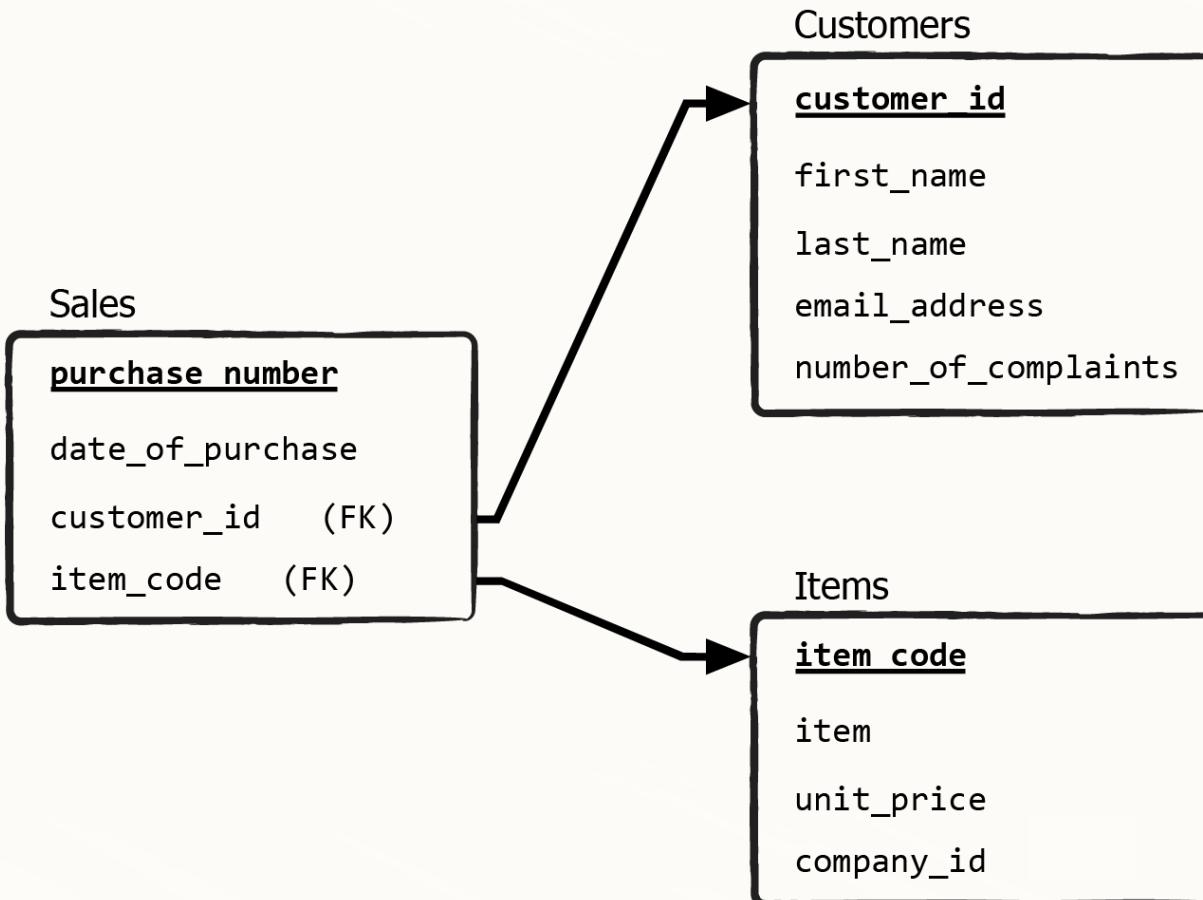
# Relational Schemas: Foreign Key



# Relational Schemas: Foreign Key



# Relational Schemas: Foreign Key

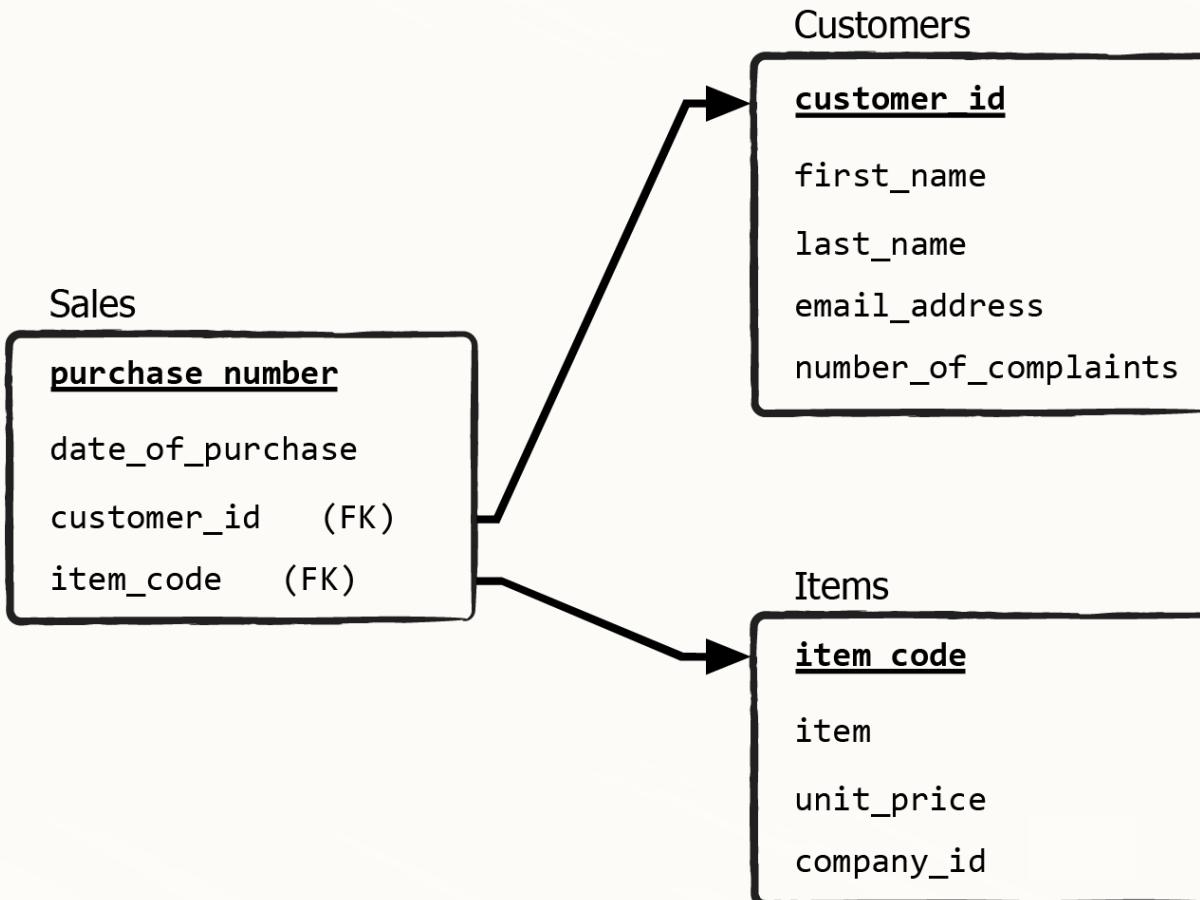


# Relational Schemas: Foreign Key

**foreign key**

identifies the relationships between tables, not the tables themselves

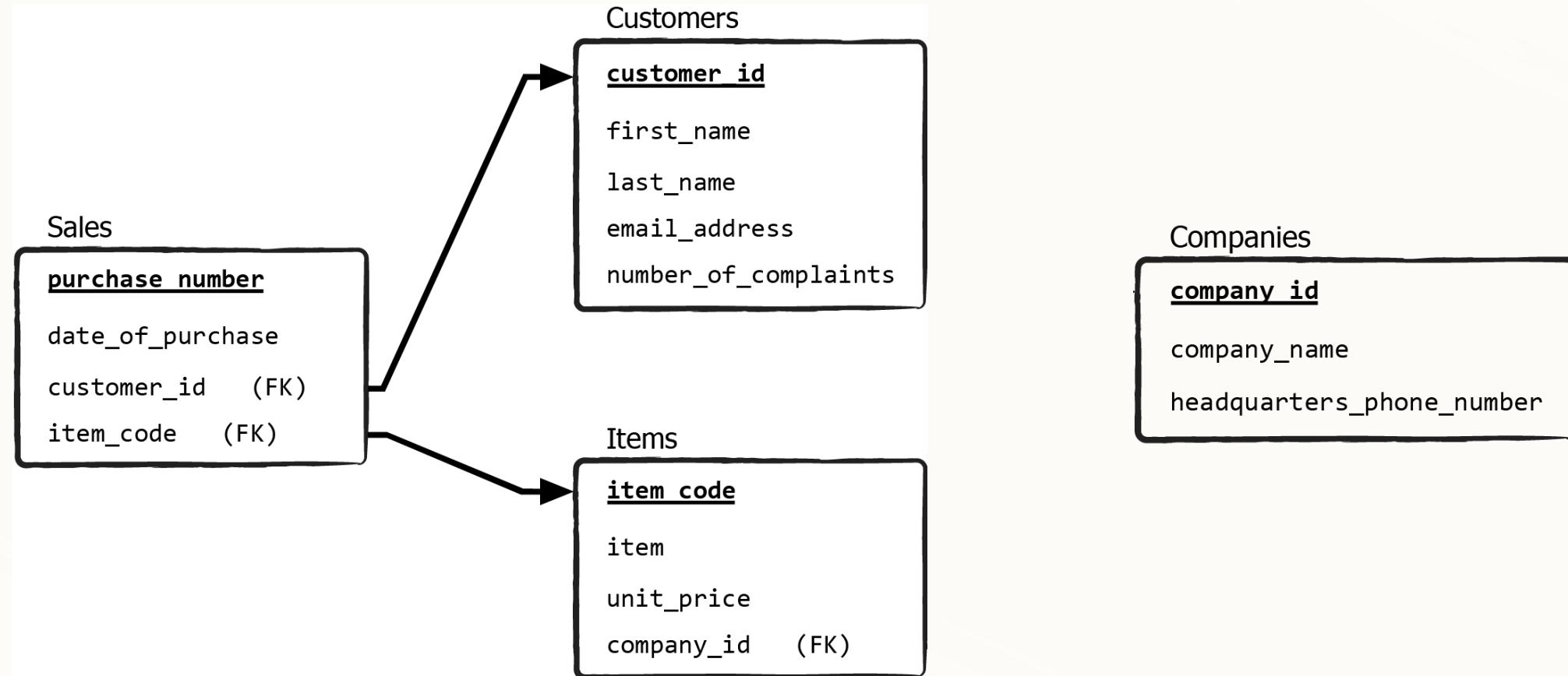
# Relational Schemas: Foreign Key



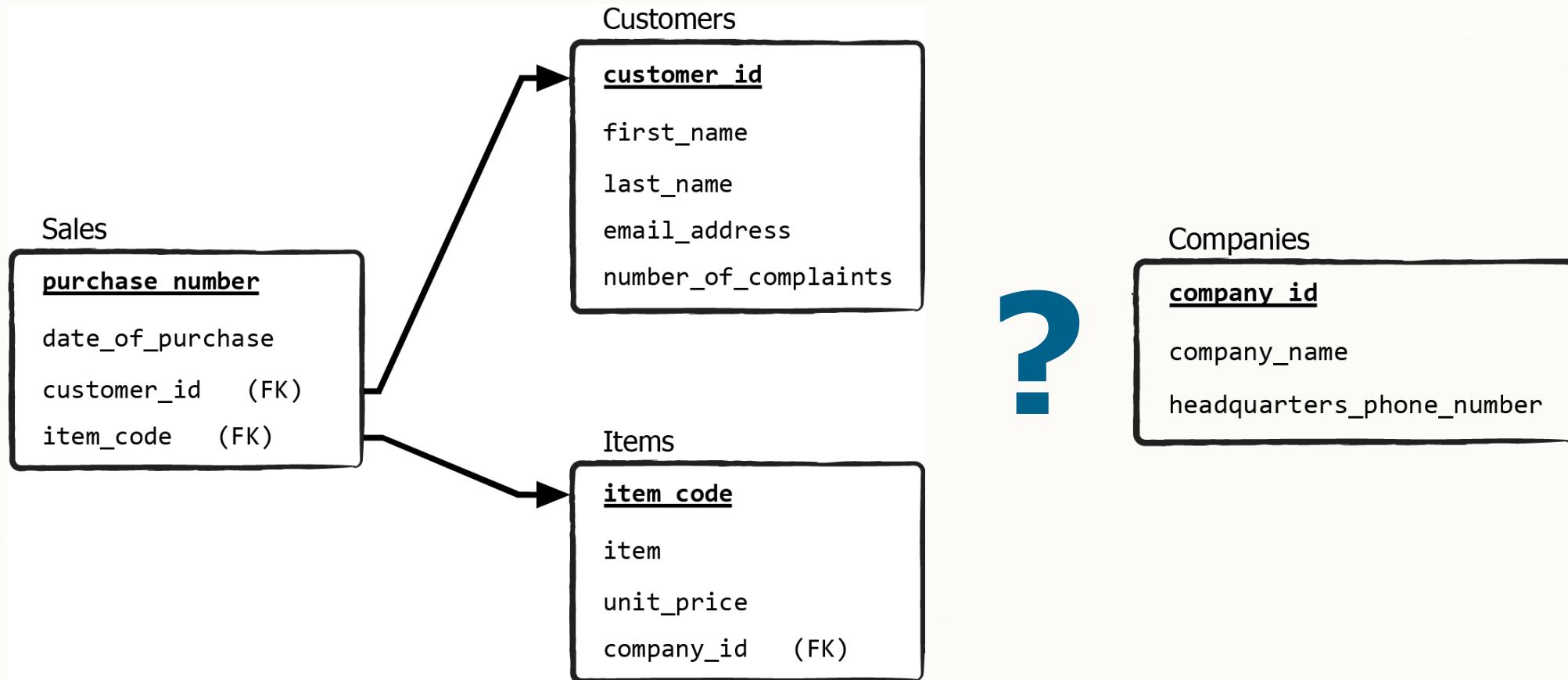


# Relational Schemas: Unique Key & Null Values

# Relational Schemas: Unique Key & Null Values



# Relational Schemas: Unique Key & Null Values



# Relational Schemas: Unique Key & Null Values

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

# Relational Schemas: Unique Key & Null Values

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

You can have two or more companies with the same name

# Relational Schemas: Unique Key & Null Values

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

# Relational Schemas: Unique Key & Null Values

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

You *cannot* have two US numbers that are completely identical

# Relational Schemas: Unique Key & Null Values

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

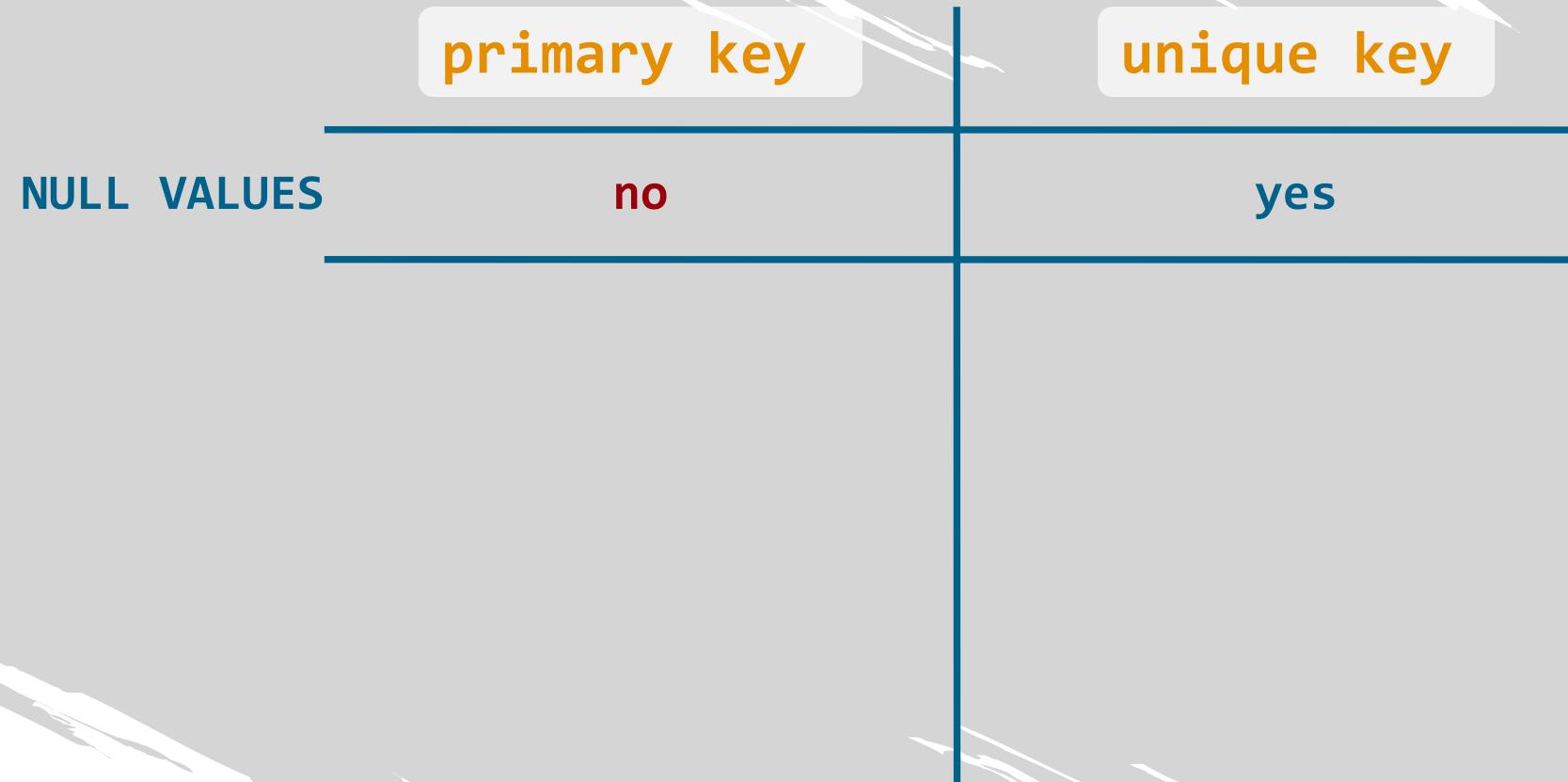
?

# Relational Schemas: Unique Key & Null Values

## unique key

used whenever you would like to specify that you don't want to see duplicate data in a given field

# Relational Schemas: Unique Key & Null Values



# Relational Schemas: Unique Key & Null Values

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4		Company D

# Relational Schemas: Unique Key & Null Values

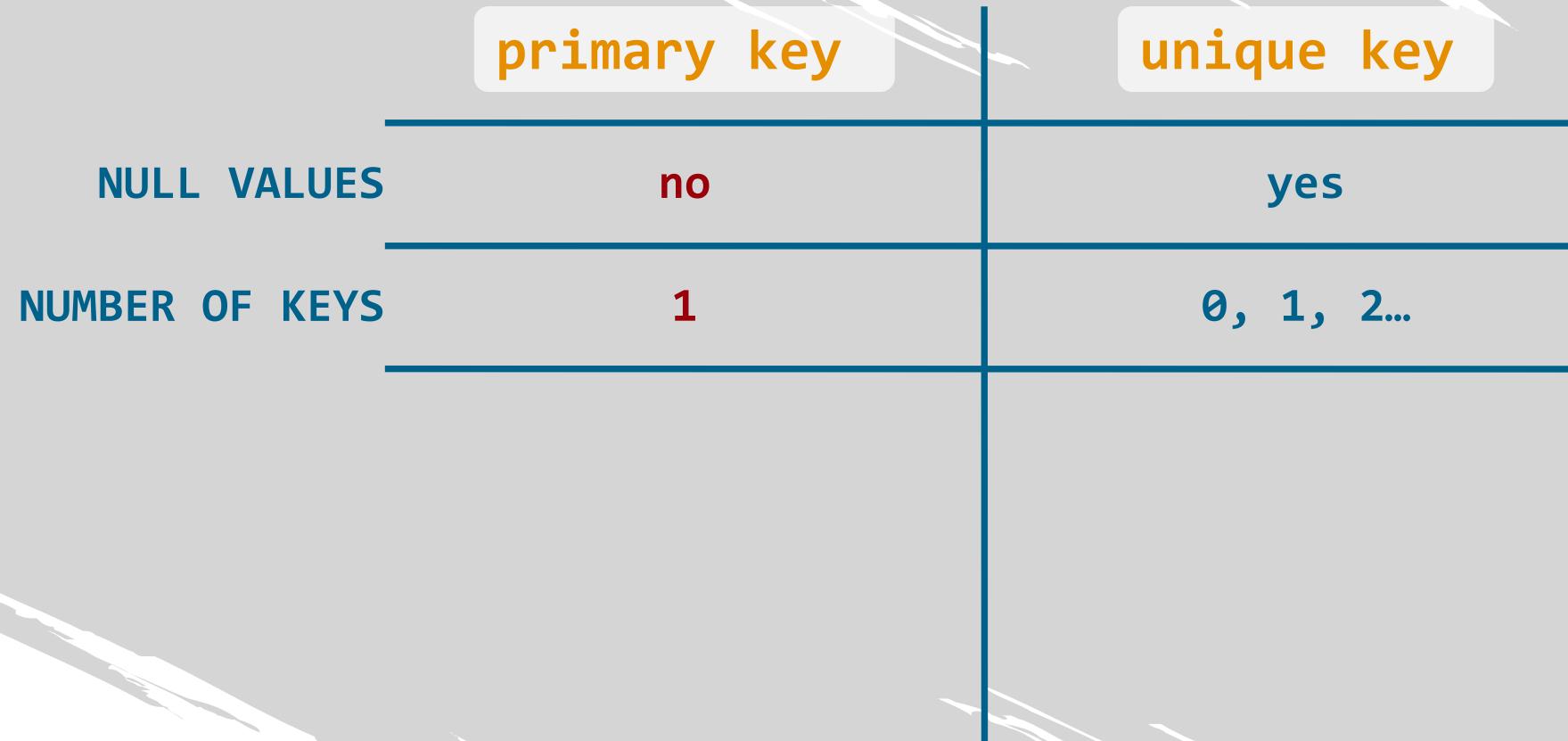
Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

# Relational Schemas: Unique Key & Null Values

ERROR

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

# Relational Schemas: Unique Key & Null Values



# Relational Schemas: Unique Key & Null Values

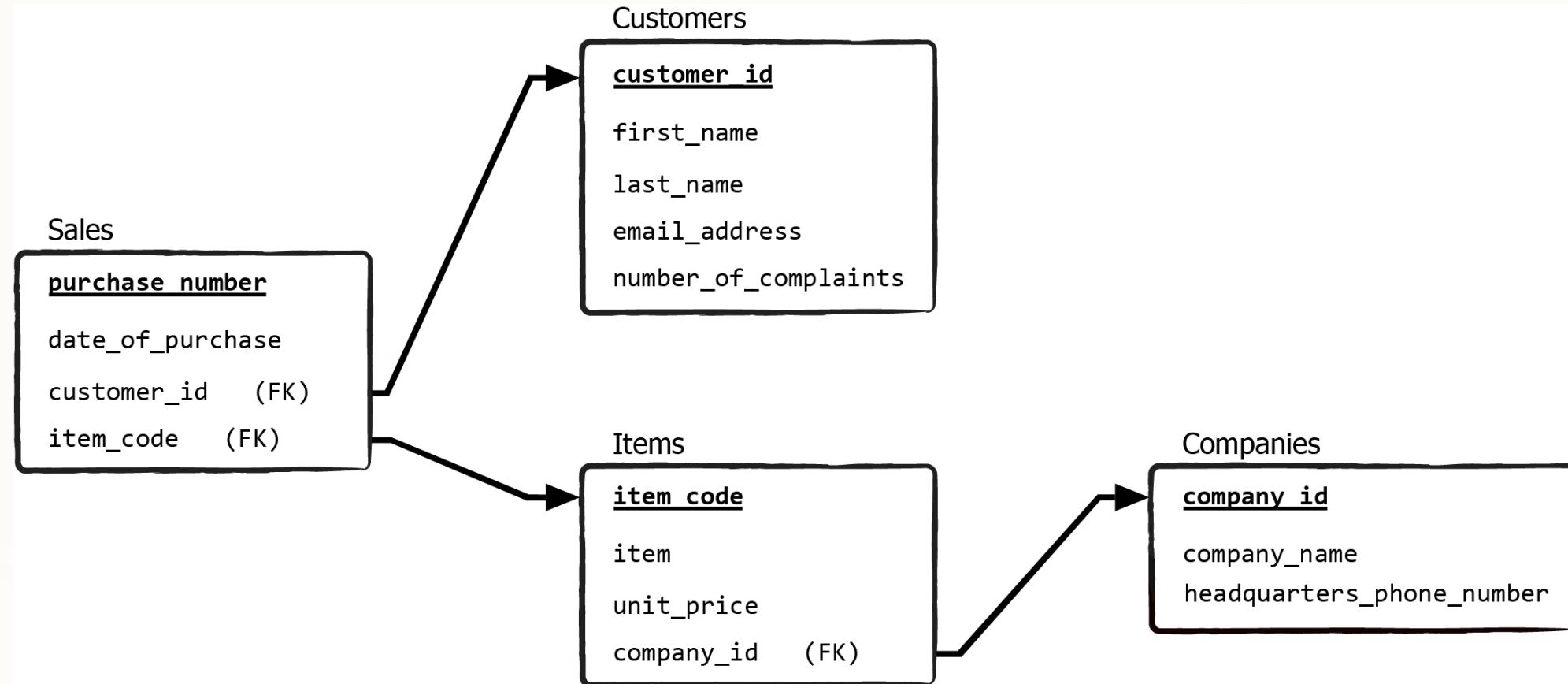
	primary key	unique key
NULL VALUES	no	yes
NUMBER OF KEYS	1	0, 1, 2...
APPLICATION TO MULTIPLE COLUMNS	yes	yes

# Relational Schemas: Unique Key & Null Values

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

# Relational Schemas: Unique Key & Null Values

# Relational Schemas: Unique Key & Null Values



A large, modern conference room is shown from a wide-angle perspective. The room features a long, dark rectangular table positioned in the center. On either side of the table, there are two rows of black office chairs with chrome legs, all facing towards the center. The room has a high ceiling with a grid of recessed lighting. Large windows line the back wall, offering a view of a flat, open landscape outside. The overall atmosphere is professional and spacious.

# Relationships

# Relationships

- Relationships

# Relationships

- **Relationships**

relationships tell you how much of the data from a foreign key field can be seen in the primary key column of the table the data is related to and vice versa

# Relationships

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Relationships

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# Relationships

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

unique values

# Relationships

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

unique values

# Relationships

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

repeated values

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

unique values

# Relationships

Sales			
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

repeated values

Customers					
<u>customer_id</u>	<u>first_name</u>	<u>last_name</u>	<u>email_address</u>	<u>number_of_complaints</u>	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

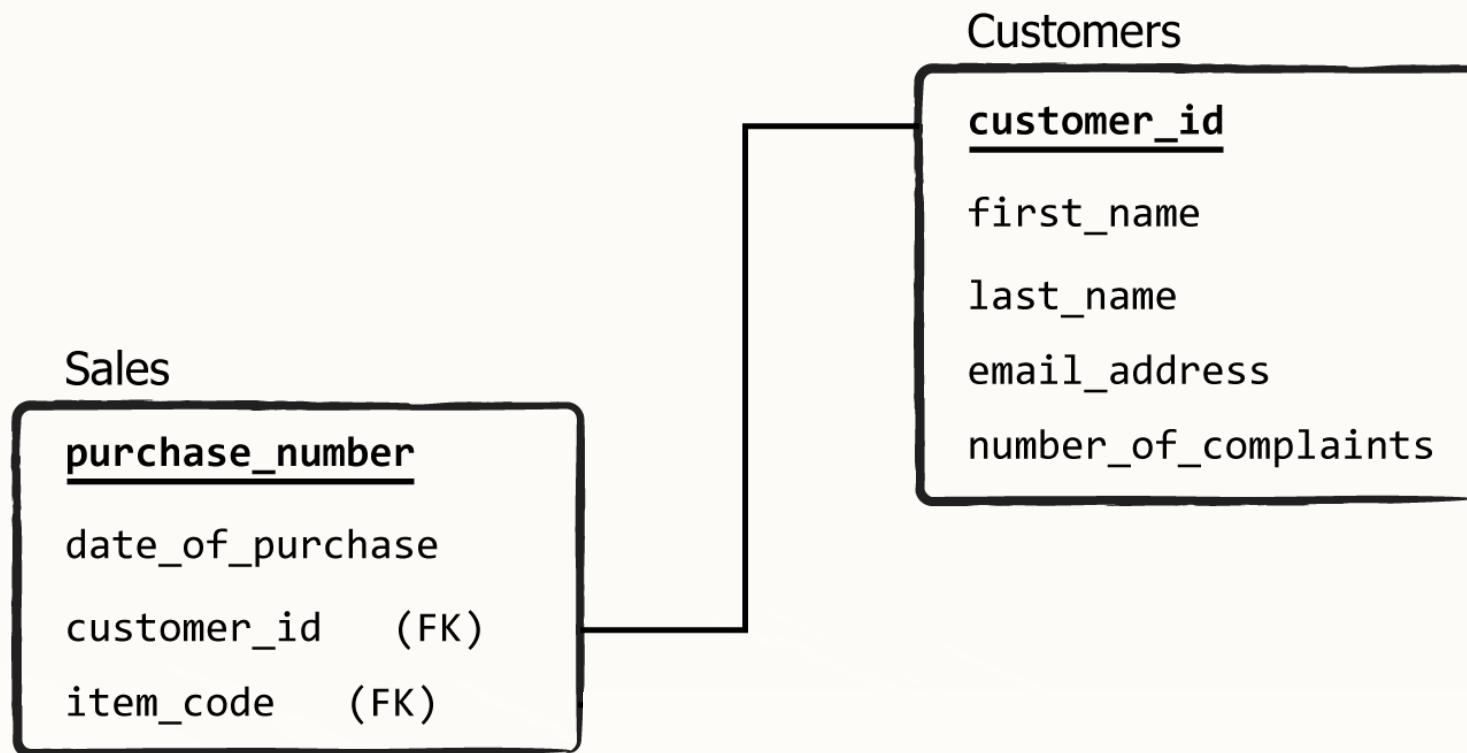
unique values

one-to-many type of relationship

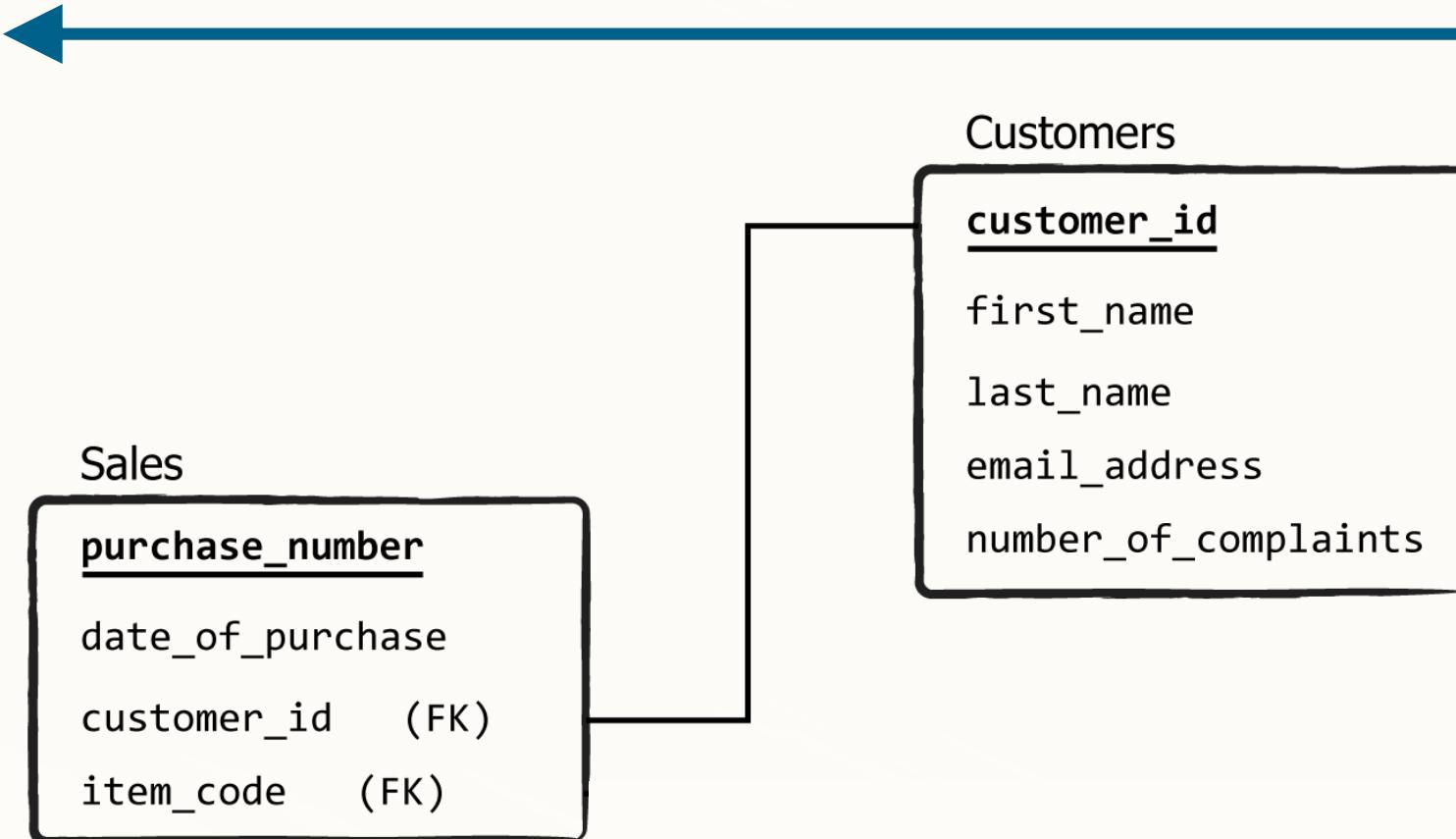
one value from the *customer\_id* column under the “Customers” table can be found many times in the *customer\_id* column in the “Sales” table.

# Relationships

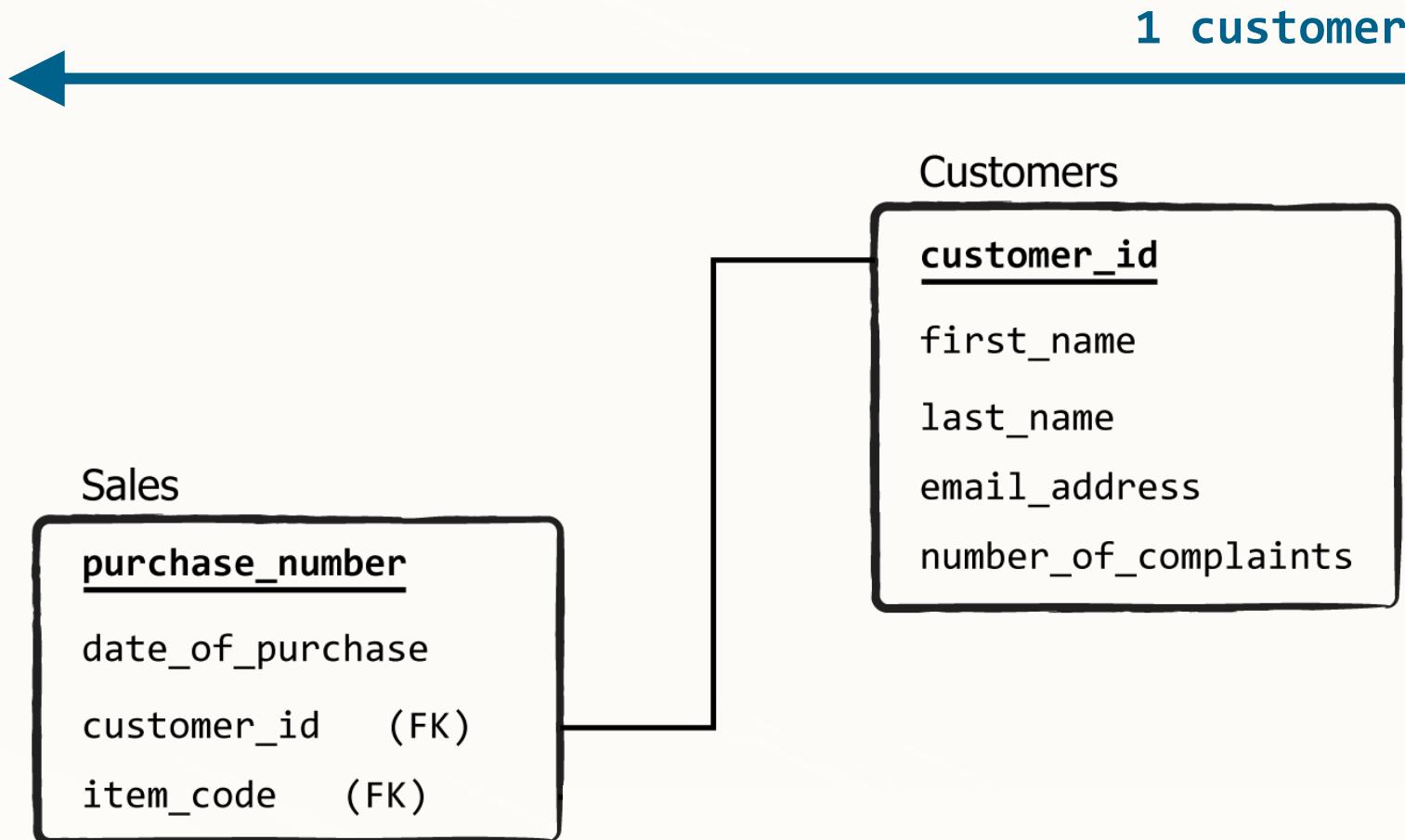
# Relationships



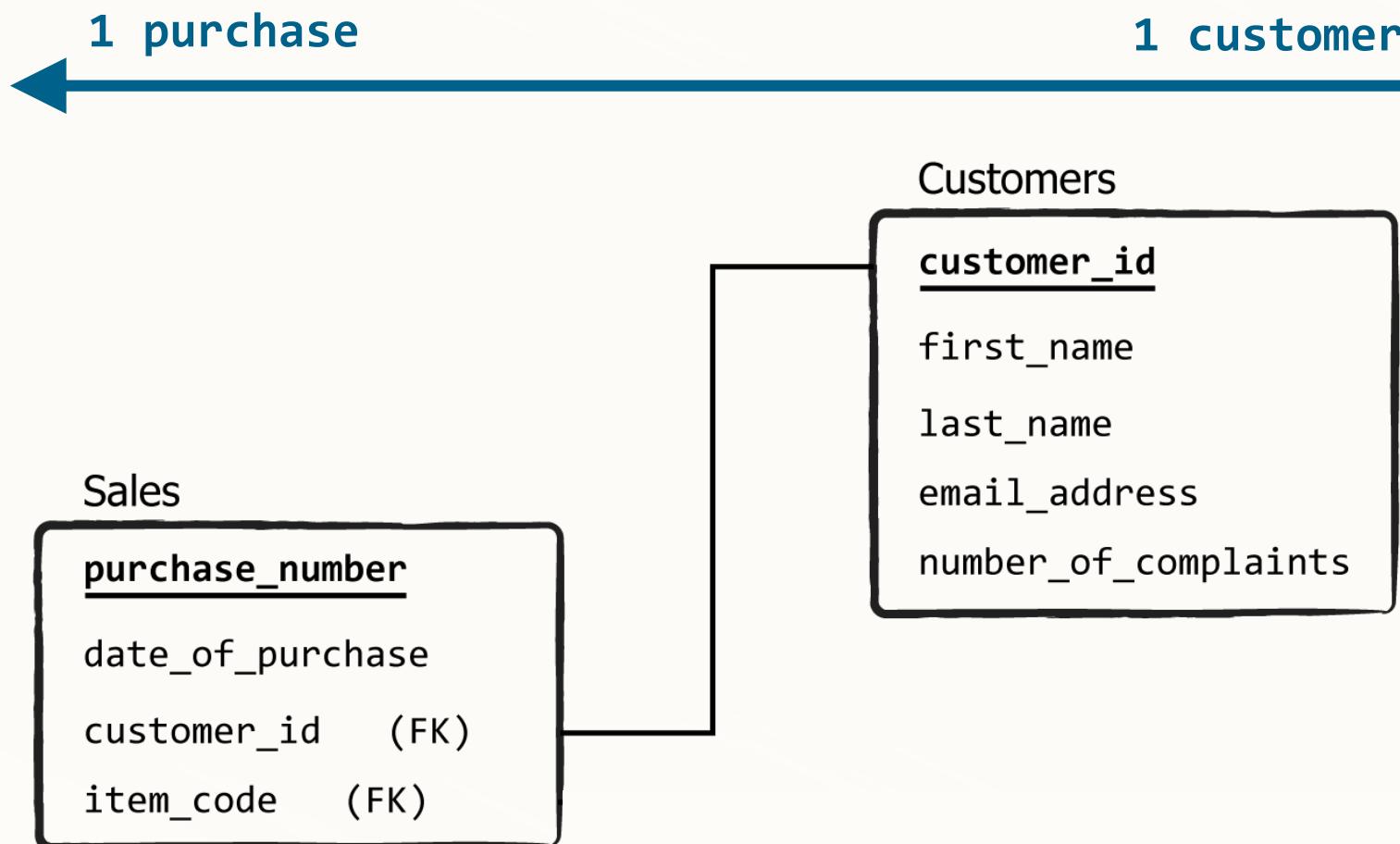
# Relationships



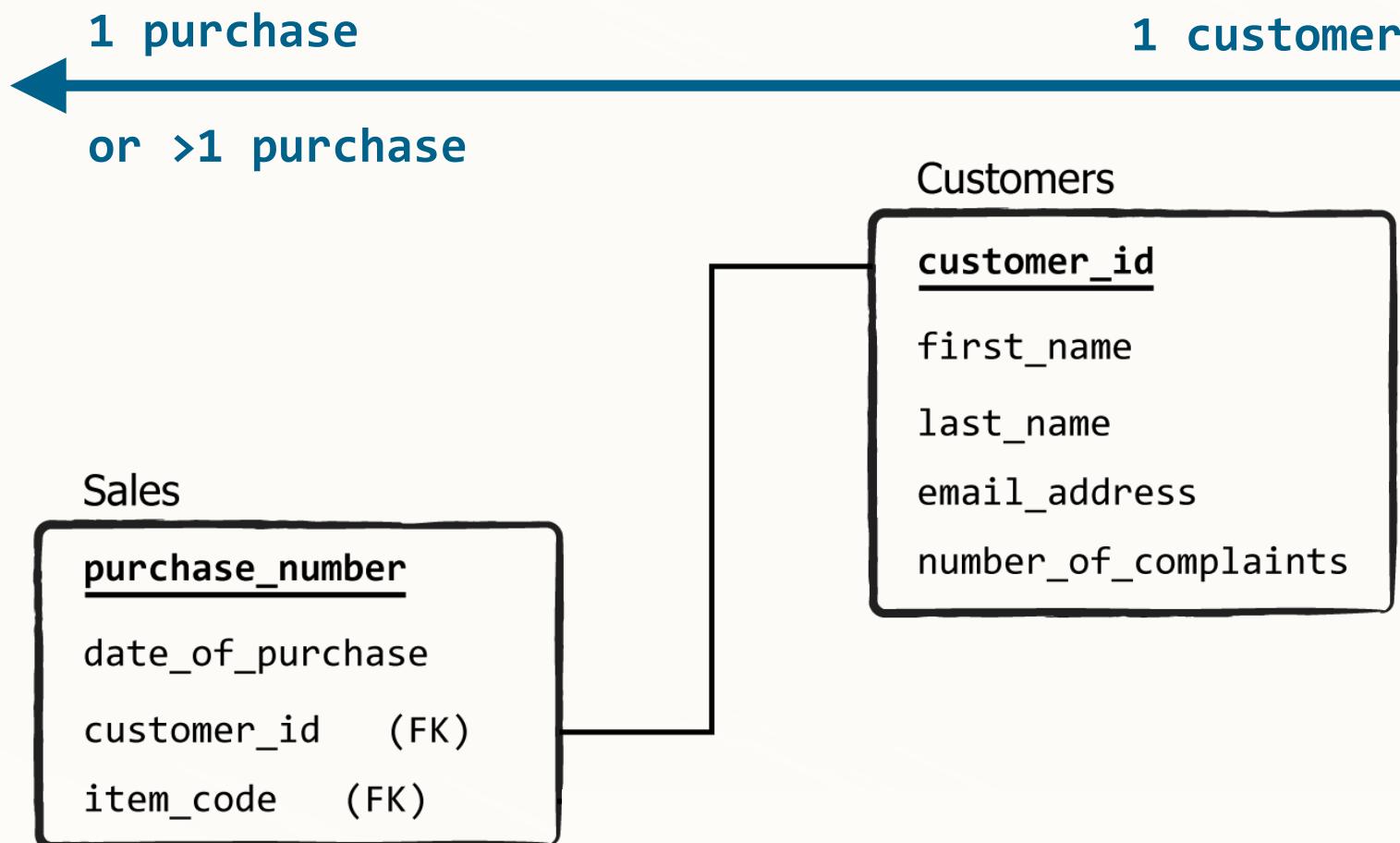
# Relationships



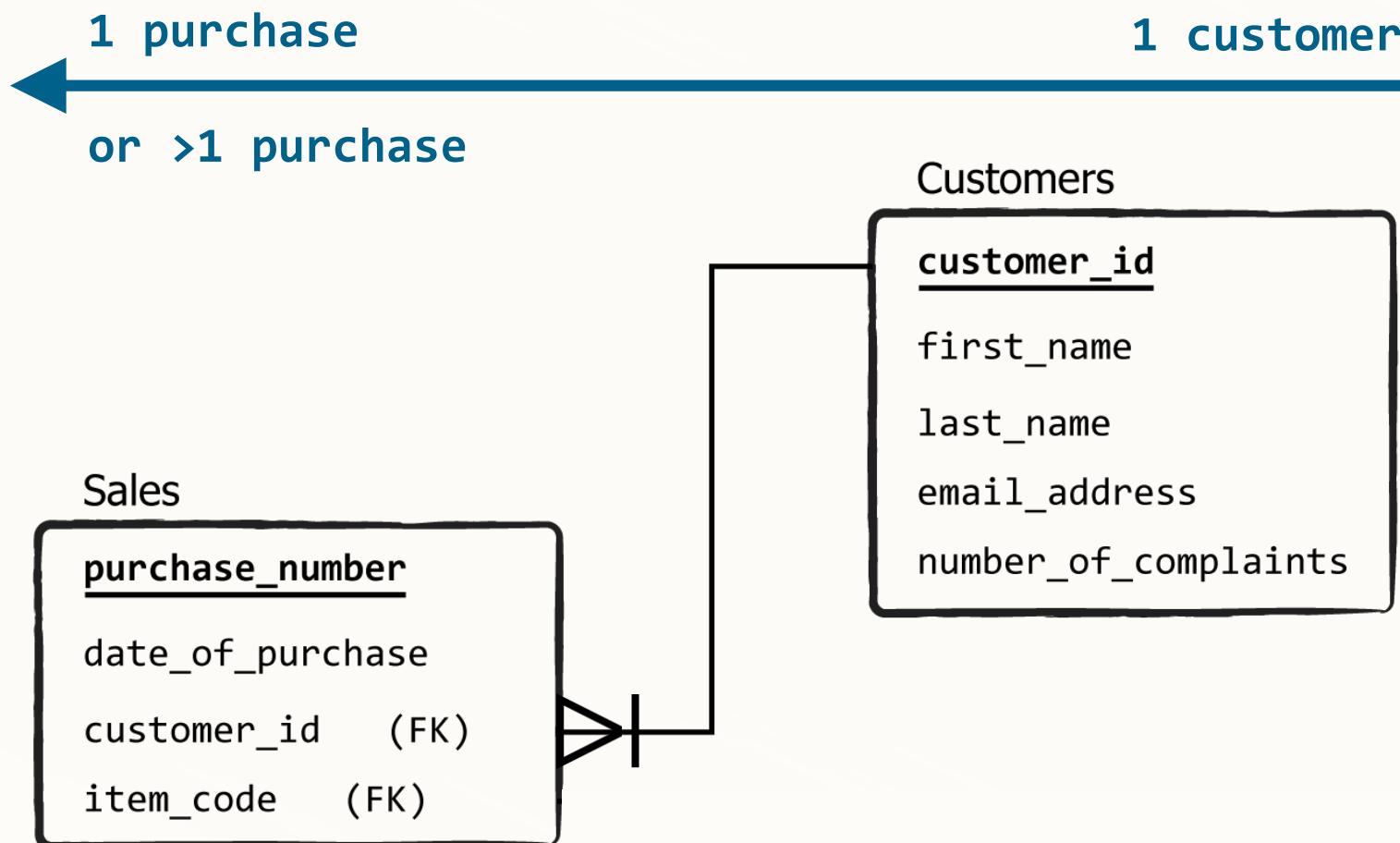
# Relationships



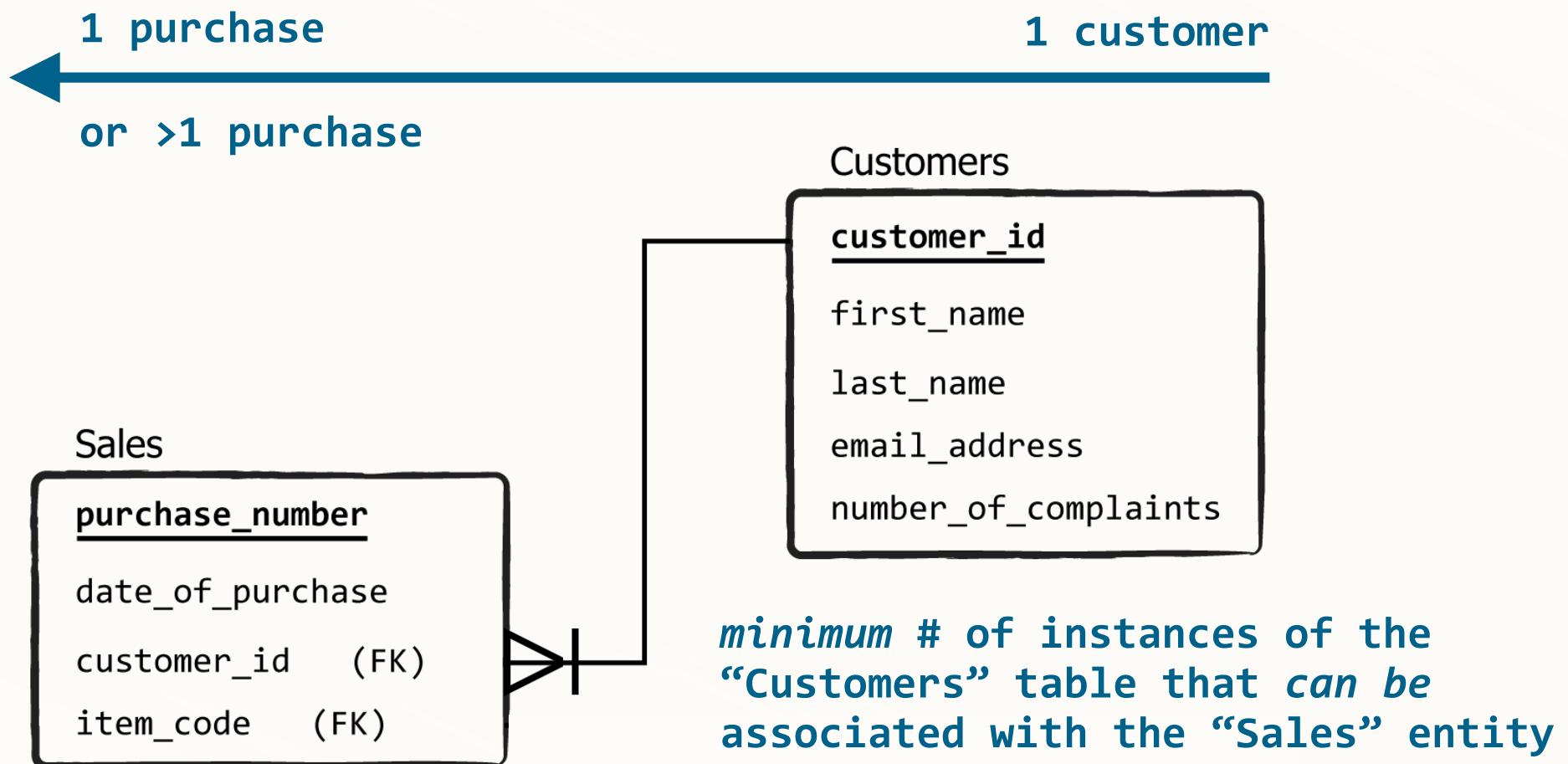
# Relationships



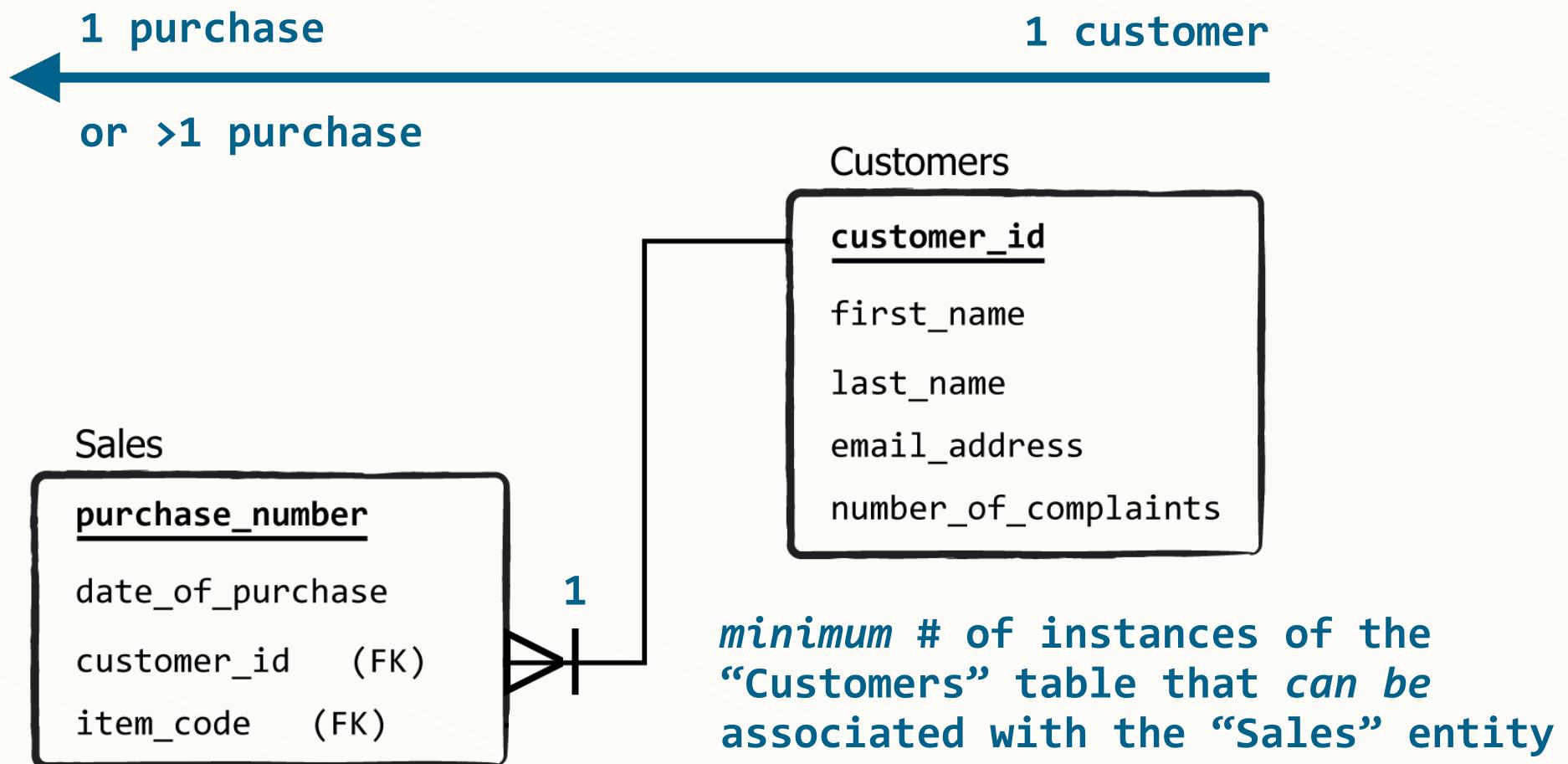
# Relationships



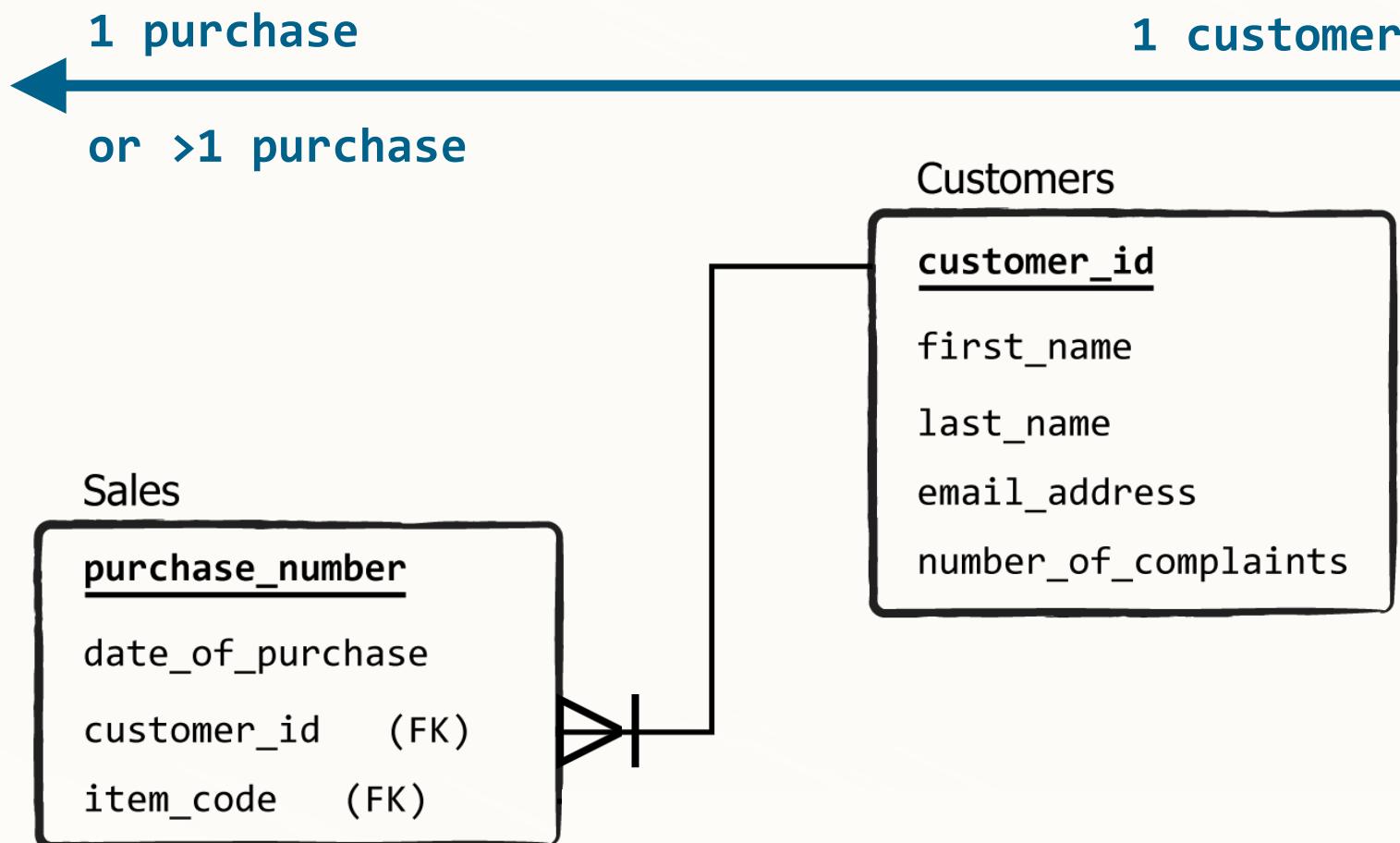
# Relationships



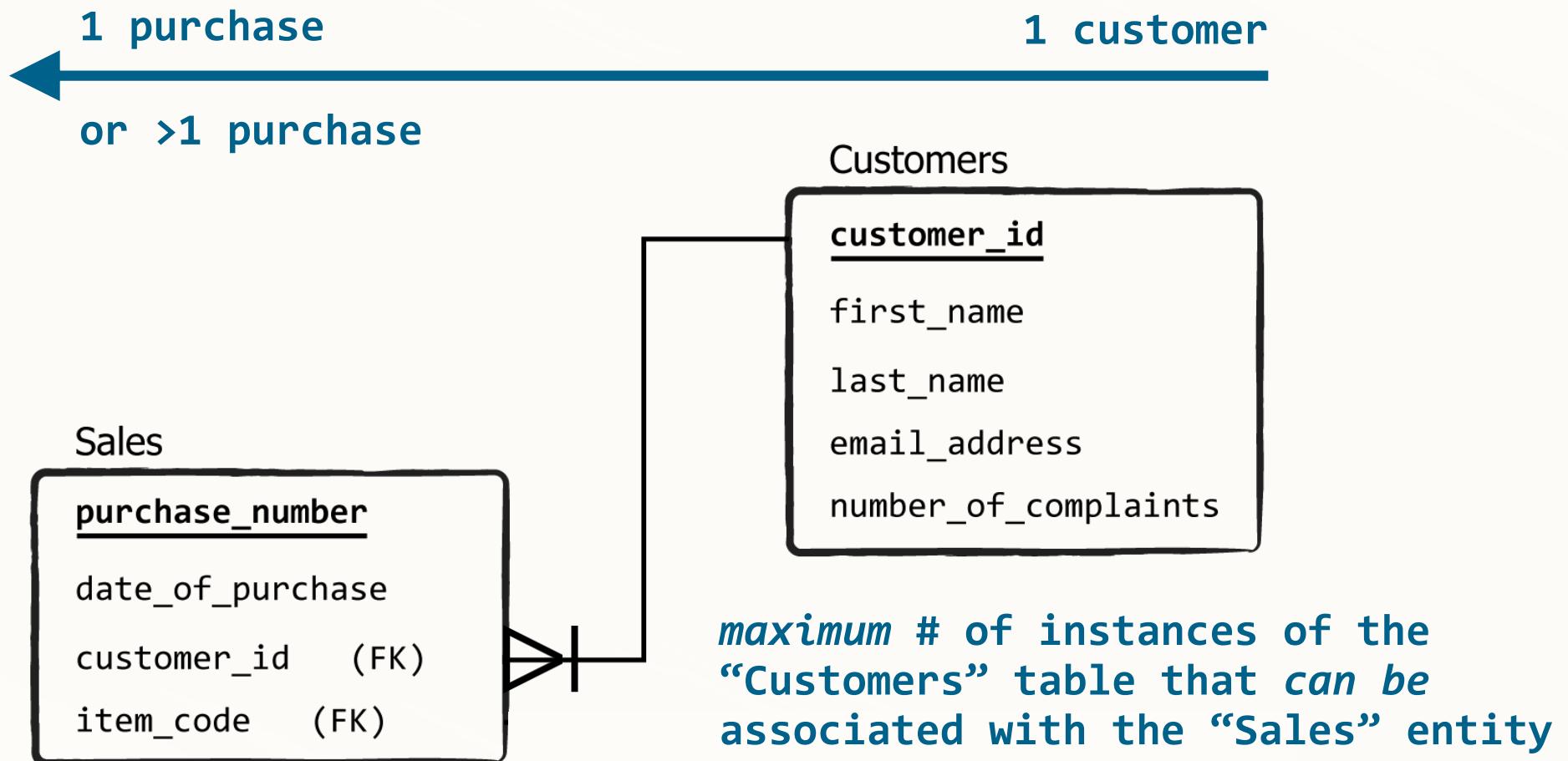
# Relationships



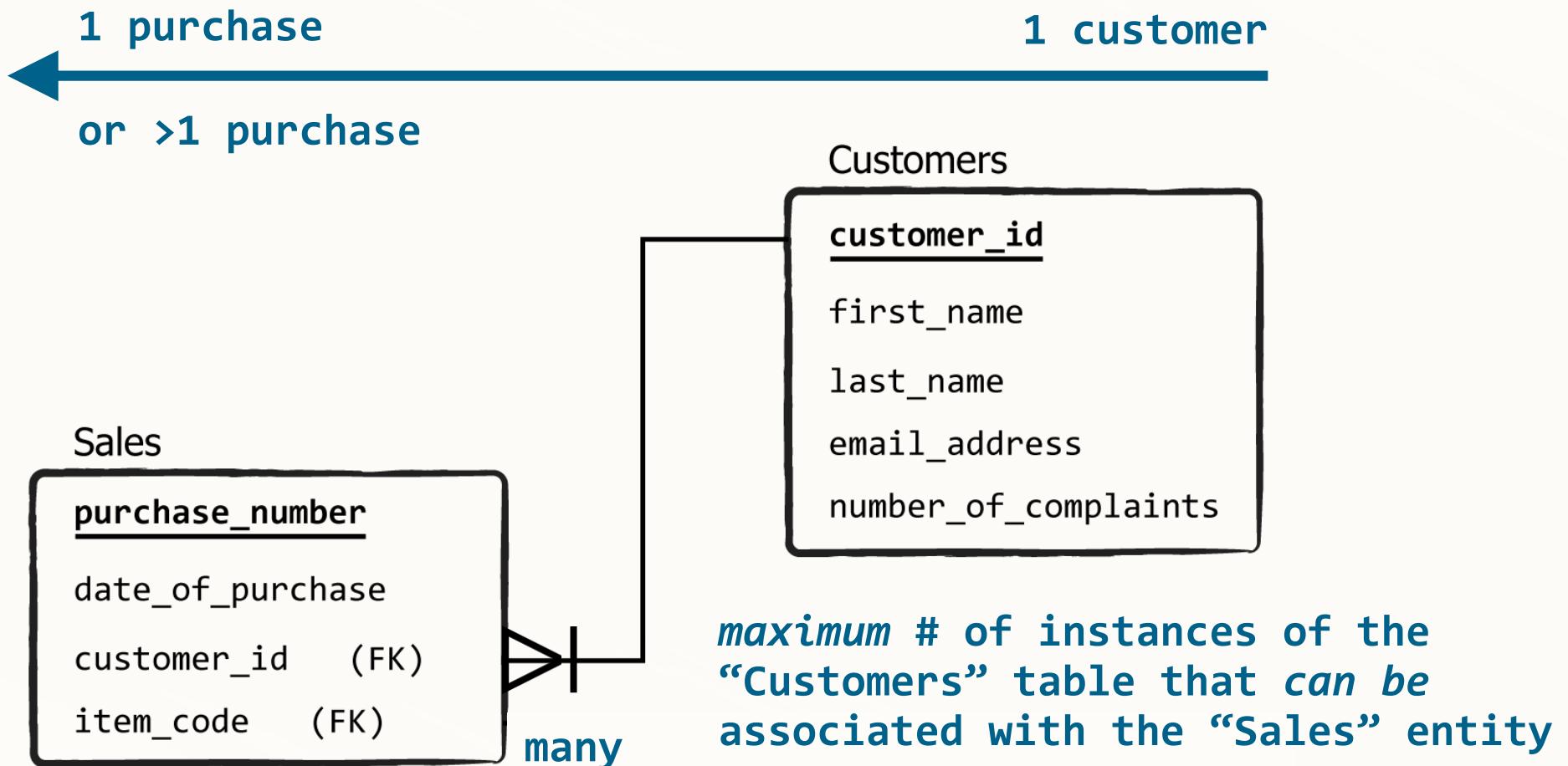
# Relationships



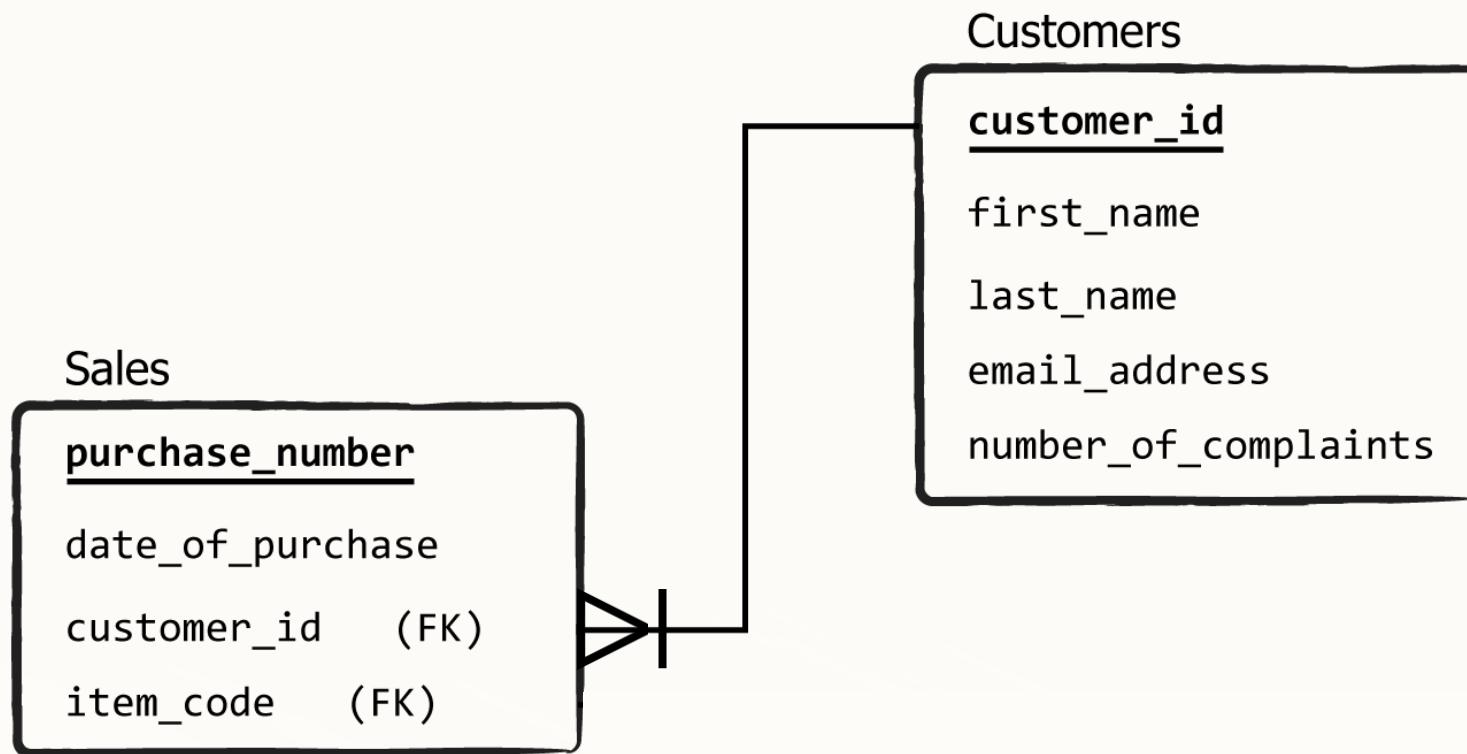
# Relationships



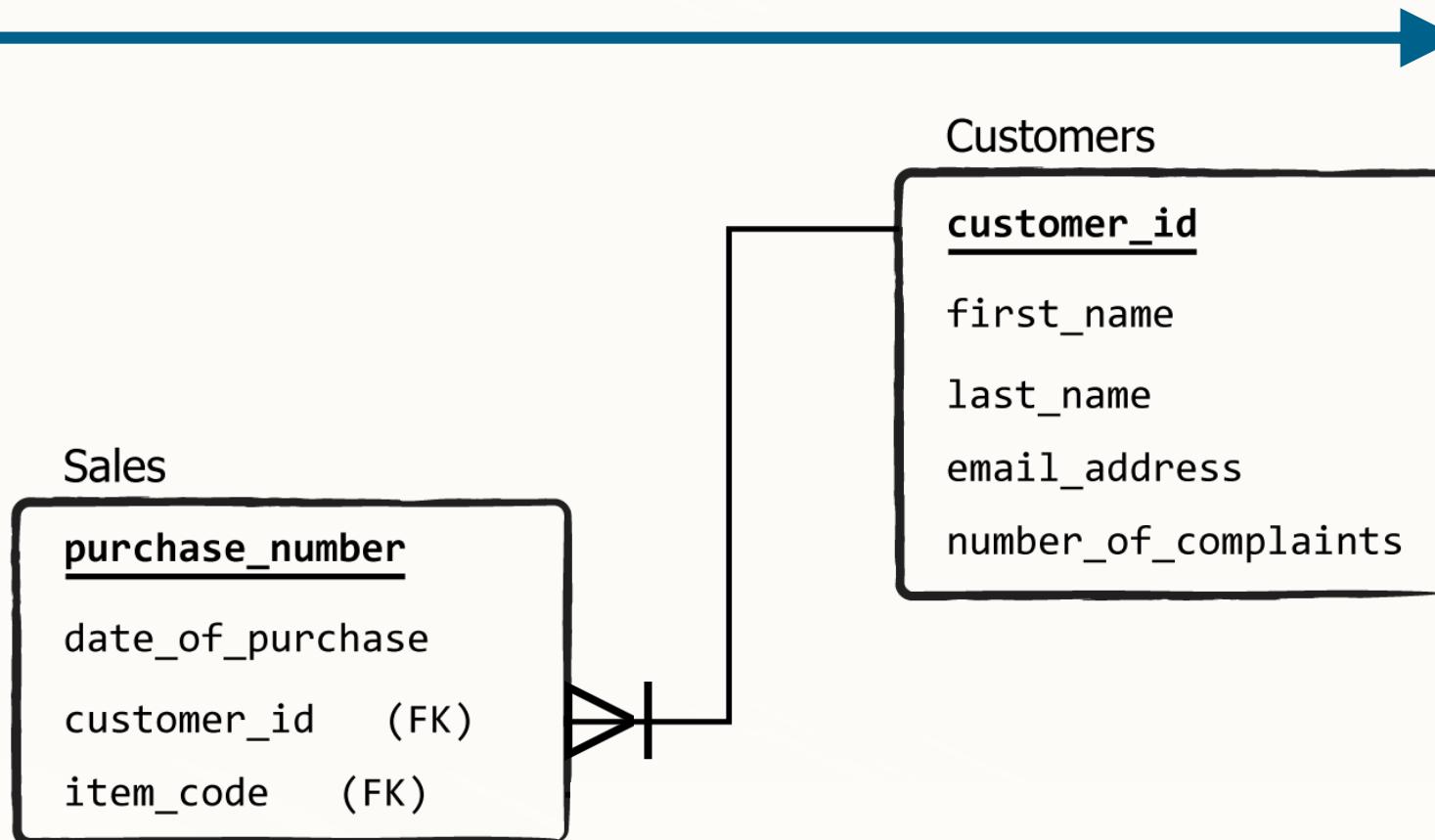
# Relationships



# Relationships

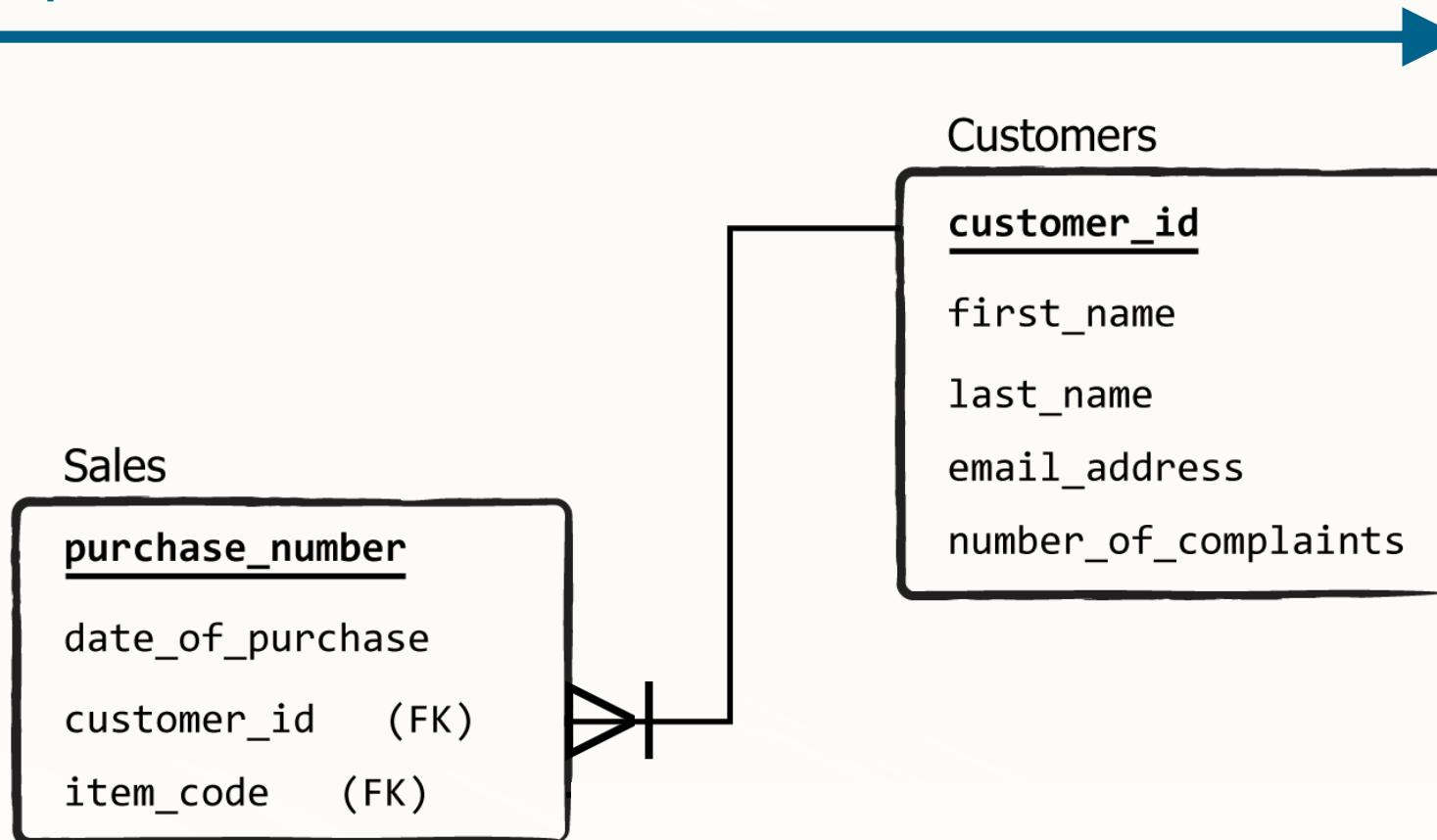


# Relationships



# Relationships

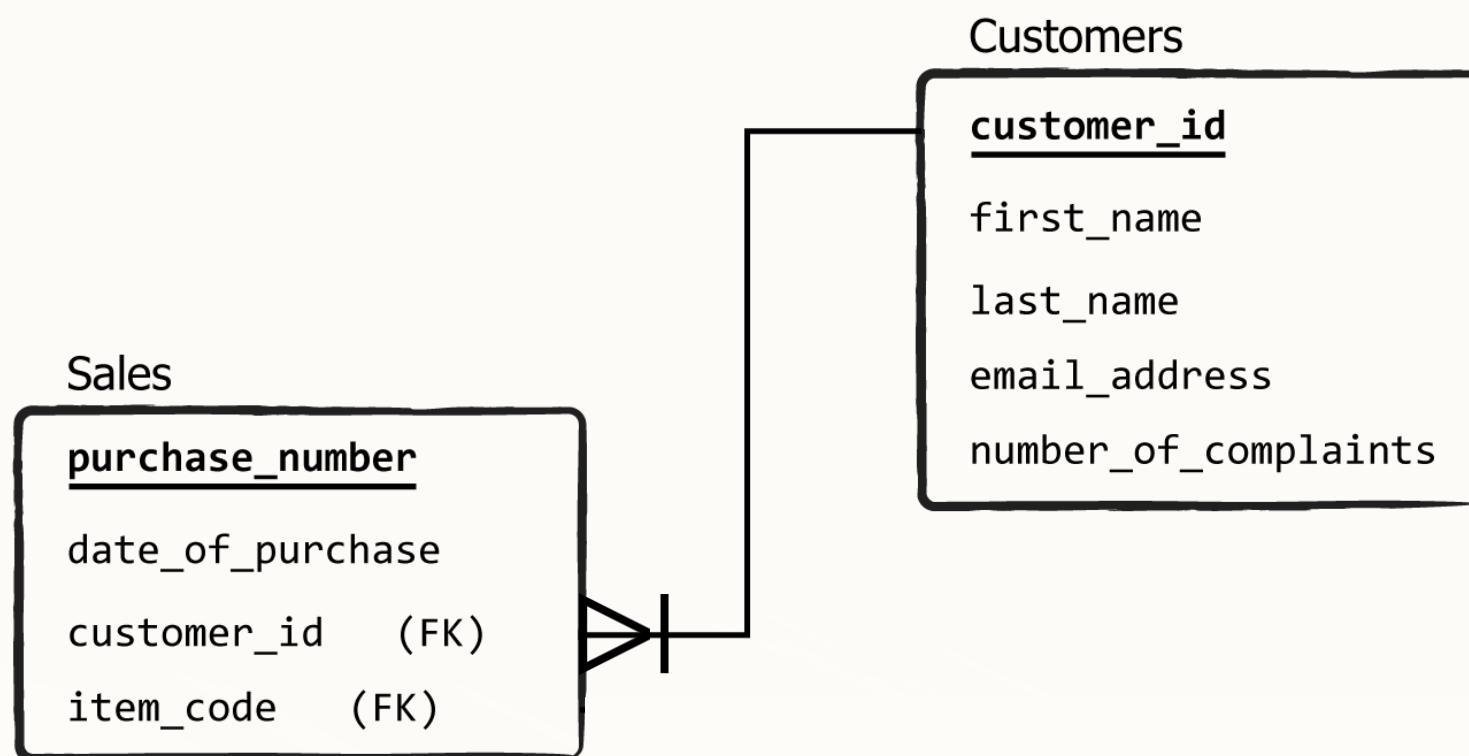
1 purchase



# Relationships

1 purchase

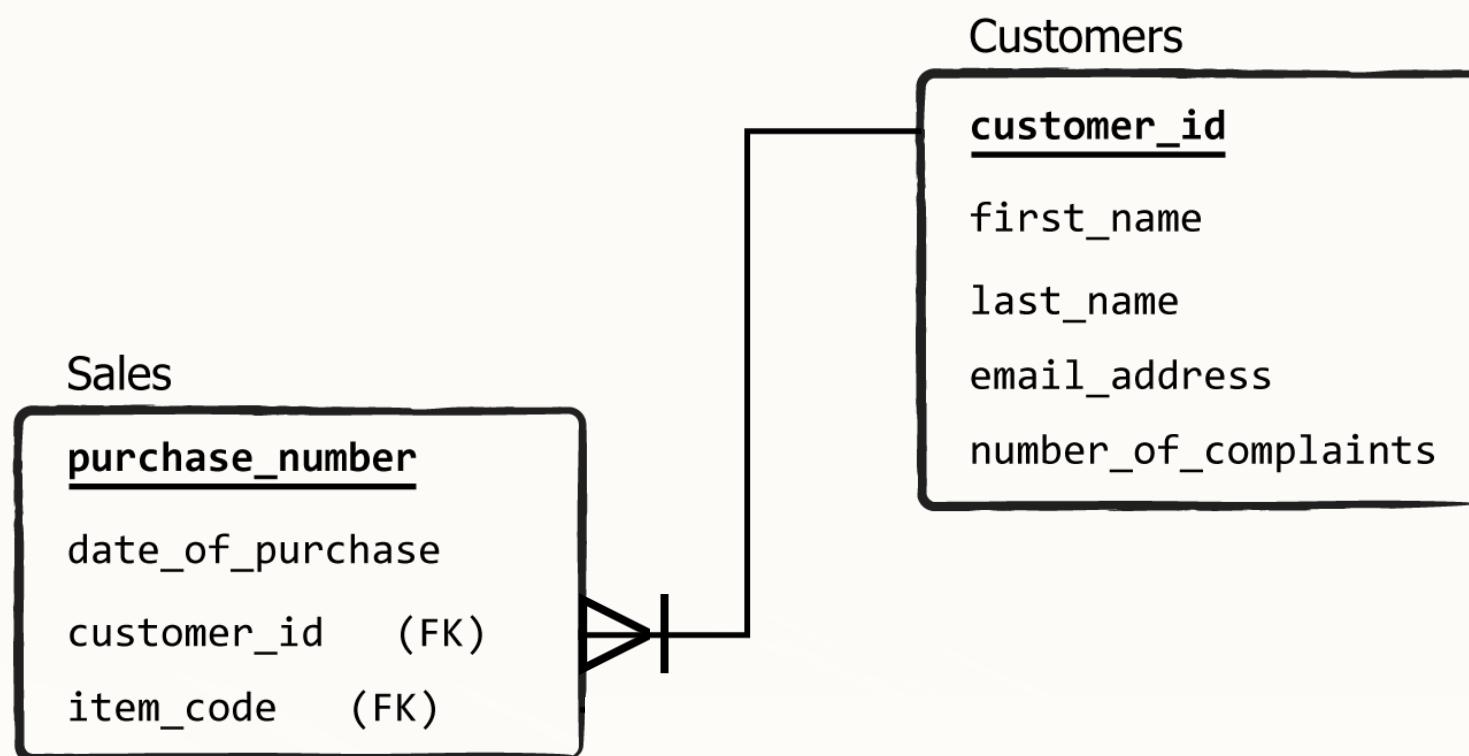
1 customer



# Relationships

1 purchase

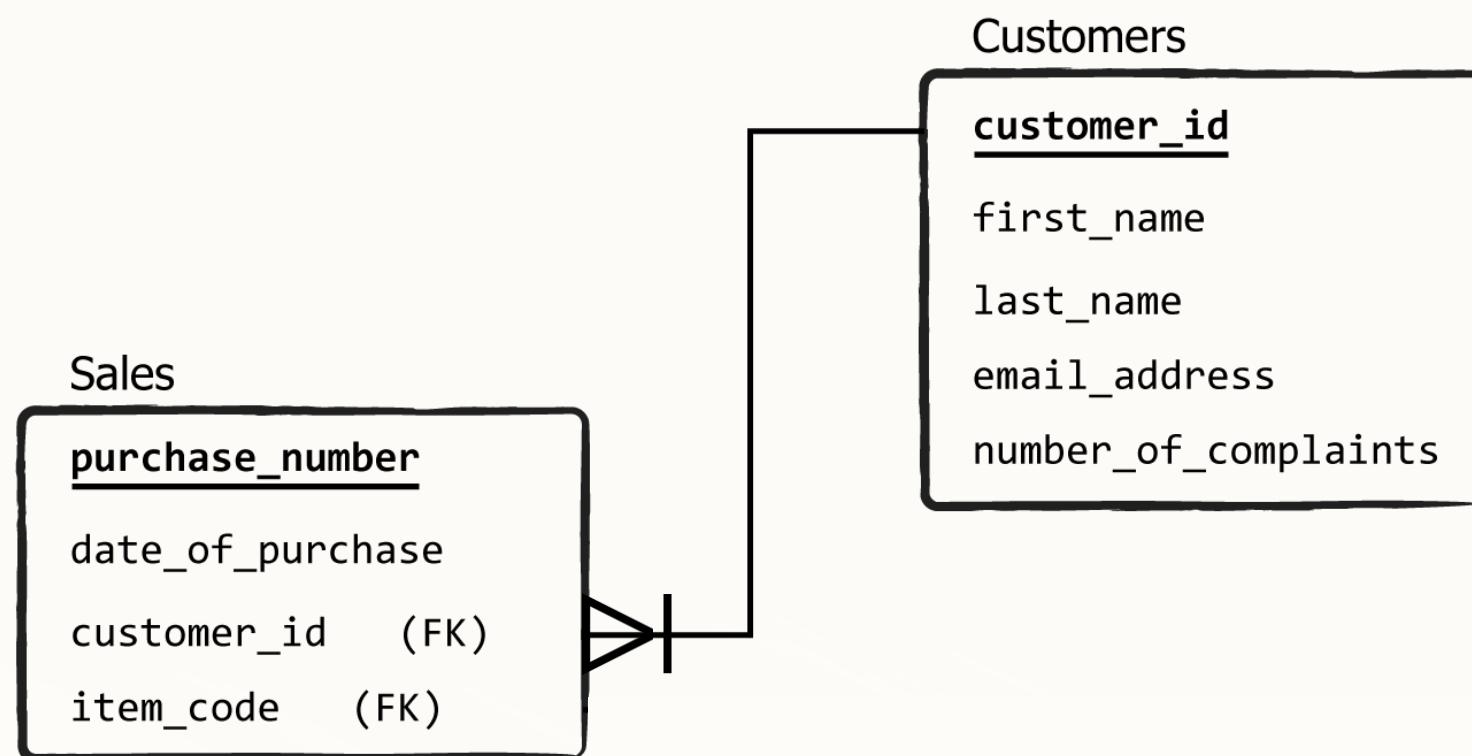
*minimum = 1 customer*



# Relationships

1 purchase

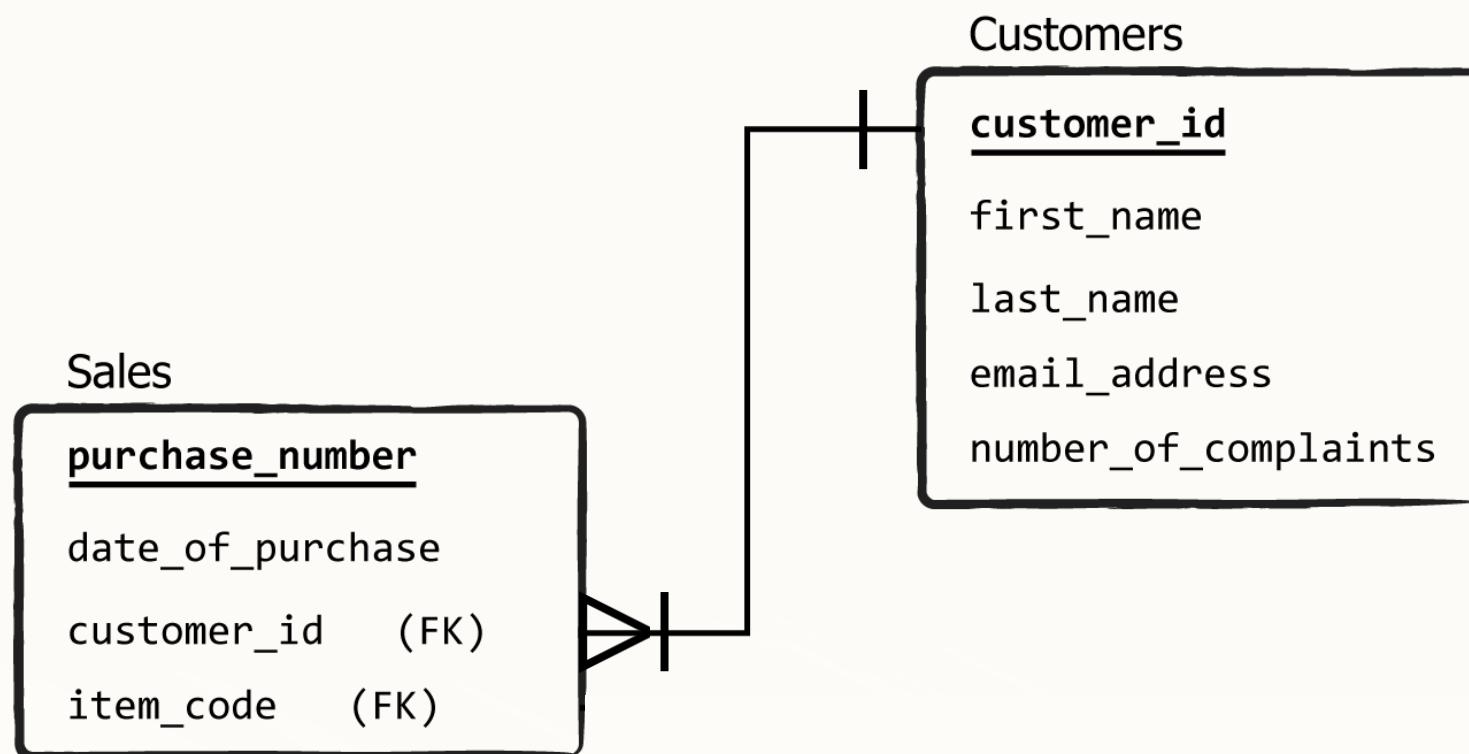
*minimum = 1 customer = maximum*



# Relationships

1 purchase

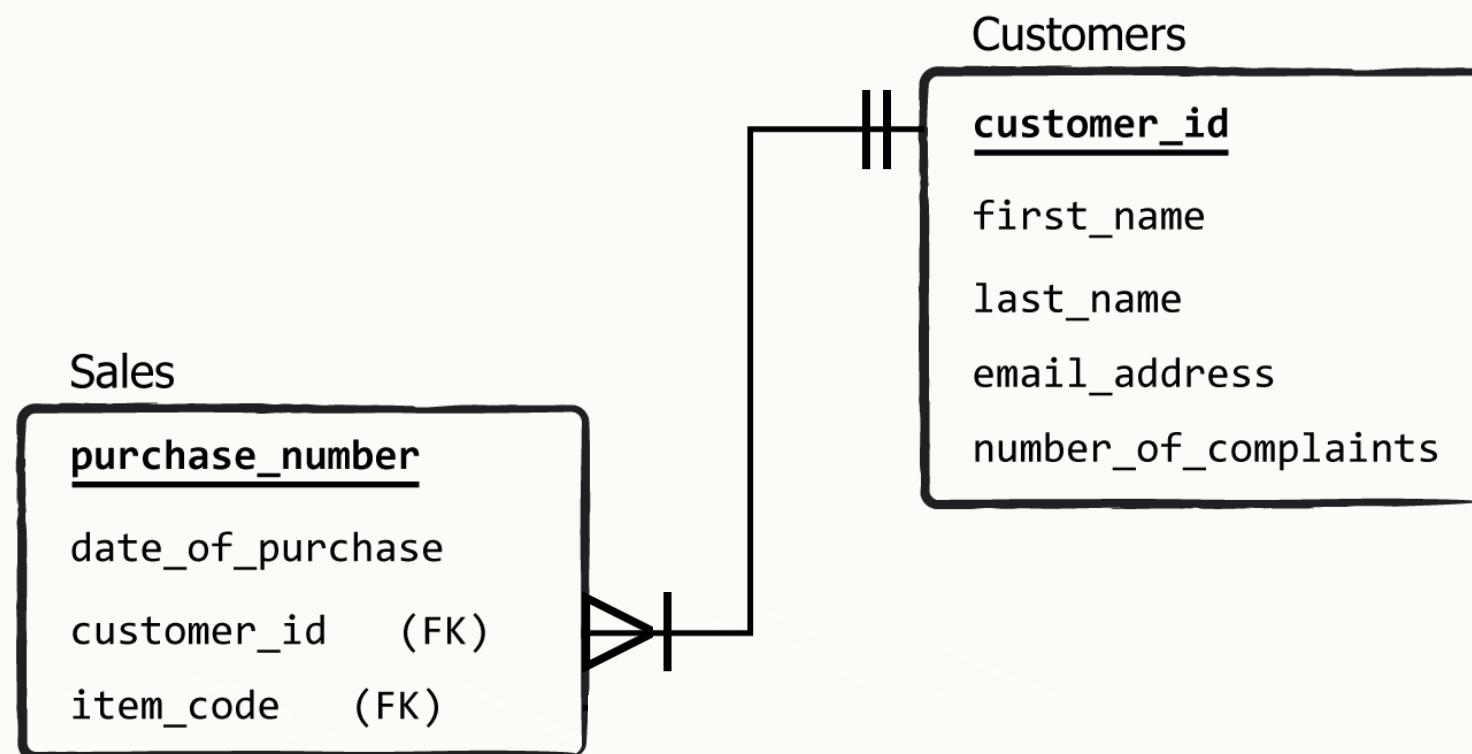
*minimum = 1 customer = maximum*



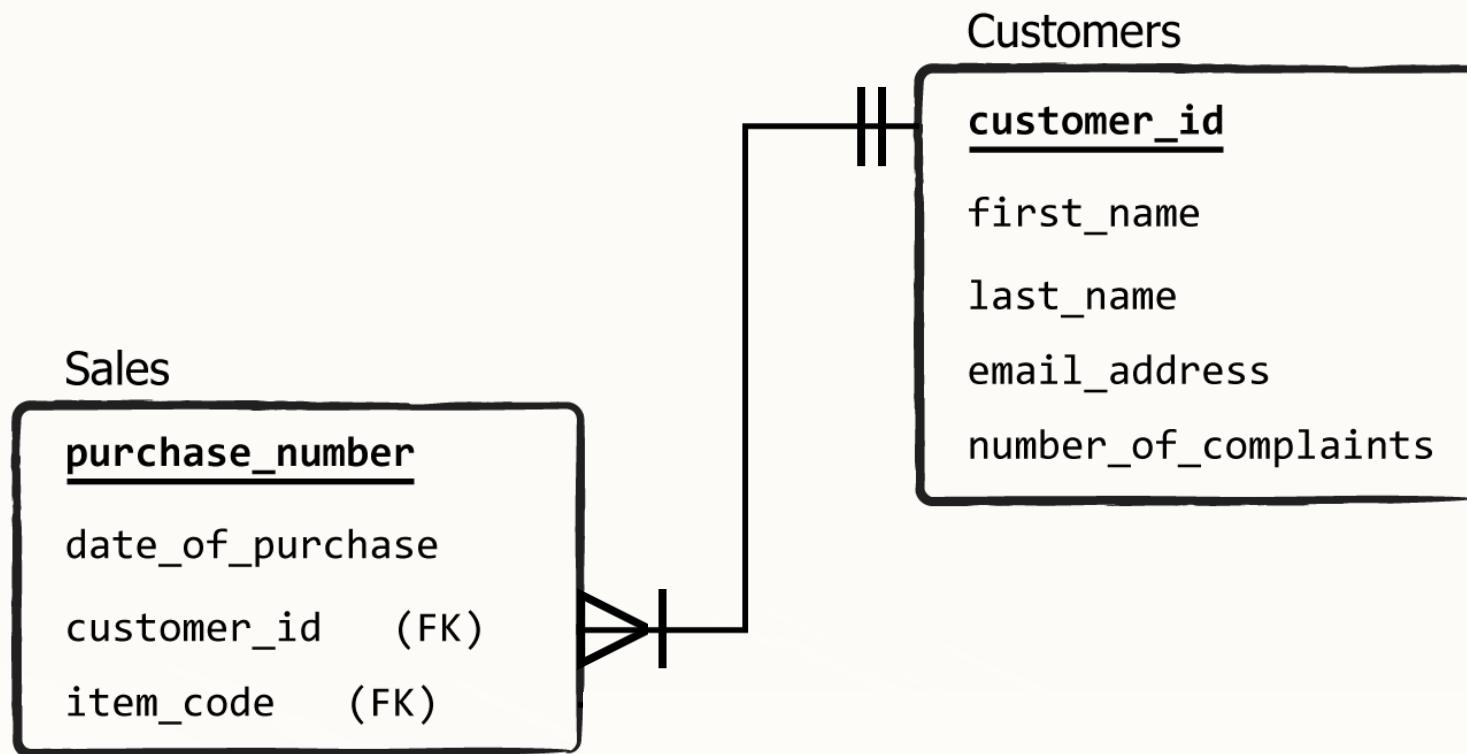
# Relationships

1 purchase

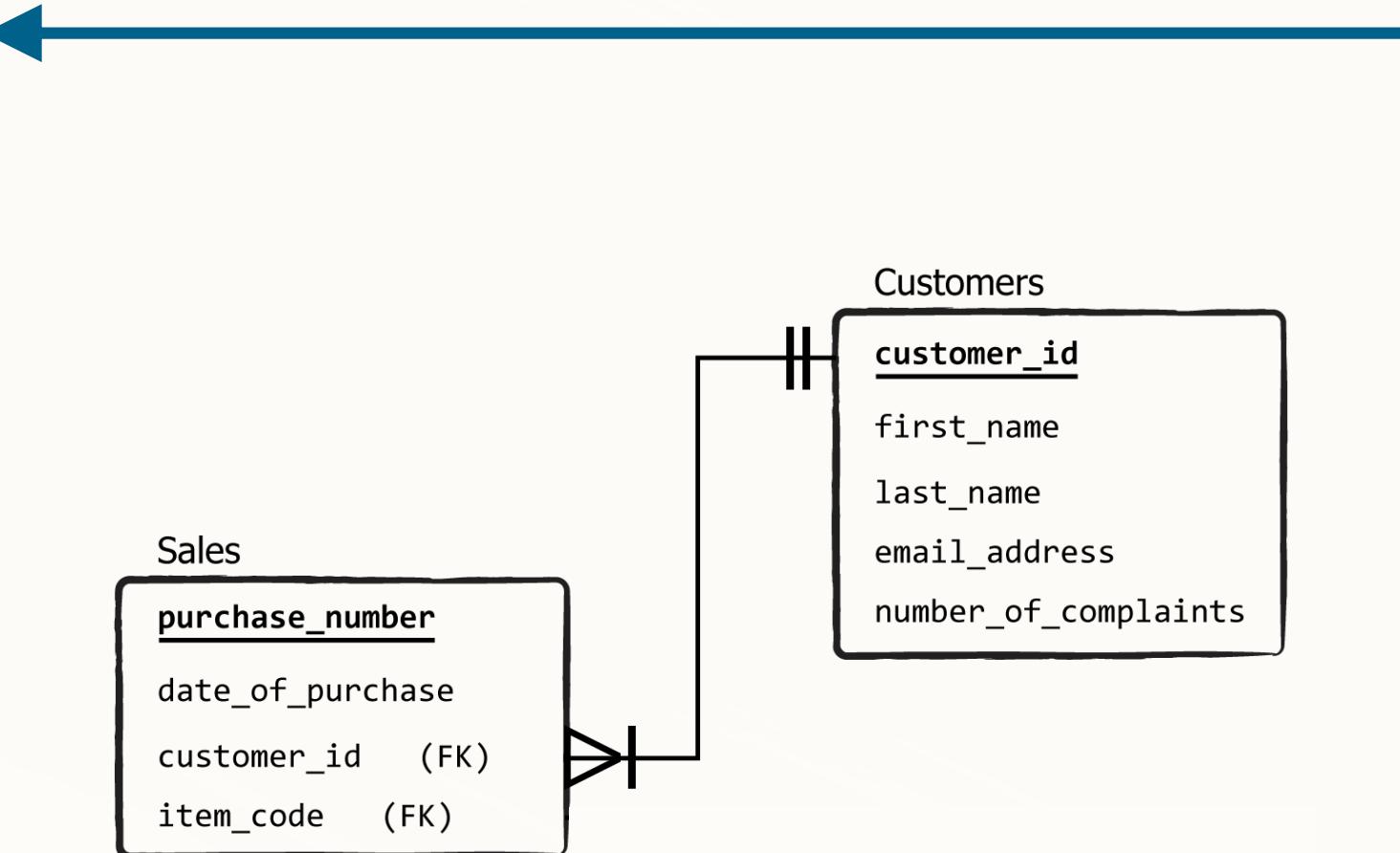
*minimum = 1 customer = maximum*



# Relationships

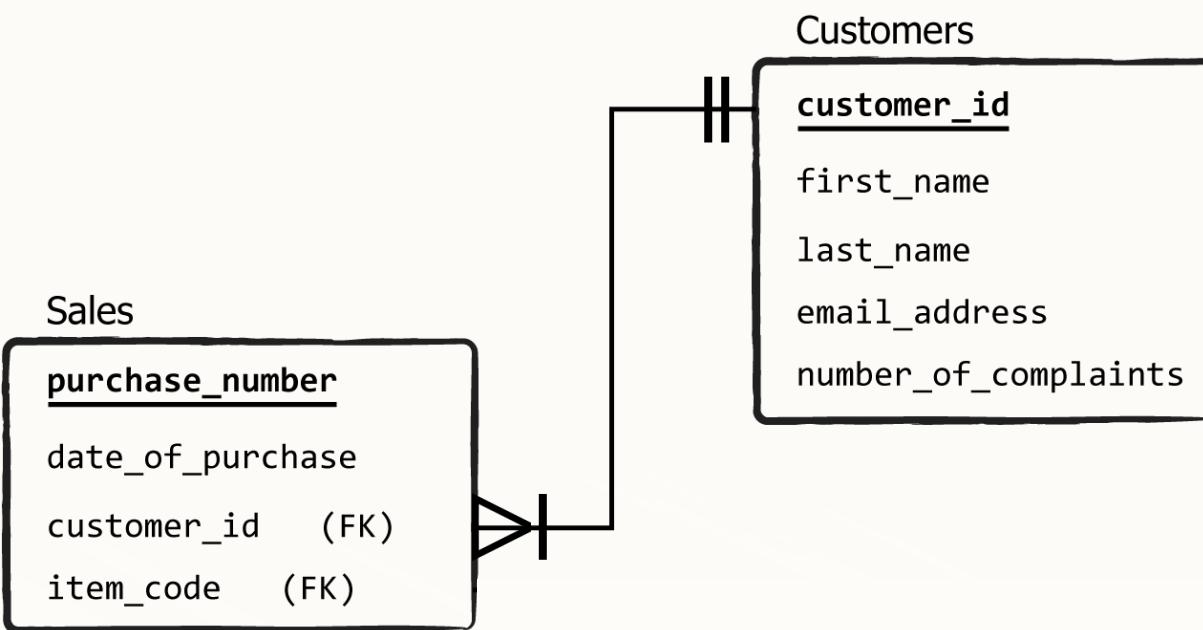


# Relationships

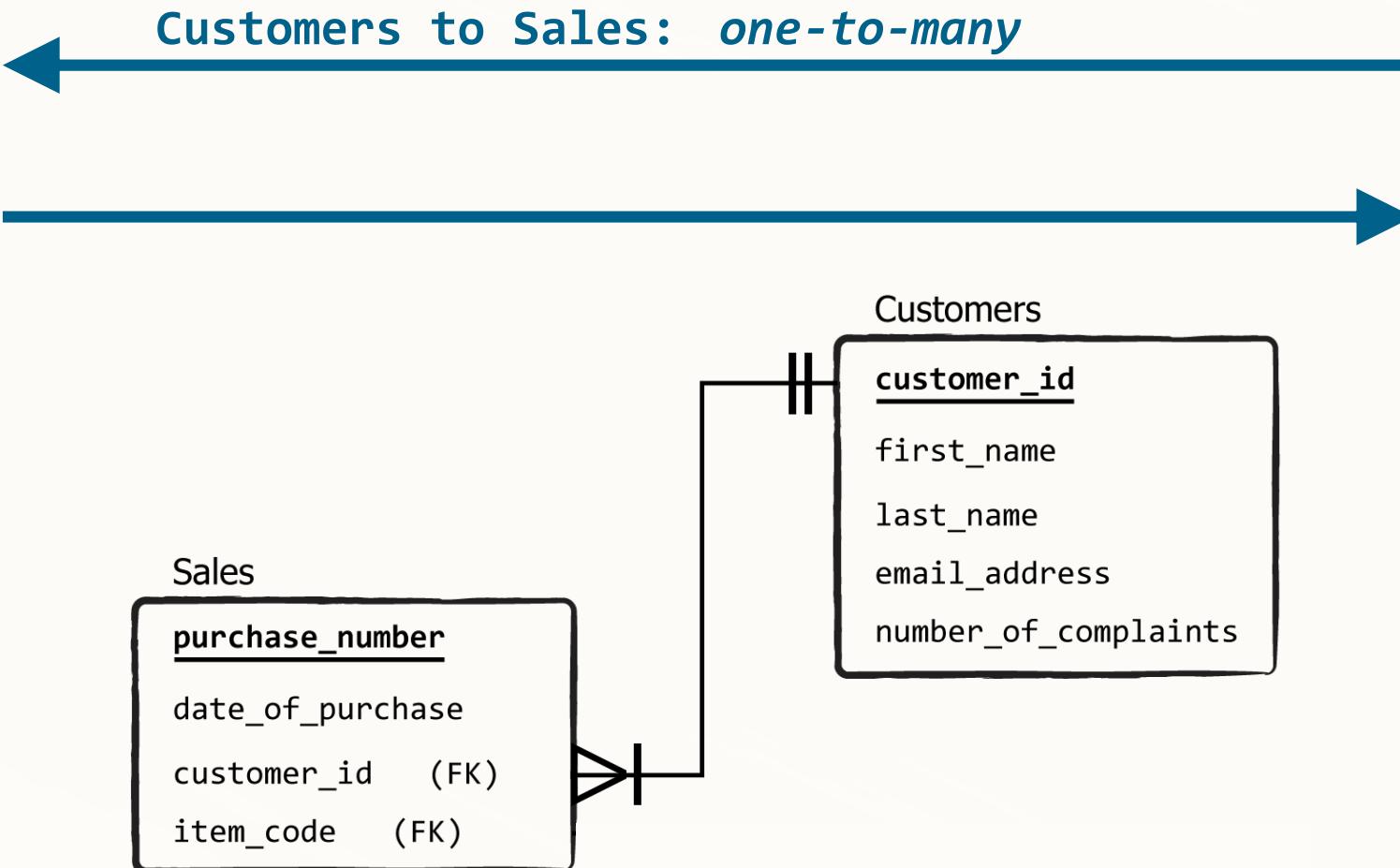


# Relationships

Customers to Sales: *one-to-many*



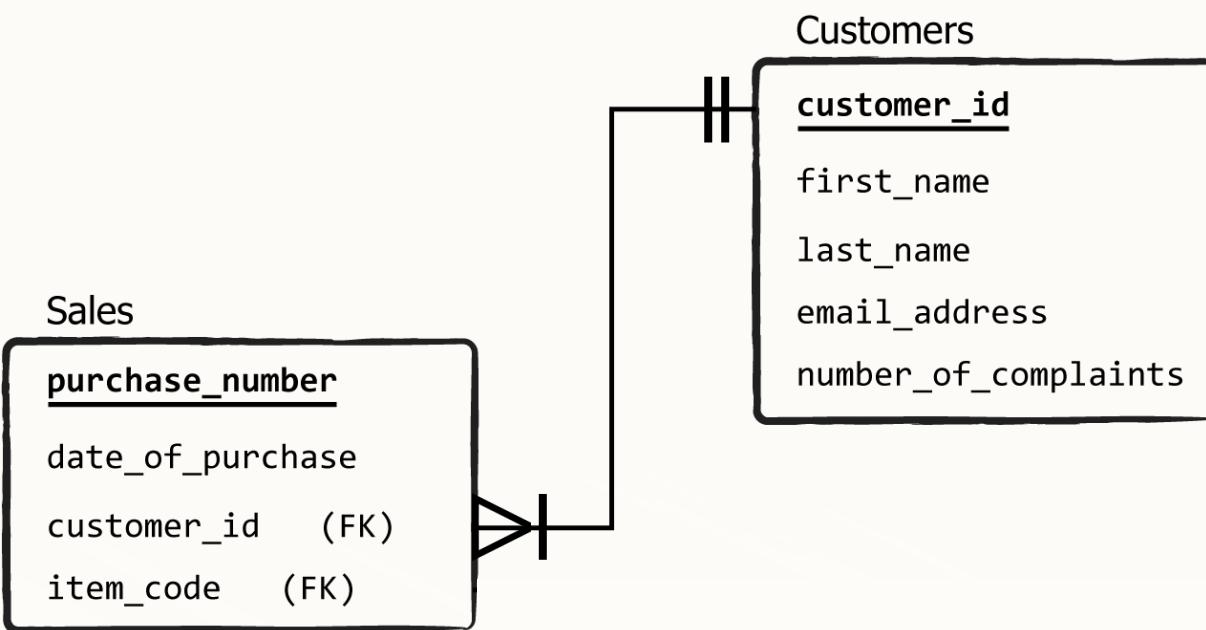
# Relationships



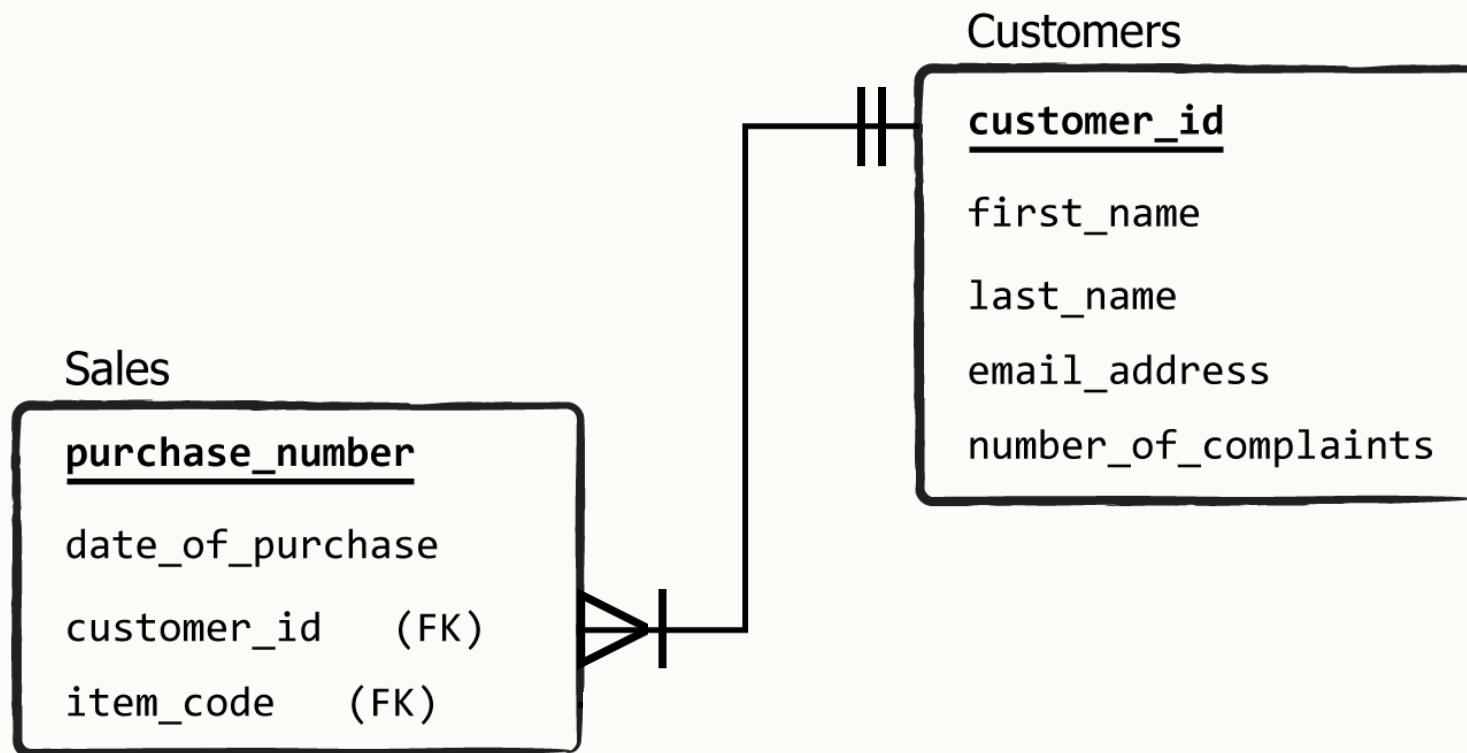
# Relationships

Customers to Sales: *one-to-many*

Sales to Customers: *many-to-one*

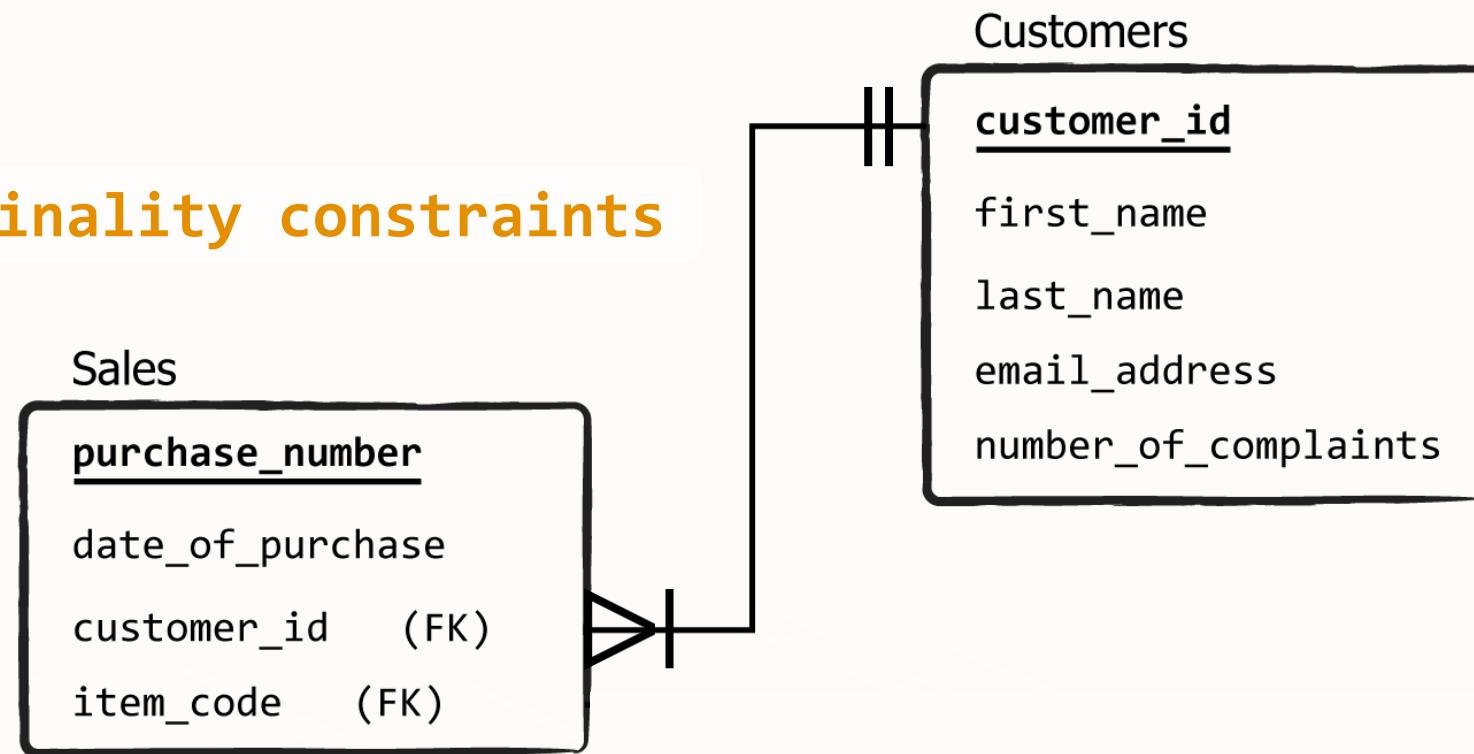


# Relationships



# Relationships

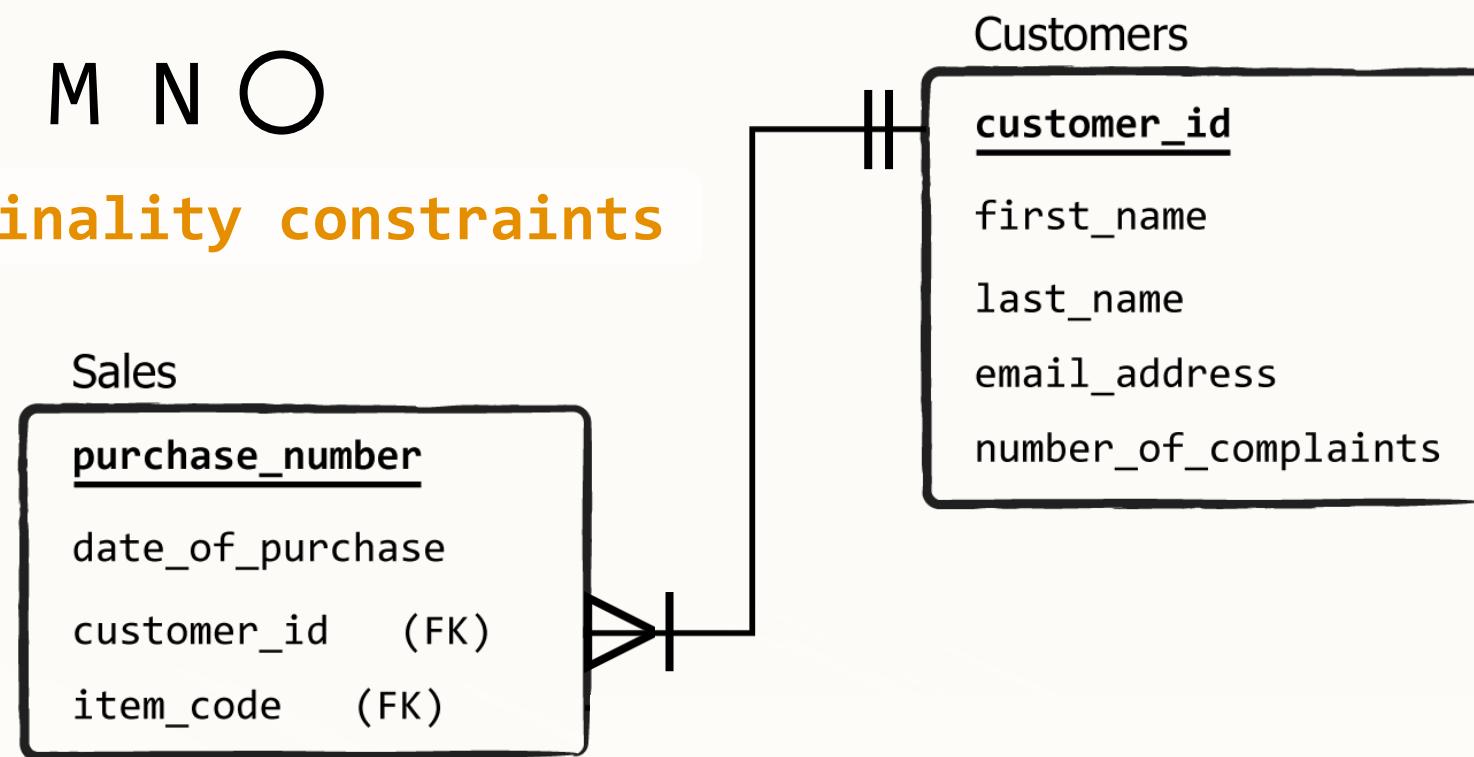
cardinality constraints



# Relationships

| > M N O

cardinality constraints



# Relationships

- **Relationships**

relationships tell you how much of the data from a foreign key field can be seen in the primary key column of the table the data is related to and vice versa

# Relationships

- **Relationships**

relationships tell you how much of the data from a foreign key field can be seen in the primary key column of the table the data is related to and vice versa

- **types of relationships**

# Relationships

- **Relationships**

relationships tell you how much of the data from a foreign key field can be seen in the primary key column of the table the data is related to and vice versa

- **types of relationships**

- one-to-many (many-to-one)

# Relationships

- **Relationships**

relationships tell you how much of the data from a foreign key field can be seen in the primary key column of the table the data is related to and vice versa

- **types of relationships**

- one-to-many (many-to-one)
- one-to-one

# Relationships

- **Relationships**

relationships tell you how much of the data from a foreign key field can be seen in the primary key column of the table the data is related to and vice versa

- **types of relationships**

- one-to-many (many-to-one)
- one-to-one
- many-to-many

# Relationships

- Relational schemas

# Relationships

- **Relational schemas**

- represent the concept database administrators must implement

# Relationships

- **Relational schemas**

- represent the concept database administrators must implement
- depict how a database is organized

# Relationships

- **Relational schemas**

- represent the concept database administrators must implement
- depict how a database is organized
- = blueprints, or a plan for a database

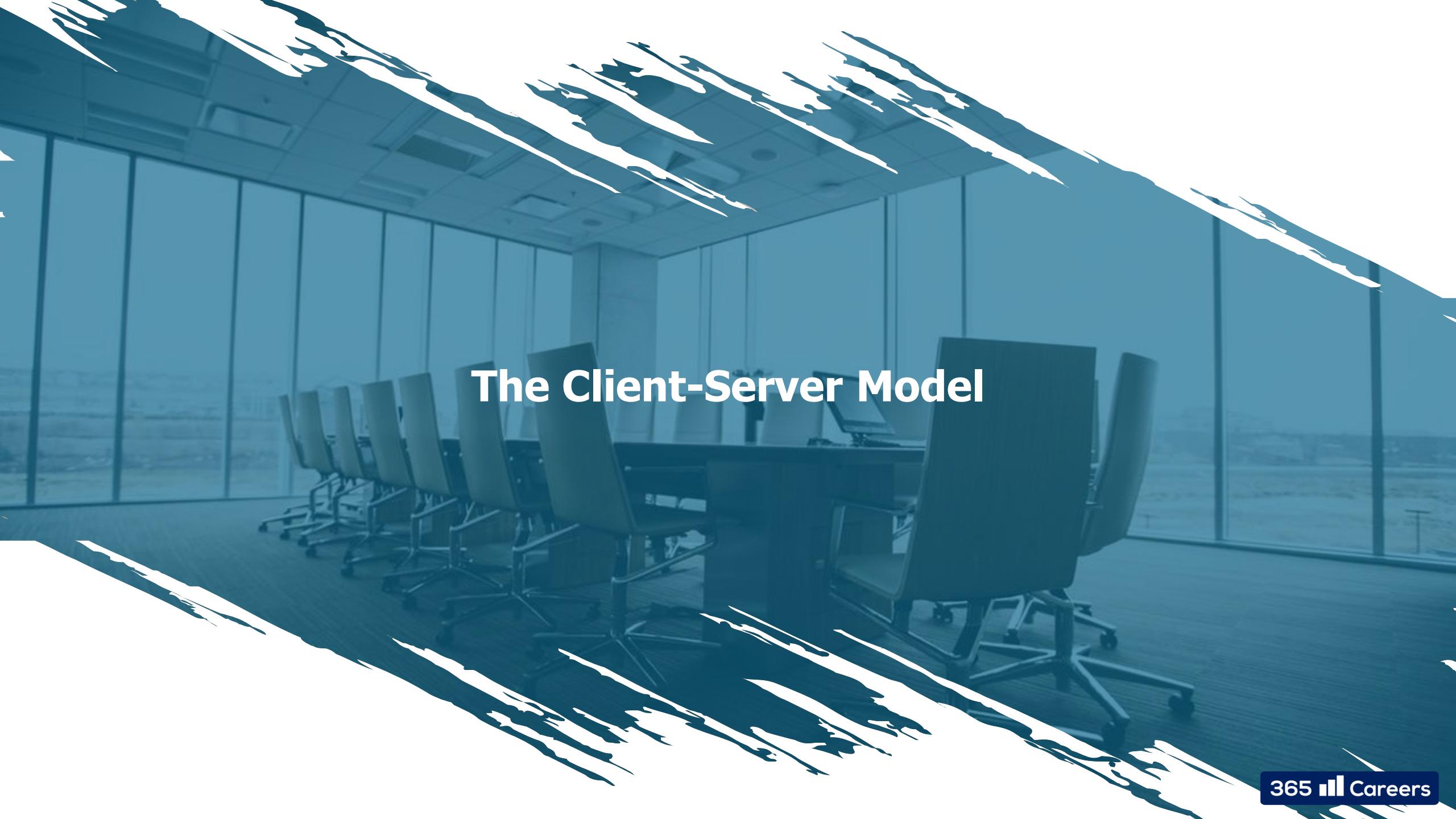
# Relationships

- **Relational schemas**

- represent the concept database administrators must implement
- depict how a database is organized
- = blueprints, or a plan for a database
- *will help you immensely while writing your queries!*

A large, modern conference room with rows of white office chairs facing a long table. The room has a high ceiling with recessed lighting and large windows in the background.

# Installing MySQL and Getting Acquainted with the Interface

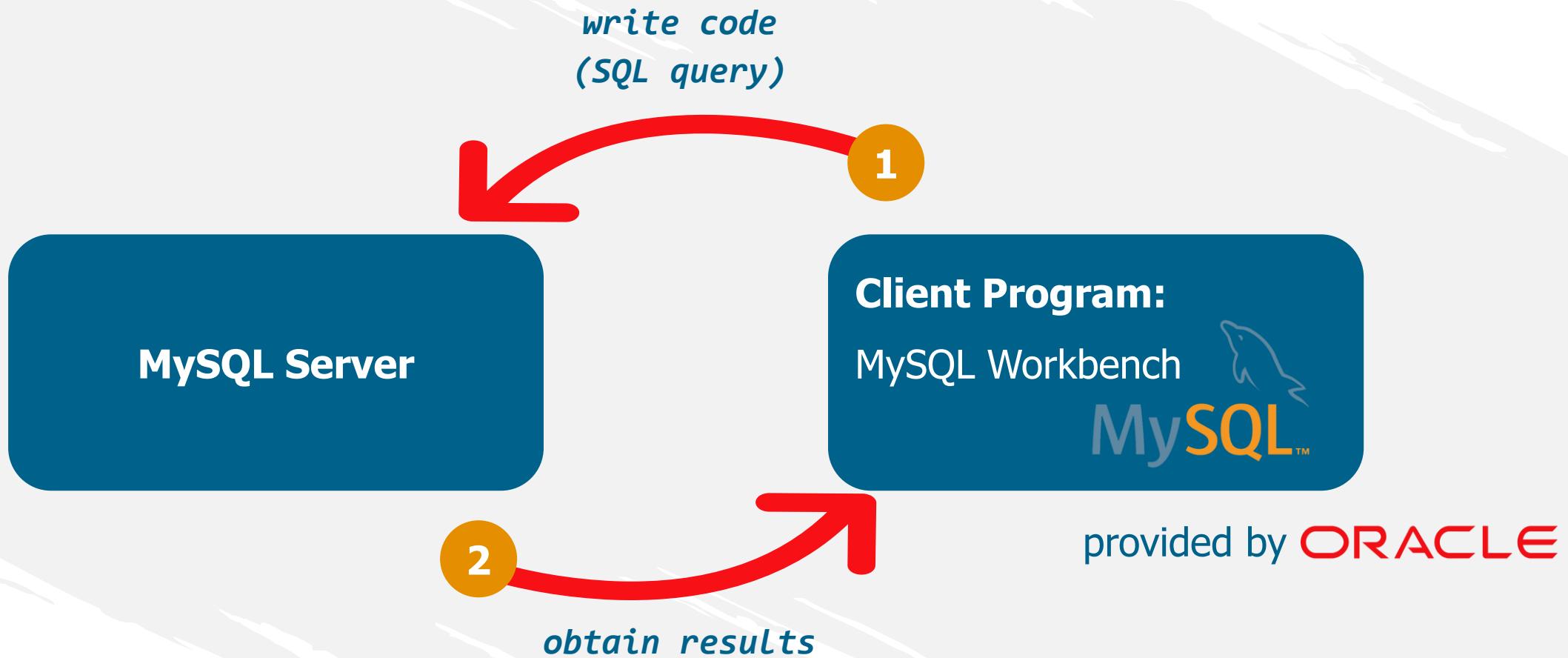
A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows overlooking a city skyline. The chairs are arranged in several rows, facing towards the front of the room where a presentation screen or whiteboard would typically be located.

# The Client-Server Model

# The Client-Server Model

This is a visualization that explains which MySQL features we must install and why.

# The Client-Server Model



# The Client-Server Model

The program we will be working with in this course is called MySQL Workbench. It is the Oracle visual tool for database design, modelling, creation, manipulation, maintenance, and administration. Professionals refer to this type of software as “Integrated Development Environment” or IDE. So, Workbench will be our IDE.

And, if you wonder what *Oracle* is, this is the software company that owns the MySQL version of SQL.

**Client Program:**

MySQL Workbench



provided by **ORACLE**

# The Client-Server Model

You could also wonder why we would need a server. Sticking to the basic theory of operation of computer networks, MySQL Workbench acts as a client program - a client of a MySQL Server.

**MySQL Server**

**Client Program:**

MySQL Workbench



provided by **ORACLE**

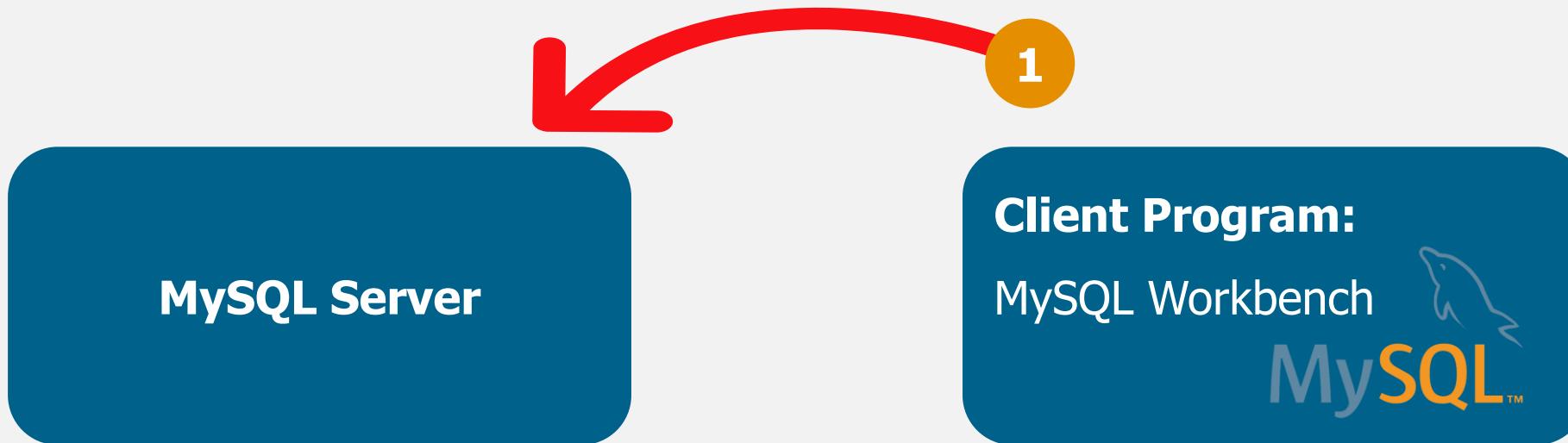
# The Client-Server Model

The server is, practically, nothing more than a machine that provides data and services to the same or other computers.

MySQL Server

The data could be provided locally or online. Regardless whether the server is installed locally on your computer or is being accessed remotely over the internet from another computer, you will need a Server to use MySQL. In our case, we installed the server locally.

# The Client-Server Model



provided by **ORACLE**

Briefly, the server will perform all calculations and operations you execute in Workbench. You will be writing queries through the Workbench interface, in the form of raw code, which MySQL server understands and processes.

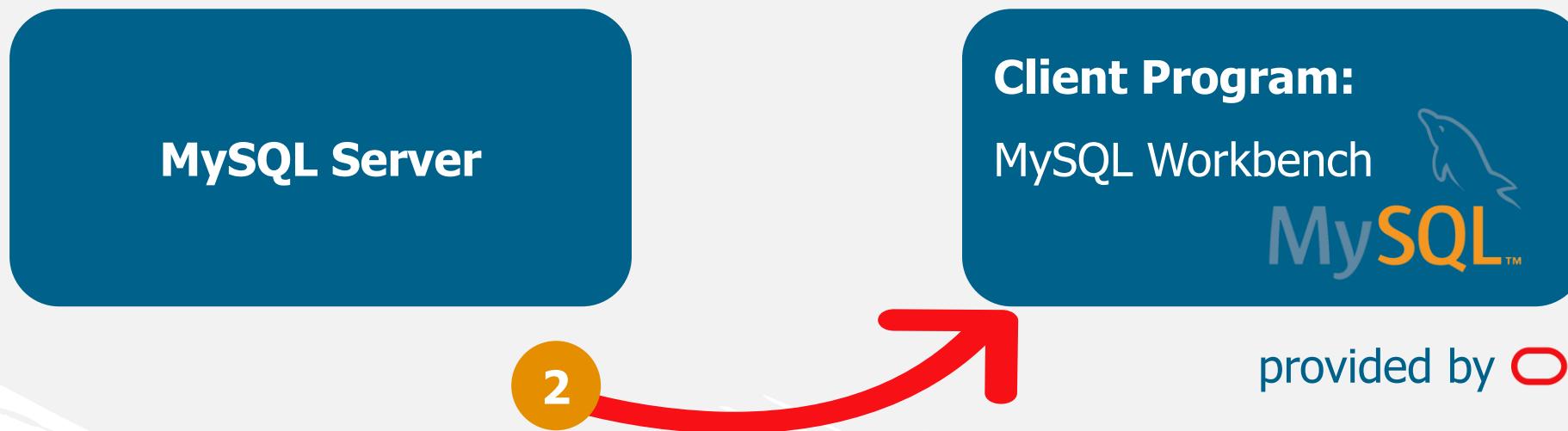
# The Client-Server Model



Briefly, the server will perform all calculations and operations you execute in Workbench. You will be writing queries through the Workbench interface, in the form of raw code, which MySQL server understands and processes.

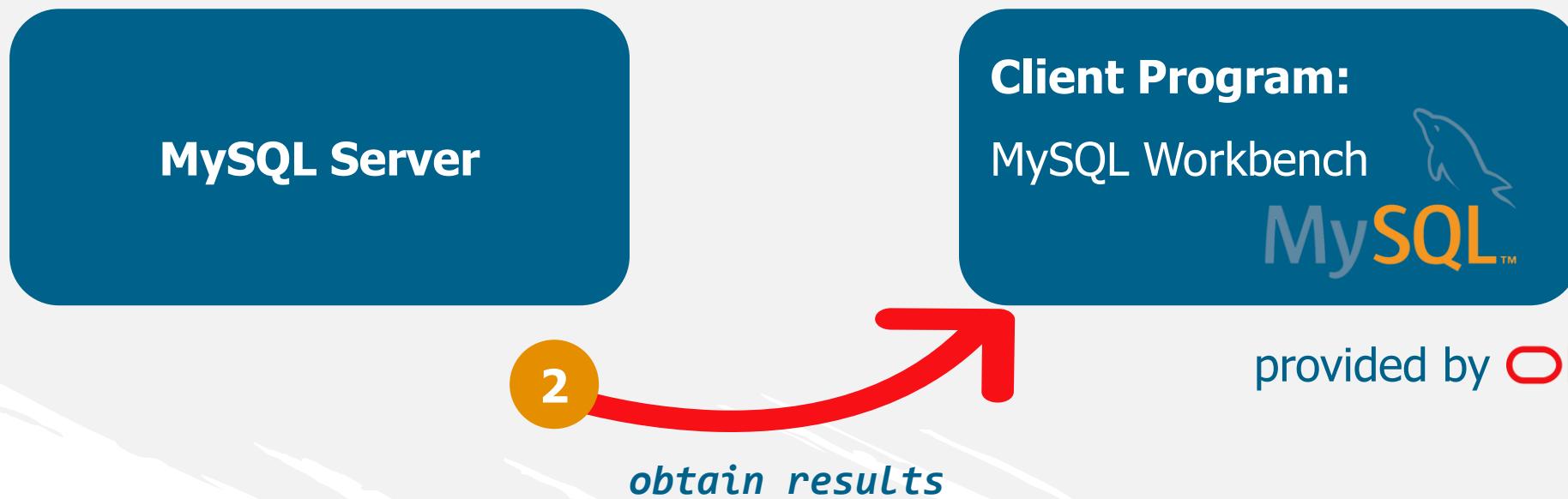
# The Client-Server Model

Finally, when it finalizes its calculations, it will bring the respective results back to you in the form of an output displayed on your screen.

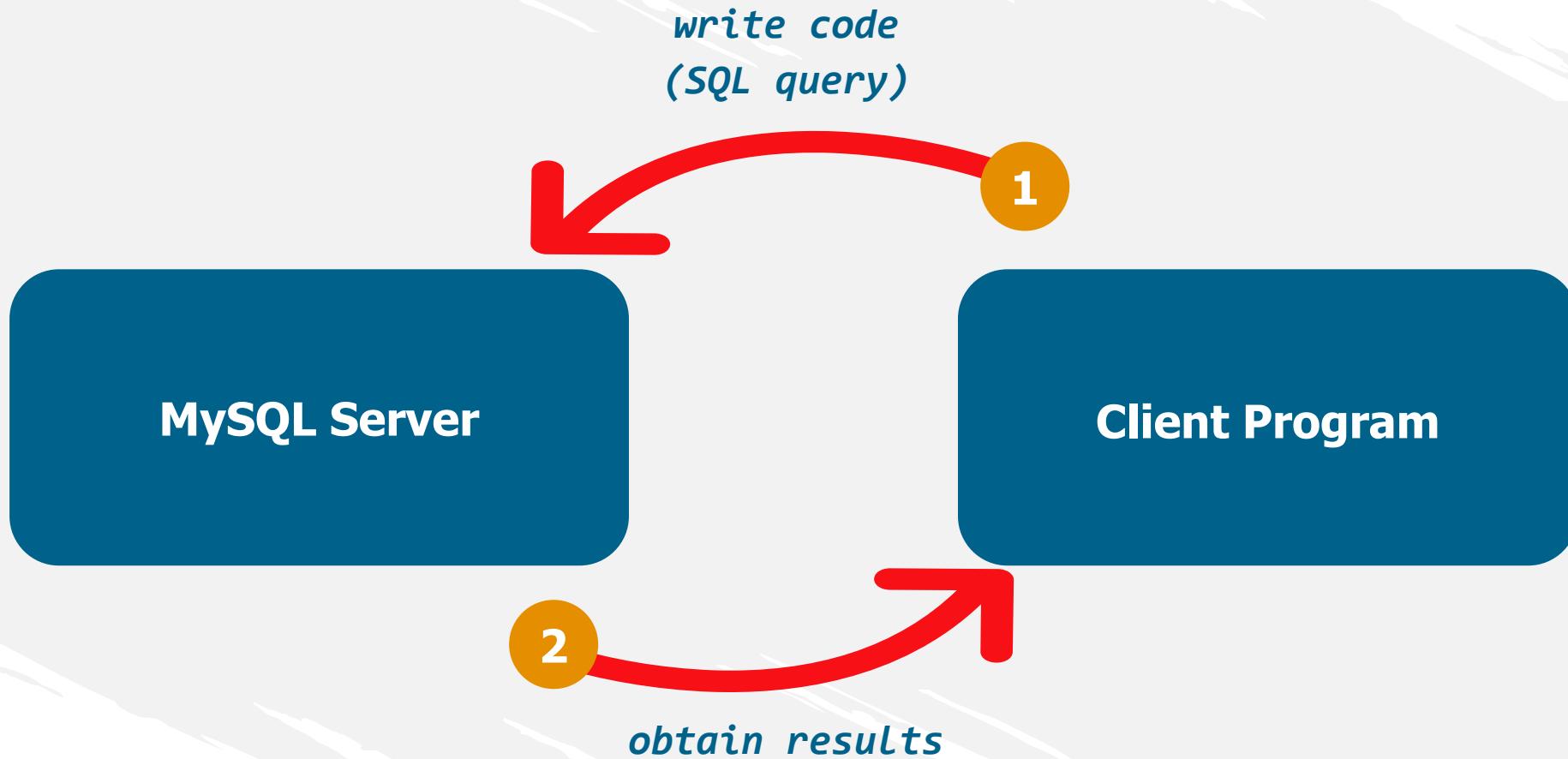


# The Client-Server Model

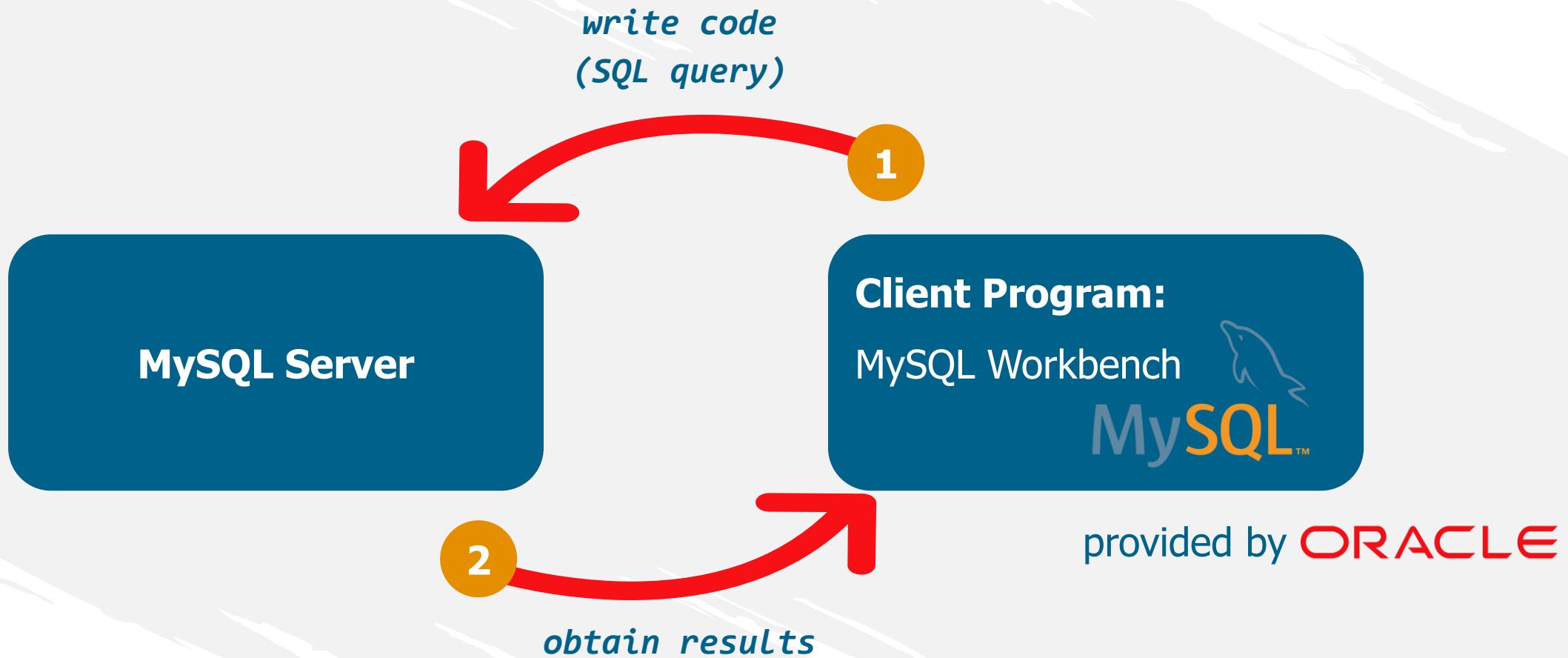
Finally, when it finalizes its calculations, it will bring the respective results back to you in the form of an output displayed on your screen.



# The Client-Server Model



# The Client-Server Model



A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

# First Steps in SQL

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of black office chairs facing towards the front. The front wall features several large, dark rectangular panels, likely projection screens or television monitors. The room has a high ceiling with a grid of recessed lighting fixtures. The floor is a polished wood. The overall atmosphere is professional and spacious.

# Creating a Database – Part I

# Creating a Database – Part I

So far:

# Creating a Database – Part I

## So far:

- Theory of Relational Databases

# Creating a Database – Part I

## So far:

- Theory of Relational Databases
- SQL Theory

# Creating a Database – Part I

## So far:

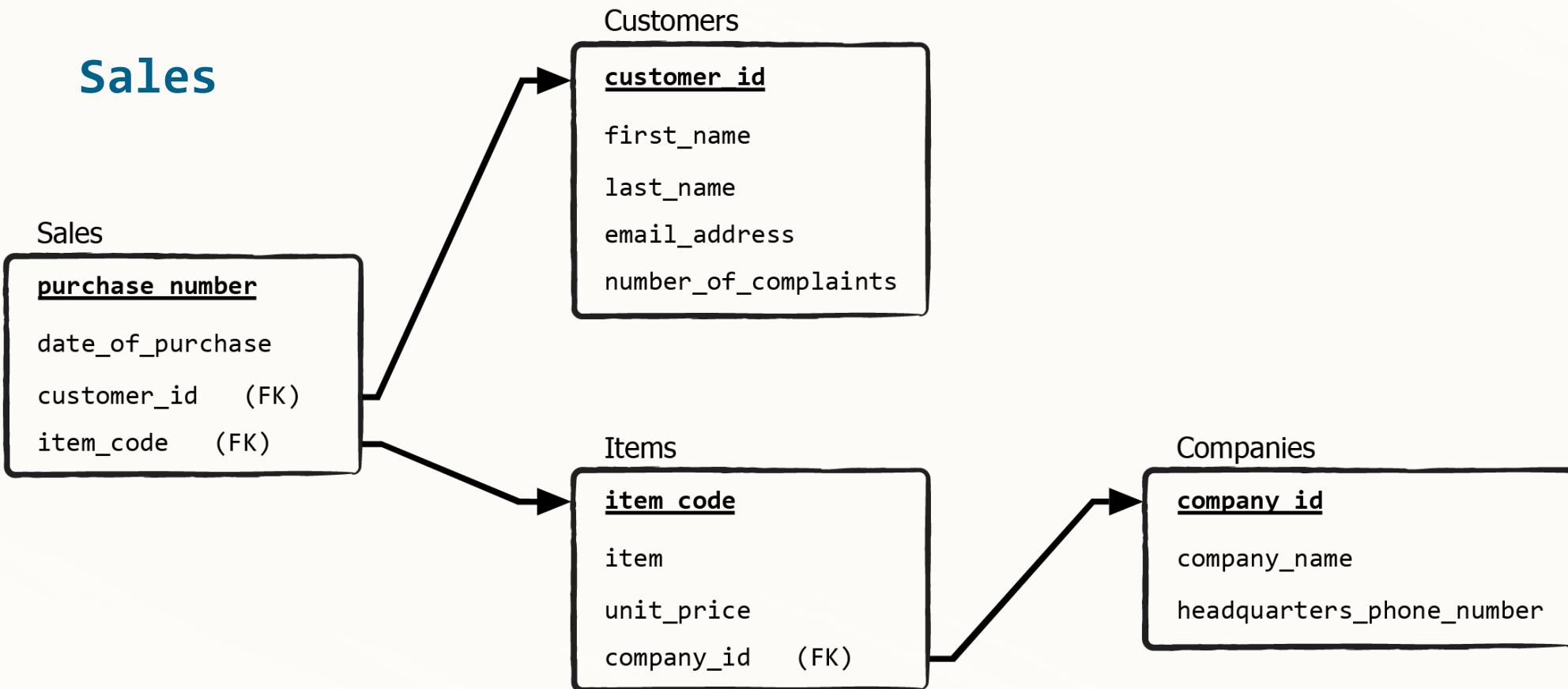
- Theory of Relational Databases
- SQL Theory
- Download and Installation of MySQL Workbench (provided by **ORACLE®**)



# Creating a Database – Part I

Sales

# Creating a Database – Part I



# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- **CREATE DATABASE**

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- **CREATE DATABASE**

*creates a database as an abstract unit*

# Creating a Database – Part I

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- [IF NOT EXISTS]

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- [IF NOT EXISTS]

verifies if a database with the same name exists already

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- [IF NOT EXISTS]

verifies if a database with the same name exists already

- the brackets around mean the statement is *optional* (you could either type or omit the statement)

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- database\_name

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- database\_name

give a name that is short but at the same time as related to the content of the data as possible

# Creating a Database – Part I

Sales

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- database\_name

give a name that is short but at the same time as related to the content of the data as possible

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- database\_name

give a name that is short but at the same time as related to the content of the data as possible

- the SQL code is not case sensitive

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- database\_name

give a name that is short but at the same time as related to the content of the data as possible

- the SQL code is not case sensitive
- in this element the quotes are optional

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- ; (the semicolon character)

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- ; (the semicolon character)

it functions as a *statement terminator*

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- ; (the semicolon character)
  - it functions as a *statement terminator*
  - when your code contains more than a single statement, ; is indispensable

# Creating a Database – Part I



SQL

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

- ; (the semicolon character)

it functions as a *statement terminator*

- when your code contains more than a single statement, ; is indispensable
- will help you avoid errors sometimes

# Creating a Database – Part I



```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

SQL

- ; (the semicolon character)

it functions as a *statement terminator*

- when your code contains more than a single statement, ; is indispensable
- will help you avoid errors sometimes
- will improve the readability of your code

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of dark-colored office chairs arranged facing a front wall. The front wall features a large, multi-panel screen or a series of monitors. The room has a high ceiling with a grid of recessed lighting fixtures. Large windows along the side walls provide a view of an outdoor area with some greenery and possibly a parking lot.

# Introduction to Data Types

# Introduction to Data Types

- We must always specify the type of data that will be inserted in each column of the table

Different data types represent different *types of information* that can be contained in a specific column

# Introduction to Data Types

	Surname of a person:
string	‘James’

# Introduction to Data Types

string

Surname of a person:

‘James’

length

a measure used to indicate how many symbols a certain string has

# Introduction to Data Types

<b>string</b>	Surname of a person: ‘James’	<b>length</b> 5 symbols	
	‘Jackson’	7 symbols	

**length**

a measure used to indicate how many symbols a certain string has

# Introduction to Data Types

- Digits, symbols, or blank spaces can also be used in the string format

they will only convey text information

e.g. addresses: ‘Hugh Street 45’

# Introduction to Data Types

## size

indicates the memory space used by a data type

- measured in bytes
- 1 byte ~ 1 symbol

# Introduction to Data Types

<b>string</b>	Surname of a person: ‘James’	<b>length</b> 5 symbols	<b>size</b> 5 bytes
---------------	---------------------------------	----------------------------	------------------------

# Introduction to Data Types

## storage

the physical space in the computer drive's memory, where the data is being saved or stored

A large conference room with rows of chairs facing a central table. The room has a modern design with a glass partition and large windows overlooking a landscape. The floor is made of light-colored wood.

# String Data Types

# String Data Types

- **String** - the text format in SQL

‘James’ - a variable of the **string** data type

= a variable of the **alphanumeric** data type

# String Data Types

<u>string data type</u>		<u>Storage</u>	<u>Example</u>
<i>character</i>	CHAR	fixed	CHAR(5)

# String Data Types

<u>string data type</u>		<u>Storage</u>	<u>Example</u>	<u>Length</u> (symbols)	<u>size</u> (bytes)
character	CHAR	fixed	CHAR(5)  ‘James’  ‘Bob’	5  3	5  5

CHAR(5)

5 represents the maximum number of symbols you are allowed to use in writing a value in this format

# String Data Types

<u>string data type</u>		<u>Storage</u>	<u>Example</u>	<u>Length</u> (symbols)	<u>size</u> (bytes)
<i>character</i>	<b>CHAR</b>	fixed	<b>CHAR(5)</b>  ‘James’ ‘Bob’	5 3	5 5
<i>variable character</i>	<b>VARCHAR</b>	variable	<b>VARCHAR(5)</b>  ‘James’ ‘Bob’	5 3	5 3

# String Data Types

<u>string data type</u>		<u>Maximum size</u> (bytes)
<i>character</i>	<b>CHAR</b>	255
<i>variable character</i>	<b>VARCHAR</b>	65,535

# String Data Types

<u>string data type</u>		<u>Maximum size</u> (bytes)	
<i>character</i>	<b>CHAR</b>	255	50% faster
<i>variable character</i>	<b>VARCHAR</b>	65,535	a lot more responsive to the data value inserted

# String Data Types

Companies		
company_id	headquarters_phone_number	company
1	+1 (202) 555-0196	COA
2	+1 (202) 555-0152	COB
3	+1 (229) 853-9913	COC
4	+1 (618) 369-7392	COD

company CHAR(3)

Password:

the symbols cannot be more than  
10 characters

password VARCHAR(10)

# String Data Types

<u>string data type</u>		<u>Example</u>
<i>character</i>	CHAR	CHAR(5)
<i>variable character</i>	VARCHAR	VARCHAR(5)
<i>ENUM ("enumerate")</i>	ENUM	ENUM('M', 'F')  ERROR

MySQL will show an error if you attempt to insert any value different from "M" or "F".

A large, modern conference room with a long rectangular table in the center. On the table, there are several open laptops. Rows of black office chairs are arranged facing the table. The room has large windows on one side, providing a view of an outdoor area. The ceiling is white with a grid of recessed lights. The overall atmosphere is professional and spacious.

# Integers

# Integers

text ≠ numbers

numeric data types

integer

fixed-point

floating-point

integers

whole numbers with no decimal point

e.g. 5; 15; -200; 1,000

INTEGER    INT

# Integers

<u>numeric data type</u>	<u>size (bytes)</u>	<u>minimum value</u> (signed/unsigned)	<u>maximum value</u> (signed/unsigned)
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32,768	32,767
		0	65,535
MEDIUMINT	3	-8,388,608	8,388,607
		0	16,777,215
INT	4	-2,147,483,648	2,147,483,647
		0	4,294,967,295
BIGINT	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
		0	18,446,744,073,709,551,615

# Integers

signed ≠ unsigned



if the encompassed range includes  
both positive and negative values

# Integers

<u>numeric data type</u>	<u>size (bytes)</u>	<u>minimum value</u> (signed/unsigned)	<u>maximum value</u> (signed/unsigned)
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32,768	32,767
		0	65,535
MEDIUMINT	3	-8,388,608	8,388,607
		0	16,777,215
INT	4	-2,147,483,648	2,147,483,647
		0	4,294,967,295
BIGINT	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
		0	18,446,744,073,709,551,615

# Integers

signed ≠ unsigned

if the encompassed range includes both positive and negative values

if integers are allowed to be only positive

# Integers

<u>numeric data type</u>	<u>size (bytes)</u>	<u>minimum value</u> (signed/unsigned)	<u>maximum value</u> (signed/unsigned)
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32,768	32,767
		0	65,535
MEDIUMINT	3	-8,388,608	8,388,607
		0	16,777,215
INT	4	-2,147,483,648	2,147,483,647
		0	4,294,967,295
BIGINT	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
		0	18,446,744,073,709,551,615

# Integers

- integer data types are ‘signed’ by default

# Integers

<u>numeric data type</u>	<u>size (bytes)</u>	<u>minimum value</u> (signed/unsigned)	<u>maximum value</u> (signed/unsigned)
TINYINT	1	-128	127
		0	256
SMALLINT	2	-32,768	32,767
		0	65,535
MEDIUMINT	3	-8,388,608	8,388,607
		0	16,777,215
INT	4	-2,147,483,648	2,147,483,647
		0	4,294,967,295
BIGINT	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
		0	18,446,744,073,709,551,615

# Integers

- integer data types are ‘signed’ by default
- if you want to use a range containing only positive, ‘unsigned’ values, you would have to specify this in your query

# Integers

<u>numeric data type</u>	<u>size (bytes)</u>	<u>minimum value</u> (signed/unsigned)	<u>maximum value</u> (signed/unsigned)
TINYINT	1	-128	127
		0	256
SMALLINT	2	-32,768	32,767
		0	65,535
MEDIUMINT	3	-8,388,608	8,388,607
		0	16,777,215
INT	4	-2,147,483,648	2,147,483,647
		0	4,294,967,295
BIGINT	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
		0	18,446,744,073,709,551,615

# Integers

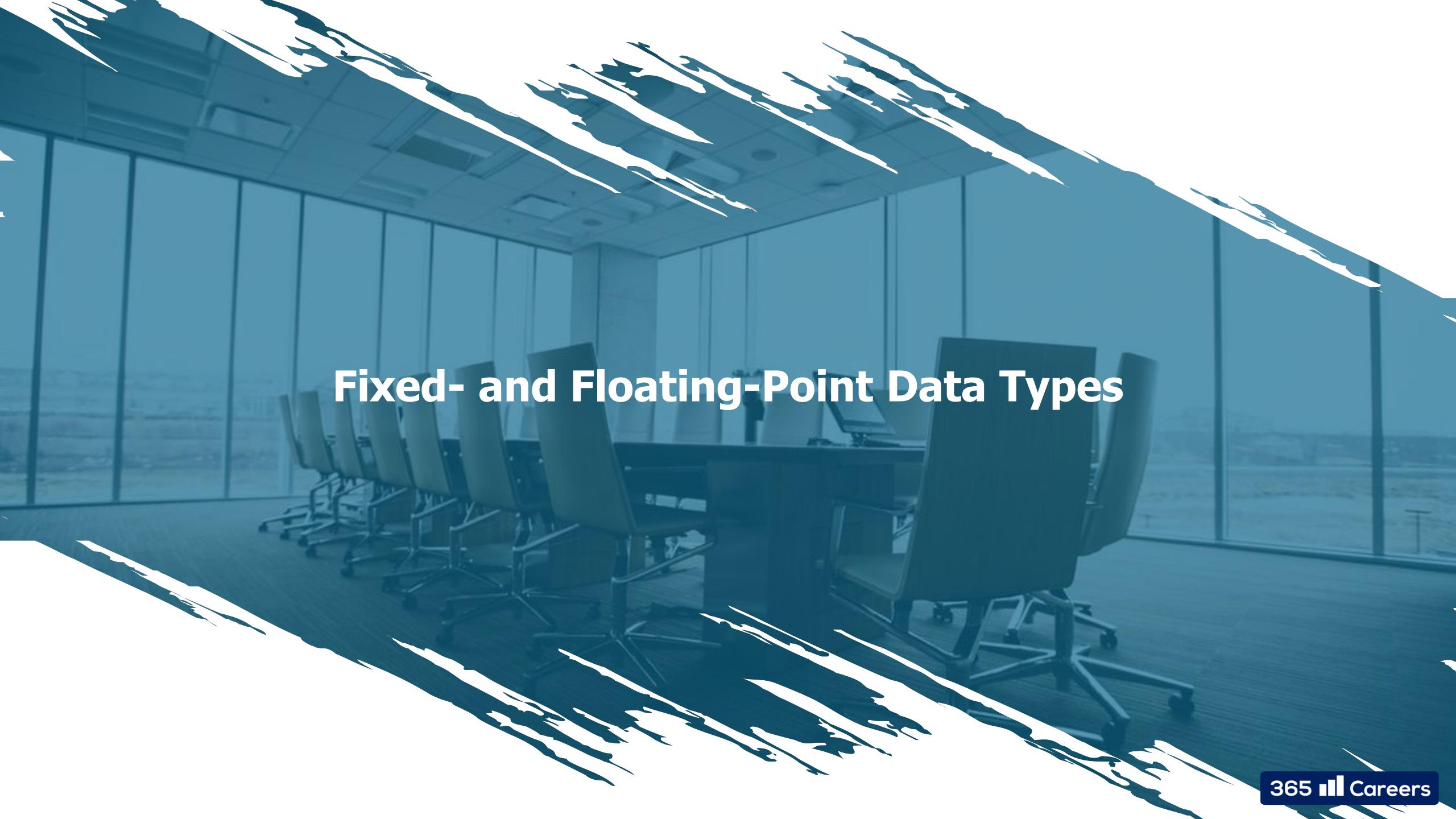
- Why not just use BIGINT all the time?

# Integers

- Why not just use **BIGINT** all the time?
- e.g. if you are sure that, in a certain column, you won't need an integer smaller than 0 or greater than 100, **TINYINT** would do the job perfectly and you would not need more storage space per data point

# Integers

- Why not just use **BIGINT** all the time?
- e.g. if you are sure that, in a certain column, you won't need an integer smaller than 0 or greater than 100, **TINYINT** would do the job perfectly and you would not need more storage space per data point
- a smaller integer type may increase the processing speed

A large, modern conference room is shown from a perspective looking down the length of the room. On the left, there are several rows of black office chairs with wheels, all facing towards the front of the room. The front wall of the room features a series of large, dark rectangular panels, likely projection screens or electronic displays. Above the chairs, the ceiling is white with a grid of recessed lighting fixtures. The floor is a light-colored wood or laminate. The overall atmosphere is professional and spacious.

# Fixed- and Floating-Point Data Types

# Fixed- and Floating-Point Data Types

<u>number:</u>	precision
10.523	5

precision

refers to the number of digits in a number

# Fixed- and Floating-Point Data Types

<u>number:</u>	precision	scale
10.523	5	3

**scale**

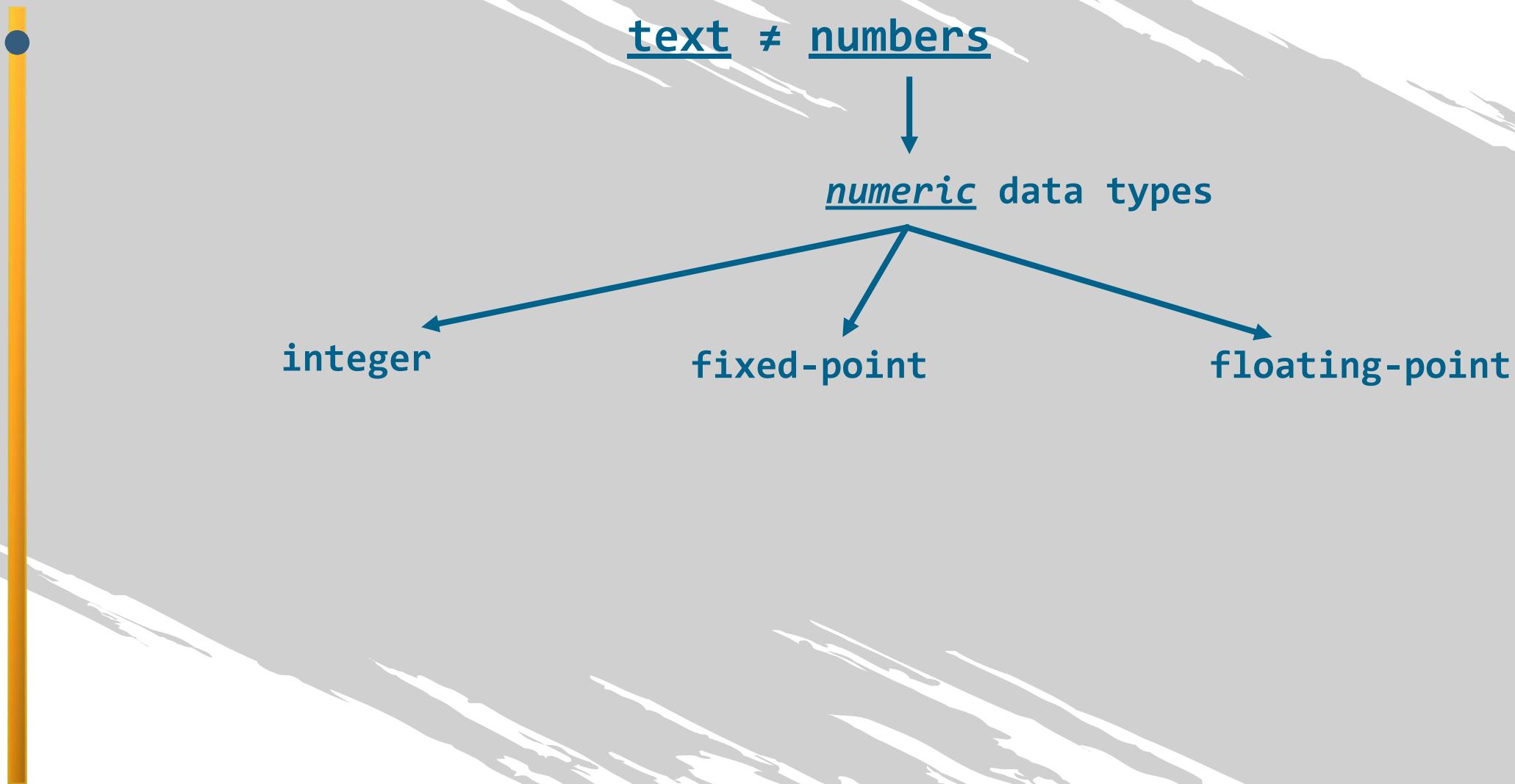
refers to the number of digits to the right of the decimal point in a number

# Fixed- and Floating-Point Data Types

<u>number:</u>	precision	scale
10.523	5	3
36.875	5	3

e.g. DECIMAL ( 5 , 3 )

# Fixed- and Floating-Point Data Types



# Fixed- and Floating-Point Data Types

- fixed-point data represent exact values

DECIMAL ( 5 , 3 )

10.523

10.5

10.500

10.5236789

10.524



# Fixed- and Floating-Point Data Types

- fixed-point data represent exact values

when only one digit is specified within the parentheses, it will be treated as the precision of the data type

**DECIMAL ( 7 )**

1234567

**DECIMAL ( 7, 0 )**

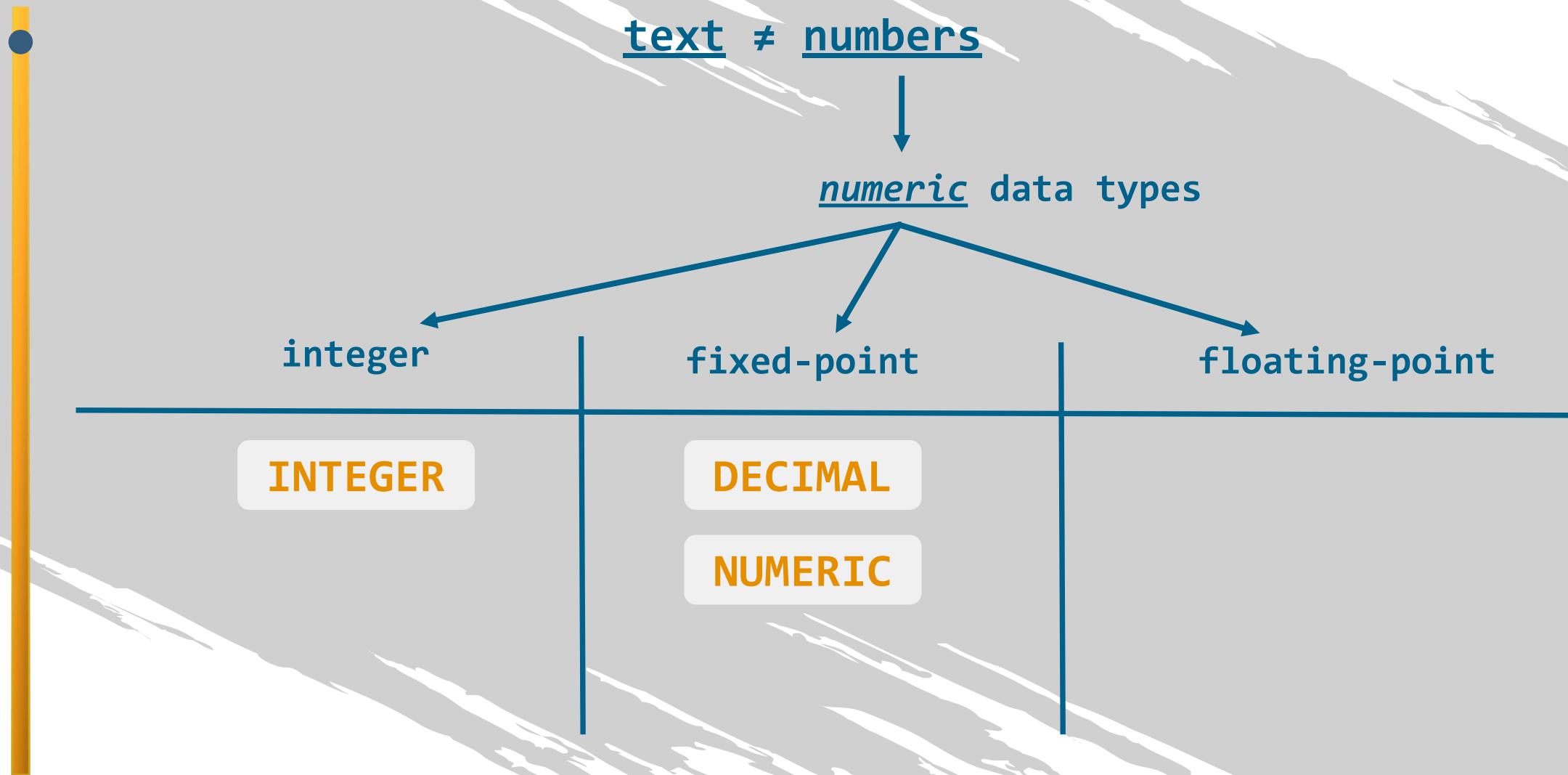
# Fixed- and Floating-Point Data Types

- fixed-point data represent exact values

DECIMAL has a synonymous data type. It is called NUMERIC.

**DECIMAL** = **NUMERIC**

# Fixed- and Floating-Point Data Types



# Fixed- and Floating-Point Data Types

• DECIMAL = NUMERIC

e.g. salaries

---

NUMERIC ( p , s )

precision: p = 7

scale: s = 2

---

e.g. NUMERIC (7,2) \$ 75,000.50

# Fixed- and Floating-Point Data Types

- floating-point data type

- used for approximate values only
- aims to balance between range and precision ( => “floating” )

FLOAT ( 5 , 3 )

10.5236789

10.523

10.524



(10.524 is an approximate value)

# Fixed- and Floating-Point Data Types

the main difference between the fixed- and the floating-point type is in the way the value is represented in the memory of the computer

**DECIMAL ( 5 , 3 )**

10.5236789

**FLOAT ( 5 , 3 )**

10.5236789

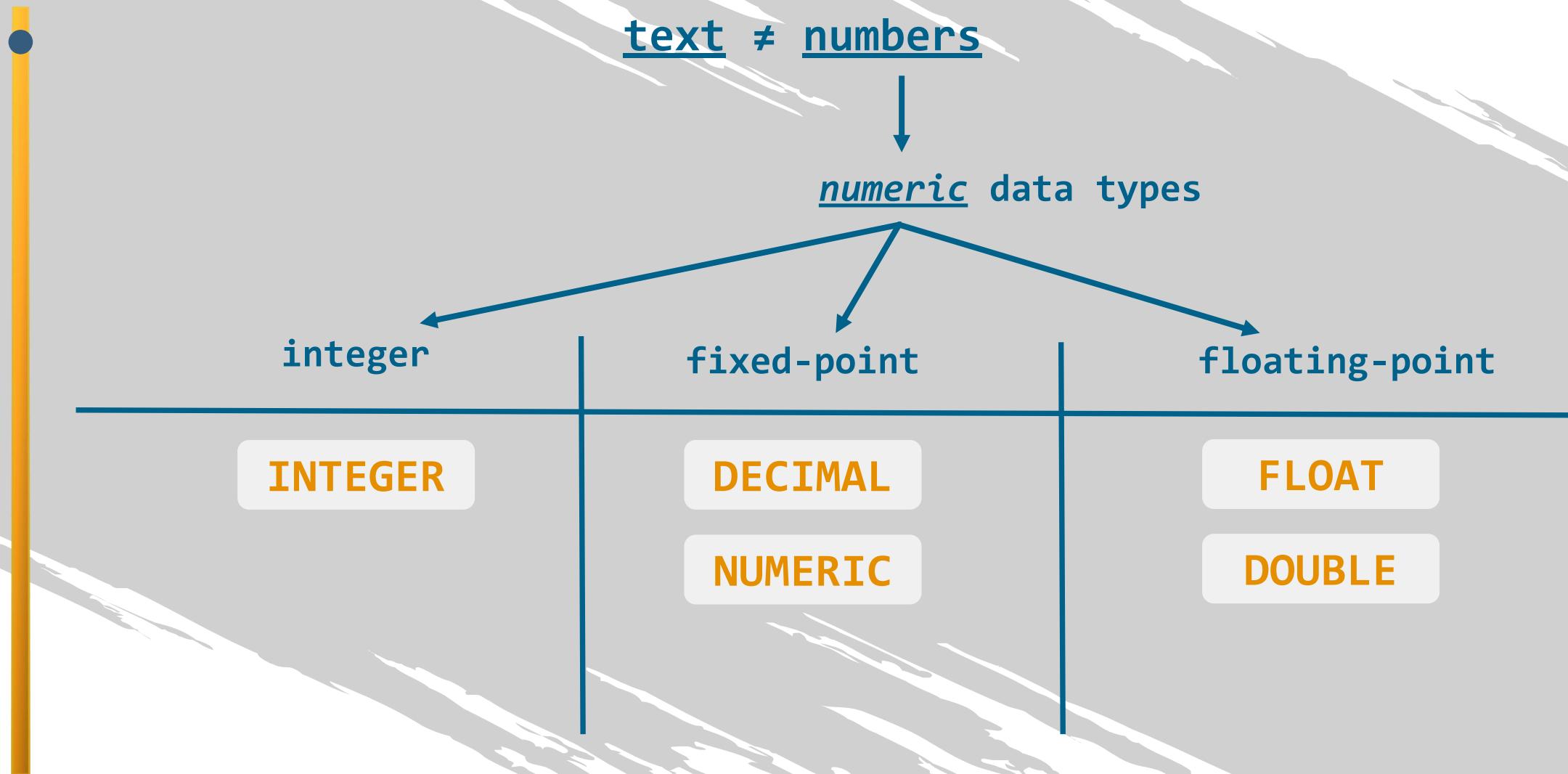
10.524



10.524



# Fixed- and Floating-Point Data Types



# Fixed- and Floating-Point Data Types

<u>Floating-point data type</u>	<u>size (bytes)</u>	<u>precision</u>	<u>maximum number of digits</u>
FLOAT	4	<i>single</i>	23
DOUBLE	8	<i>double</i>	53

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of black office chairs arranged facing a central rectangular table. The room has a high ceiling with a grid of recessed lighting. Large windows along the back wall provide a view of an outdoor area with trees and possibly a parking lot. The overall atmosphere is professional and spacious.

# Other Useful Data Types

# Other Useful Data Types

- DATE



used to represent a date in the format YYYY-MM-DD

1<sup>st</sup> of January 1000 - 31<sup>st</sup> of December 9999

---

e.g. 25<sup>th</sup> of July 2018: ‘2018-07-25’

# Other Useful Data Types

DATE +  = DATETIME

next to the date, we could save the time:

YYYY-MM-DD HH:MM:SS [.fraction]

0 - 23:59:59.99999

---

e.g. 25<sup>th</sup> of July 2018 9:30 a.m.: ‘2018-07-25 9:30:00’

# Other Useful Data Types

- **DATETIME**

represents the date shown on the calendar and the time shown on the clock

vs.

- **TIMESTAMP**

used for a *well-defined, exact point in time*

# Other Useful Data Types

## TIMESTAMP

used for a *well-defined, exact point in time*

1<sup>st</sup> of January 1970 UTC – 19<sup>th</sup> of January 2038, 03:14:07 UTC

- records the moment in time as the number of seconds passed after the 1<sup>st</sup> of January 1970 00:00:00 UTC

---

e.g. 25<sup>th</sup> of July 2018:

1,535,155,200

# Other Useful Data Types

## TIMESTAMP

- representing a moment in time as a number allows you to easily obtain the difference between two TIMESTAMP values

e.g. end time:

‘2018-07-25 10:30:00’ UTC

TIMESTAMP

start time:

‘2018-07-25 09:00:00’ UTC

TIMESTAMP

---

5,400

TIMESTAMP

# Other Useful Data Types

**TIMESTAMP**

is appropriate if you need to handle time zones

London



= 1:00 a.m

Paris



= 2:00 a.m

'1970-01-01 01:00:00' UTC

# Other Useful Data Types

*string, date, and time data types*

**CHAR**

**VARCHAR**

**DATE**

**DATETIME**

**TIMESTAMP**

**data must be written  
within quotes**

*numeric data types*

**INTEGER**

**DECIMAL**

**NUMERIC**

**FLOAT**

**DOUBLE**

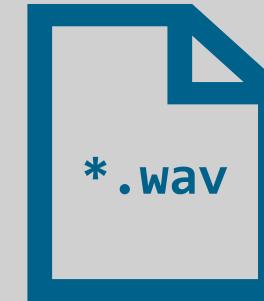
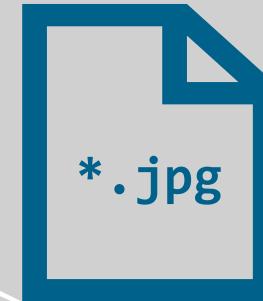
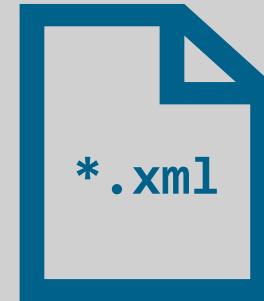
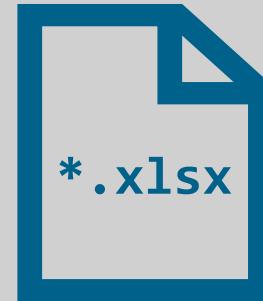
**only numeric values are  
written without quotes**

# Other Useful Data Types

## BLOB

### Binary Large OBject

- refers to a file of binary data - data with 1s and 0s
- involves saving *files* in a record



# Other Useful Data Types

Customers						
customer_id	first_name	last_name	email_address	number_of_complaints	photo	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0		
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2		
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1		
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0		*.jpg

# Other Useful Data Types

*string, date, and time data types*

*numeric data types*

**CHAR**

**VARCHAR**

**DATE**

**DATETIME**

**TIMESTAMP**

**INTEGER**

**DECIMAL**

**NUMERIC**

**FLOAT**

**DOUBLE**





# Creating a Table

# Creating a Table



SQL

```
CREATE DATABASE [IF NOT EXISTS] sales;
```

# Creating a Table



SQL

```
CREATE DATABASE [IF NOT EXISTS] sales;
```

```
CREATE TABLE table_name ( );
```

# Creating a Table



SQL

```
CREATE DATABASE [IF NOT EXISTS] sales;
```

```
CREATE TABLE table_name ( );
```

- compulsory requirement: add *at least one column*

# Creating a Table

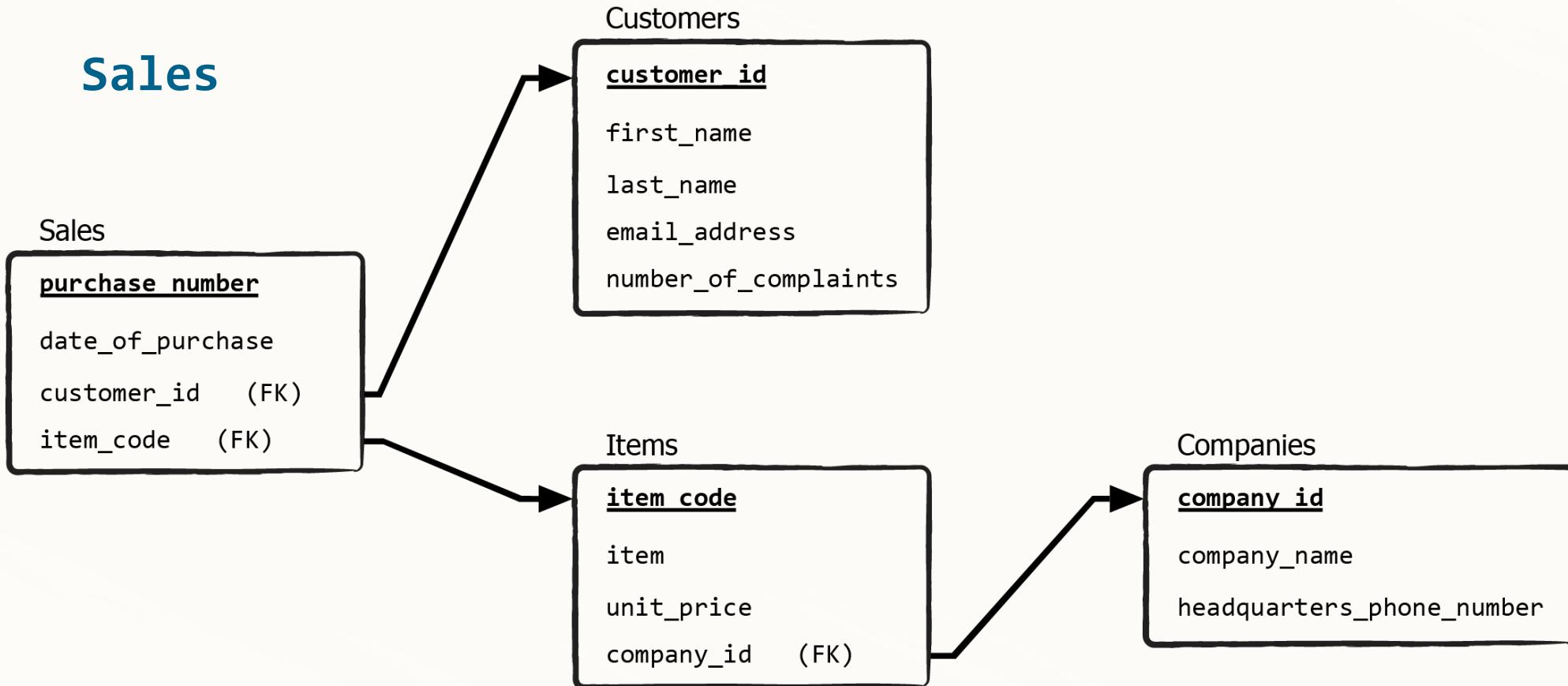


SQL

```
CREATE TABLE table_name
(
    column_1 data_type constraints,
    column_2 data_type constraints,
    ...
    column_n data_type constraints
);
```

# Creating a Table

## Sales



# Creating a Table

- **AUTO\_INCREMENT**

frees you from having to insert all purchase numbers manually through the INSERT command at a later stage

# Creating a Table

- **AUTO\_INCREMENT**

frees you from having to insert all purchase numbers manually through the INSERT command at a later stage

- assigns 1 to the first record of the table and automatically *increments* by 1 for every subsequent row

# Creating a Table

## AUTO\_INCREMENT

sales
purchase_number
1
2
3
4
...
n

# Creating a Table

## AUTO\_INCREMENT

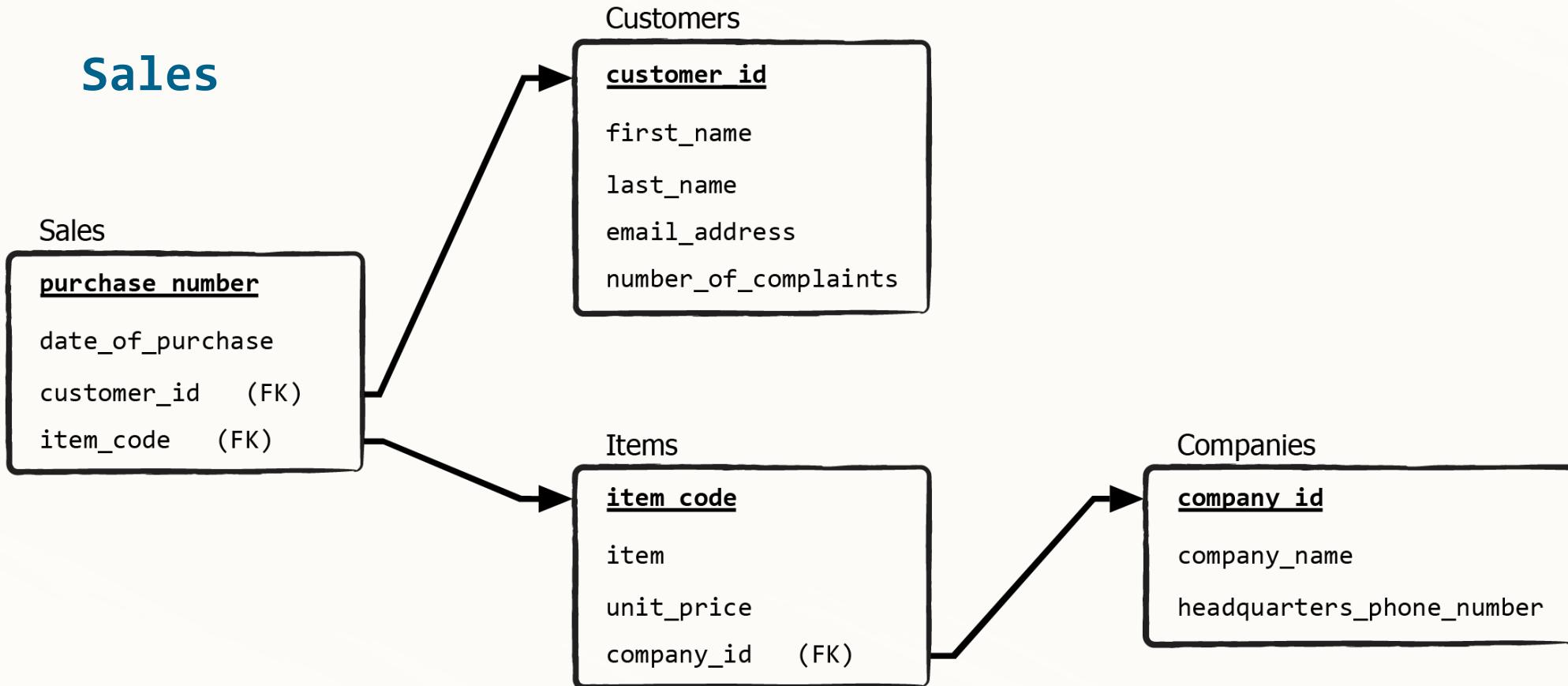
sales

purchase_number
1
2
3
4
...
n

SALES				
purchase_number	date_of_purchase	customer_id	item_code	
1	03/09/2016	1	A_1	
2	02/12/2016	2	C_1	
3	15/04/2017	3	D_1	
4	24/05/2017	1	B_2	
5	25/05/2017	4	B_2	
6	06/06/2017	2	B_1	
7	10/06/2017	4	A_2	
8	13/06/2017	3	C_1	
9	20/07/2017	1	A_1	
10	11/08/2017	2	B_1	

# Creating a Table

## Sales



A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of black office chairs facing towards the front. The front wall is a light-colored paneling that has been converted into a large display area, featuring several computer monitors showing different content. The room has a high ceiling with a grid of recessed lighting fixtures. Large windows along the right side provide a view of an outdoor area with some greenery and a clear sky.

# Using Databases and Tables

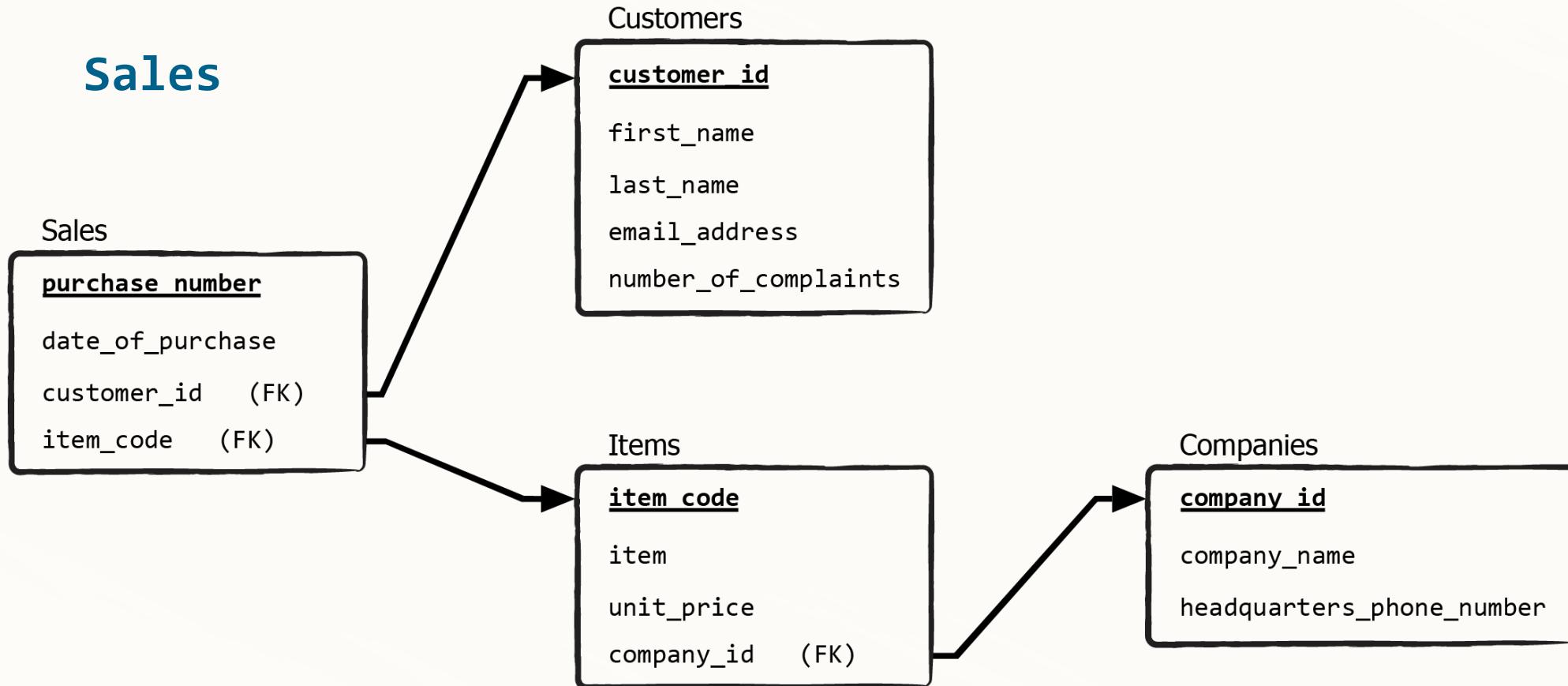
# Using Databases and Tables

- **queries**

one of their main features is to manipulate data within a database

# Using Databases and Tables

## Sales



# Using Databases and Tables

- queries

one of their main features is to manipulate data within a database

e.g.



SQL

```
SELECT * FROM customers;
```

# Using Databases and Tables

- Whenever you would like to refer to an *SQL object* in your queries, you must specify the database to which it is applied

# Using Databases and Tables

- Whenever you would like to refer to an *SQL object* in your queries, you must specify the database to which it is applied

---

## *SQL Objects:*

- SQL table

# Using Databases and Tables

- Whenever you would like to refer to an *SQL object* in your queries, you must specify the database to which it is applied

---

## *SQL Objects:*

- SQL table
- views
- stored procedures
- functions

# Using Databases and Tables

- Whenever you would like to refer to an *SQL object* in your queries, you must specify the database to which it is applied
  - 1) set a default database

# Using Databases and Tables

- Whenever you would like to refer to an *SQL object* in your queries, you must specify the database to which it is applied
- 1) set a default database



SQL

```
USE sales;  
SELECT * FROM customers;
```

# Using Databases and Tables

- Whenever you would like to refer to an *SQL object* in your queries, you must specify the database to which it is applied
  - 1) set a default database
  - 2) call a table from a certain database

# Using Databases and Tables

- Whenever you would like to refer to an *SQL object* in your queries, you must specify the database to which it is applied
  - 1) set a default database
  - 2) call a table from a certain database



SQL

database\_object . sql\_object

# Using Databases and Tables



SQL

database\_object . sql\_object

- . – “dot operator”  
signals the existence of a connection between the two object types

# Using Databases and Tables



SQL

database\_object . sql\_object

```
SELECT * FROM sales.customers;
```

. - “dot operator”

signals the existence of a connection between the two object types

The background image shows a modern conference room. In the foreground, there are two rows of black office chairs with wheels, facing towards the back of the room. The back wall features a large glass partition, through which a person can be seen standing near a whiteboard or screen. Above the partition, the ceiling has a grid of recessed lights. The overall atmosphere is professional and spacious.

## Additional Notes on Using Tables

# Additional Notes on Using Tables

## query

a command you write in SQL with the idea of either *retrieving information* from the database on which you are working, or, alternatively, to *insert*, *update*, or *delete* data from it

# Additional Notes on Using Tables

## query

a command you write in SQL with the idea of either *retrieving information* from the database on which you are working, or, alternatively, to *insert*, *update*, or *delete* data from it

- it is a representation of a complete logical thought

# Additional Notes on Using Tables

- the DROP statement  
used for deleting an SQL object

# Additional Notes on Using Tables

- the DROP statement

used for deleting an SQL object



SQL

```
DROP TABLE table_name;
```

# Additional Notes on Using Tables

- the DROP statement

used for deleting an SQL object



SQL

```
DROP TABLE table_name;
```

```
DROP TABLE sales;
```

# MySQL Constraints

A large conference room with rows of chairs facing a large screen. The room has a modern design with a glass partition and a wooden floor. The text "PRIMARY KEY Constraint" is overlaid on the image.

# PRIMARY KEY Constraint

# PRIMARY KEY Constraint

So far:

# PRIMARY KEY Constraint

## So far:

- Theory of Relational Databases

# PRIMARY KEY Constraint

## So far:

- Theory of Relational Databases
- SQL Theory

# PRIMARY KEY Constraint

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating a Database

# PRIMARY KEY Constraint

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating a Database

## In this section:

# PRIMARY KEY Constraint

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating a Database

## In this section:

- Constraints

# PRIMARY KEY Constraint

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating a Database

## In this section:

- Constraints

## In this lesson:

# PRIMARY KEY Constraint

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating a Database

## In this section:

- Constraints

## In this lesson:

- the PRIMARY KEY Constraint

# PRIMARY KEY Constraint

## constraints

specific rules, or limits, that we define in our tables

# PRIMARY KEY Constraint

## constraints

specific rules, or limits, that we define in our tables

- the role of constraints is to outline the existing relationships between different tables in our database

# PRIMARY KEY Constraint

## constraints

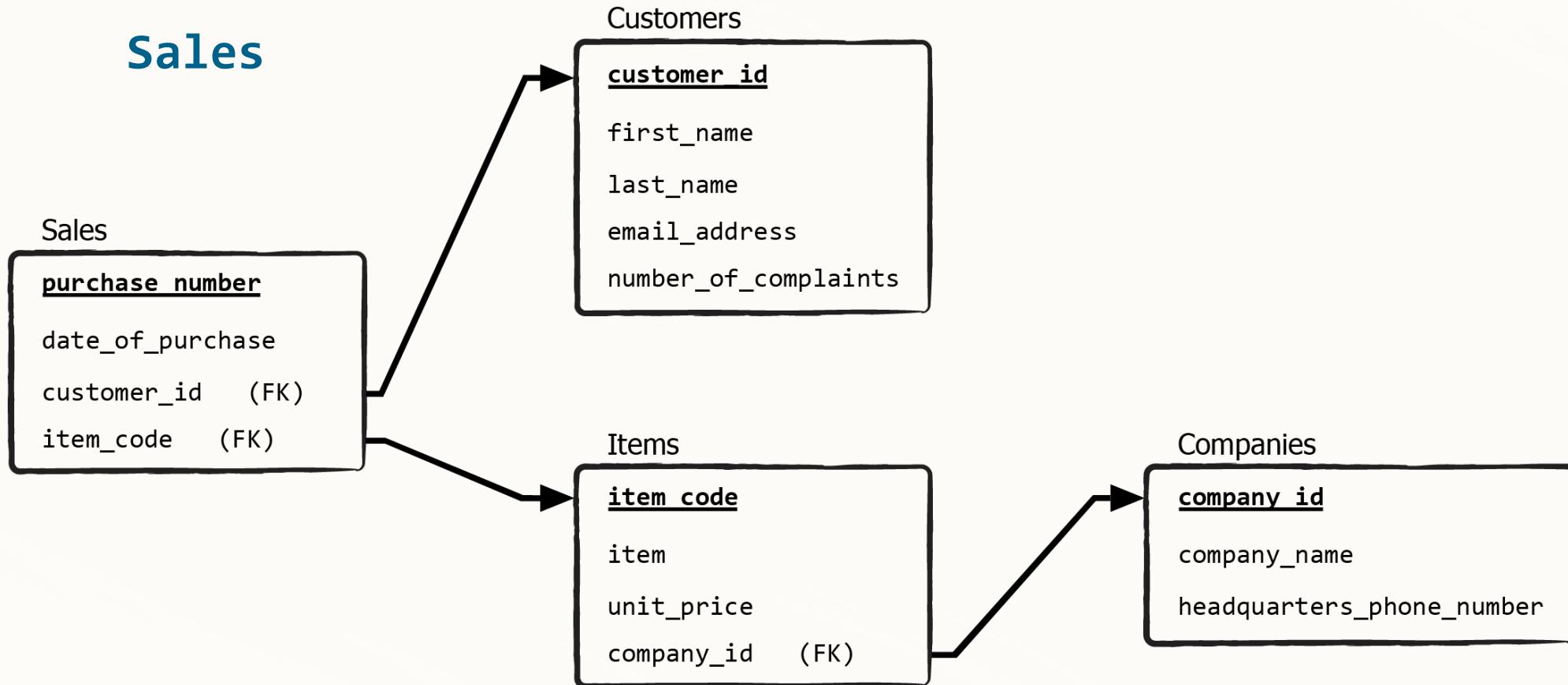
specific rules, or limits, that we define in our tables

- the role of constraints is to outline the existing relationships between different tables in our database

e.g. NOT NULL

# PRIMARY KEY Constraint

## Sales



A large conference room with rows of chairs facing a large screen. The room has a modern design with a glass partition and a large window overlooking a landscape.

# FOREIGN KEY Constraint

# FOREIGN KEY Constraint

## Sales

Sales

**purchase number**

date\_of\_purchase

customer\_id (FK)

item\_code (FK)

Customers

**customer id**

first\_name

last\_name

email\_address

number\_of\_complaints

Items

**item code**

item

unit\_price

company\_id (FK)

Companies

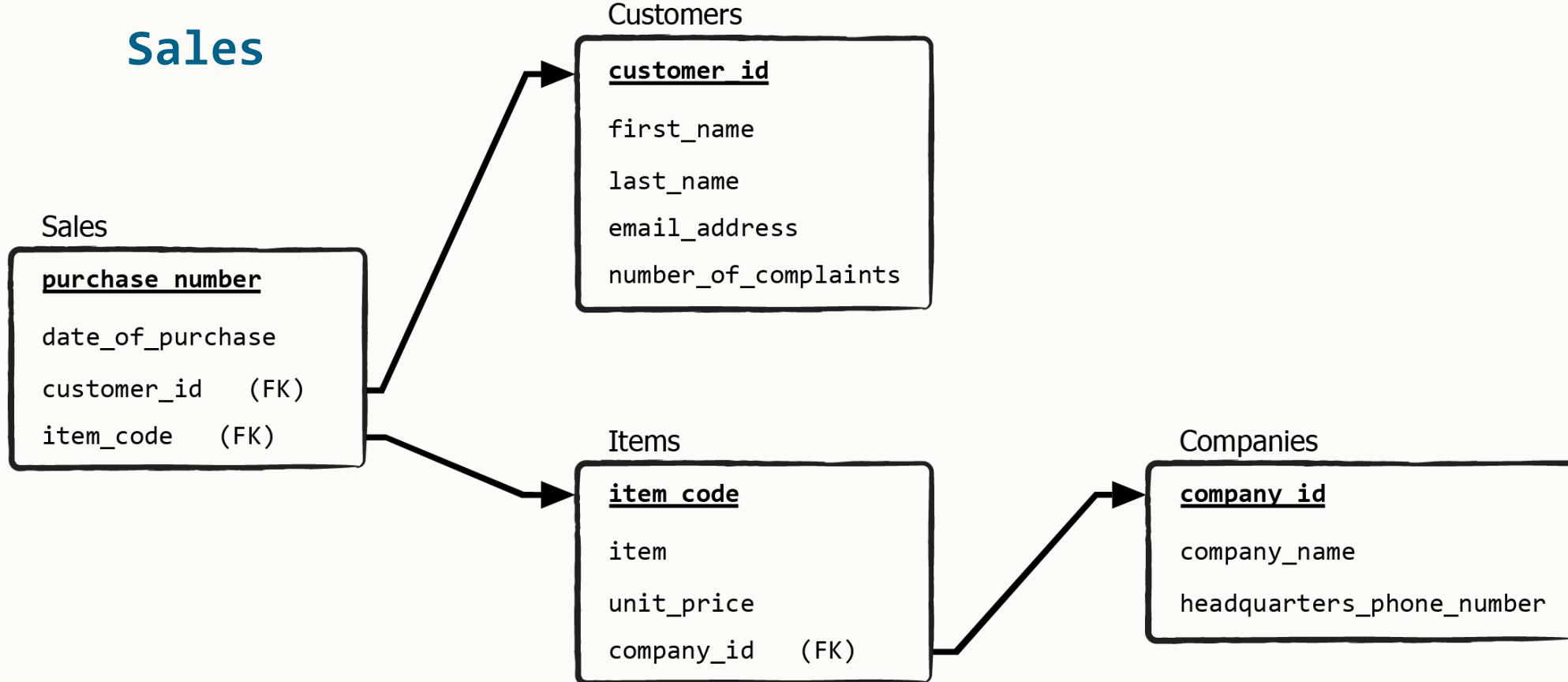
**company id**

company\_name

headquarters\_phone\_number

# FOREIGN KEY Constraint

## Sales



# FOREIGN KEY Constraint

**foreign key**

points to a column of another table and, thus, links the two tables

# FOREIGN KEY Constraint

Table 1

column\_name

Table 2

column\_name

# FOREIGN KEY Constraint

Table 1

column_name
-------------

child table

Table 2

column_name
-------------

# FOREIGN KEY Constraint

Table 1

column\_name

child table

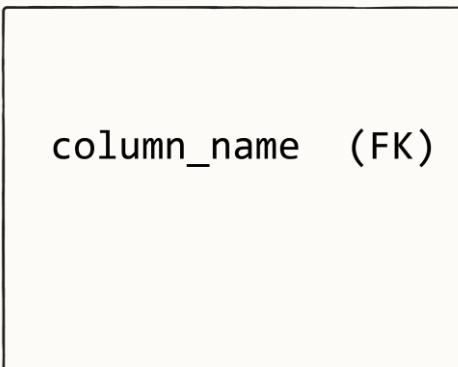
Table 2

column\_name

parent table

# FOREIGN KEY Constraint

Table 1



child table

Table 2

column\_name

parent table

# FOREIGN KEY Constraint

Table 1

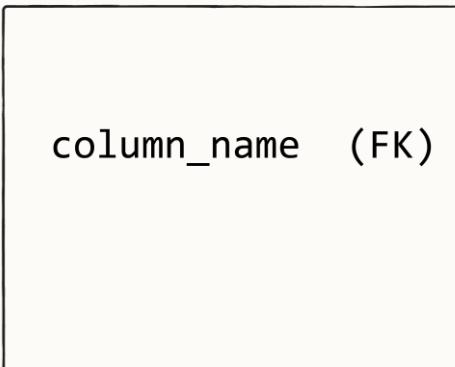


Table 2

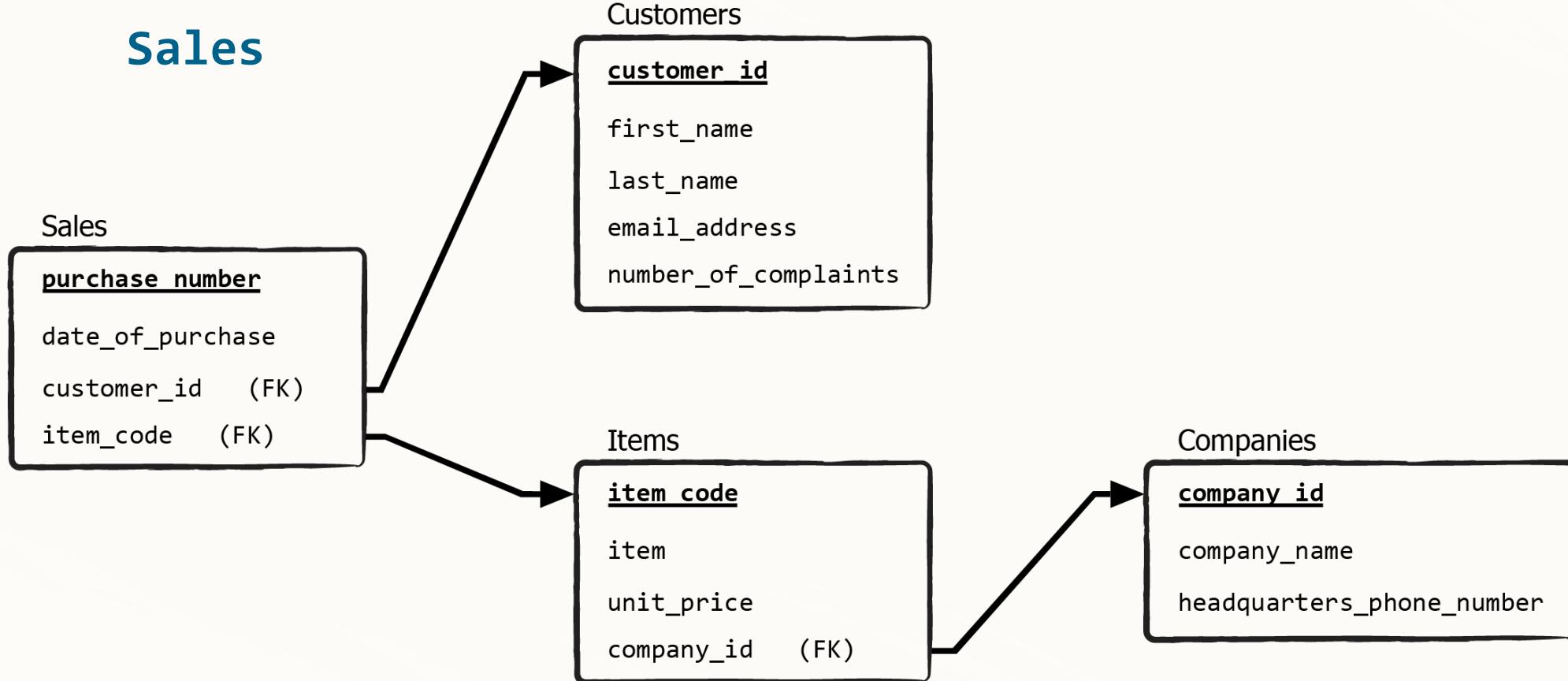
column\_name

parent table = referenced table

child table = referencing table

# FOREIGN KEY Constraint

## Sales



# FOREIGN KEY Constraint

## Sales

### Sales

**purchase number**  
date\_of\_purchase  
customer\_id  
item\_code

### Customers

**customer\_id**  
first\_name  
last\_name  
email\_address  
number\_of\_complaints

# FOREIGN KEY Constraint

## Sales

### Sales

**purchase number**

date\_of\_purchase

customer\_id (FK)

item\_code

### Customers

**customer\_id**

first\_name

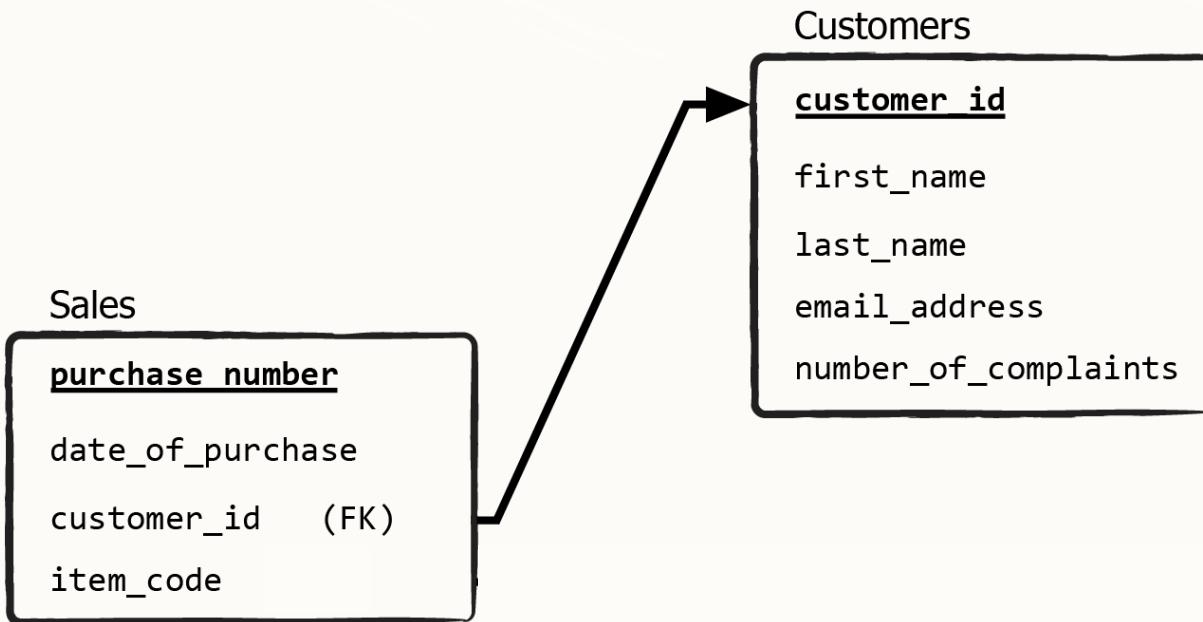
last\_name

email\_address

number\_of\_complaints

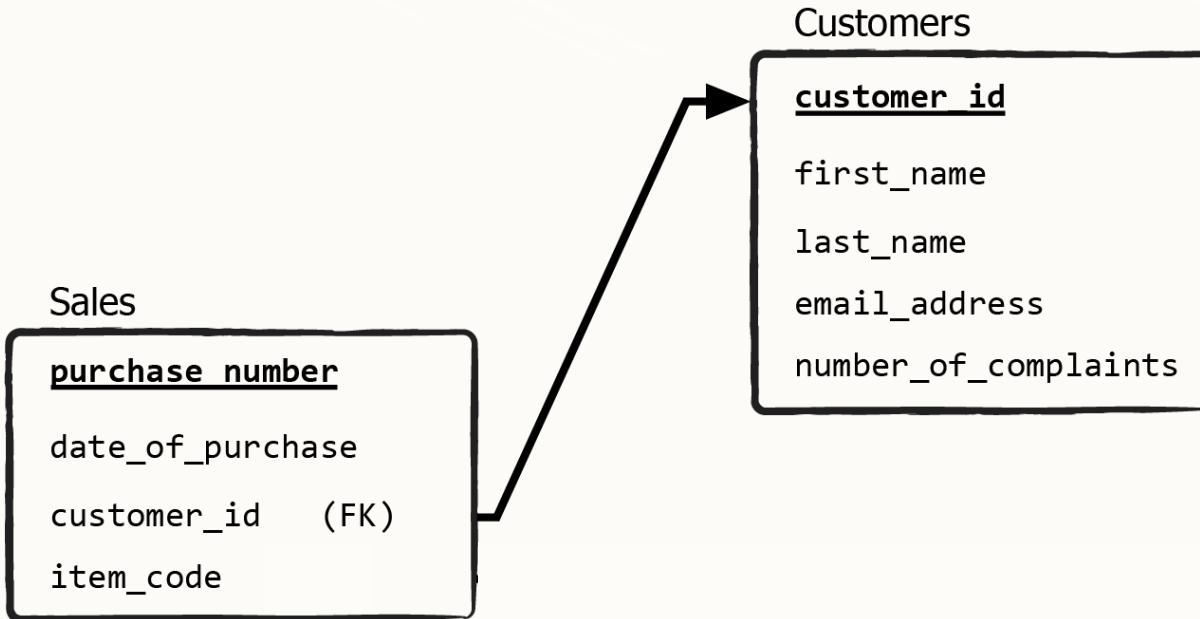
# FOREIGN KEY Constraint

## Sales



# FOREIGN KEY Constraint

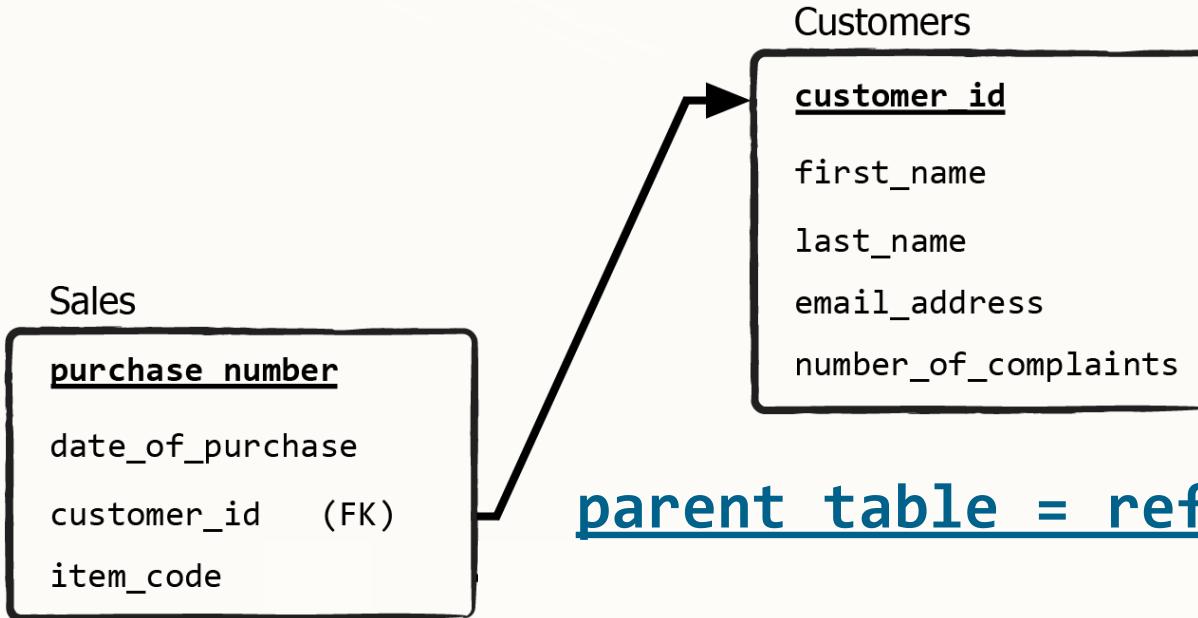
## Sales



*Remember, this is not an obligatory requirement - these two keys may have two completely different names. What's important is that the data types and the information match! It's just common practice to use, if not the same, then similar names for both keys.*

# FOREIGN KEY Constraint

Sales



parent table = referenced table

child table = referencing table

# FOREIGN KEY Constraint

- a *foreign key* in SQL is defined through a foreign key constraint

# FOREIGN KEY Constraint

- a *foreign key* in SQL is defined through a foreign key constraint

the foreign key maintains the *referential integrity* within the database

# FOREIGN KEY Constraint

- **ON DELETE CASCADE**

*if a specific value from the parent table's primary key has been deleted, all the records from the child table referring to this value will be removed as well*

# FOREIGN KEY Constraint

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# FOREIGN KEY Constraint

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017		B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/13/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# FOREIGN KEY Constraint

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/13/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# FOREIGN KEY Constraint

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	

## ON DELETE CASCADE

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
6	6/6/2017	2	B_1	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

A large conference room with rows of chairs facing a front wall with a large screen. The room has large windows overlooking a landscape.

# UNIQUE Constraint

# UNIQUE Constraint

## unique key

used whenever you would like to specify that you don't want to see duplicate data in a given field

# UNIQUE Constraint

## unique key

used whenever you would like to specify that you don't want to see duplicate data in a given field

- ensures that all values in a column (or a set of columns) are different

# UNIQUE Constraint

- *unique keys* are implemented in SQL through a constraint - the UNIQUE Constraint

# UNIQUE Constraint

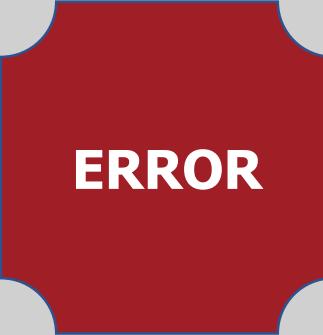
- *unique keys* are implemented in SQL through a constraint - the UNIQUE Constraint

*if you attempt to insert an already existing, duplicate value in the unique column, SQL will display an error*

# UNIQUE Constraint

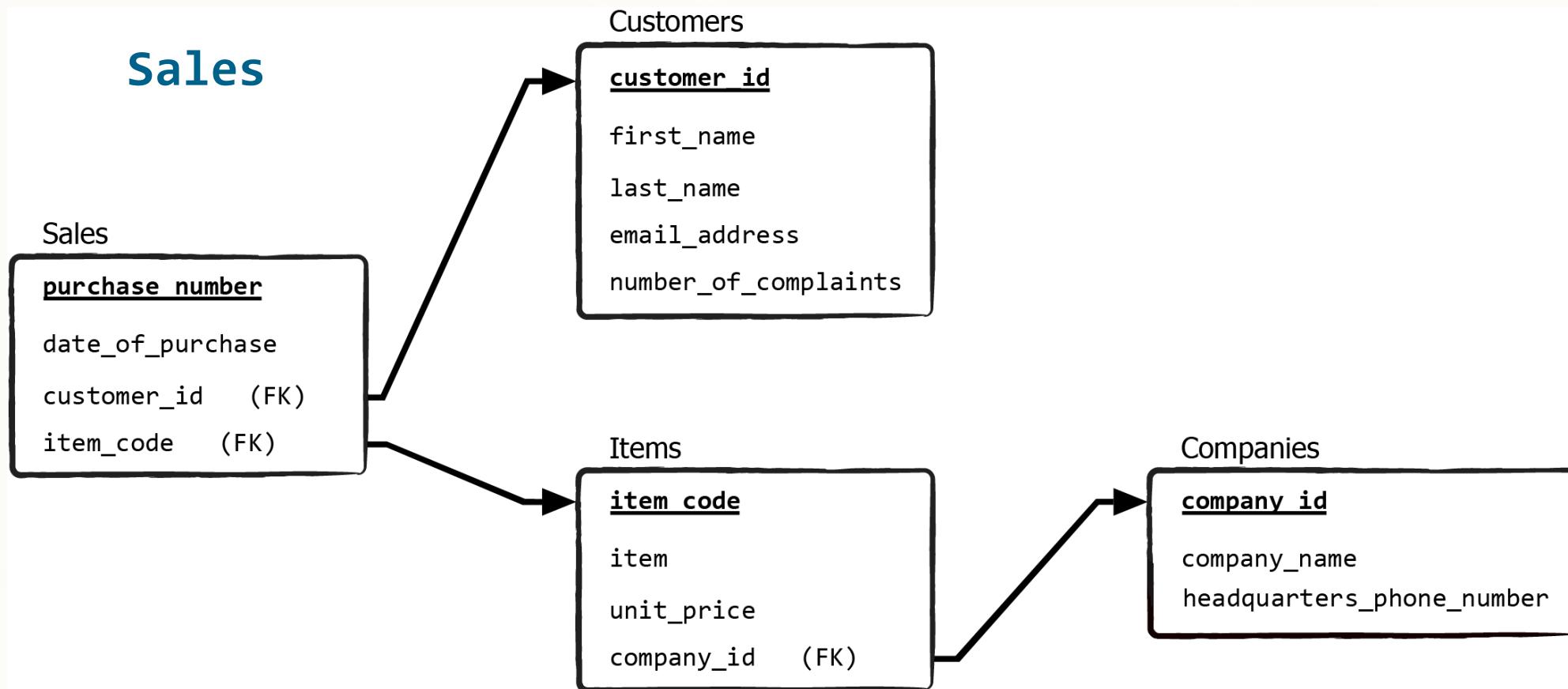
- *unique keys* are implemented in SQL through a constraint - the UNIQUE Constraint

*if you attempt to insert an already existing, duplicate value in the unique column, SQL will display an error*



ERROR

# UNIQUE Constraint



# UNIQUE Constraint

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# UNIQUE Constraint

- Indexes

# UNIQUE Constraint

- **Indexes**

*unique keys* in MySQL have the same role as *indexes*

# UNIQUE Constraint

- **Indexes**

*unique keys* in MySQL have the same role as *indexes*

the reverse isn't true !!!

# UNIQUE Constraint

- **Indexes**

*unique keys* in MySQL have the same role as *indexes*

the reverse isn't true !!!

index of a table

an organizational unit that helps retrieve data more easily

# UNIQUE Constraint

- **Indexes**

*unique keys* in MySQL have the same role as *indexes*

the reverse isn't true !!!

index of a table

an organizational unit that helps retrieve data more easily

- it takes more time to update a table because indexes must be updated, too, and that's time consuming

# UNIQUE Constraint

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# UNIQUE Constraint



SQL

```
ALTER TABLE table_name  
DROP INDEX unique_key_field;
```

A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

# DEFAULT Constraint

# DEFAULT Constraint

- **DEFAULT Constraint**

# DEFAULT Constraint

- **DEFAULT Constraint**

helps us assign a particular default value to every row of a column

# DEFAULT Constraint

- **DEFAULT Constraint**

helps us assign a particular default value to every row of a column

- a value different from the default can be stored in a field where the DEFAULT constraint has been applied, *only if specifically indicated.*

# DEFAULT Constraint

Customers					
customer_id	first_name	last_name	email_address	number_of_complaints	
1	John	McKinley	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# DEFAULT Constraint



SQL

```
CREATE TABLE customers
(
    customer_id INT,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    email_address VARCHAR(255),
    number_of_complaints INT DEFAULT 0,
    PRIMARY KEY (customer_id)
);
```

# DEFAULT Constraint

Customers						
customer_id	first_name	last_name	gender		email_address	number_of_complaints
1	John	McKinley	M		<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0

# DEFAULT Constraint

Customers						
customer_id	first_name	last_name	gender		email_address	number_of_complaints
1	John	McKinley	M		<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0
2						

# DEFAULT Constraint

Customers						
customer_id	first_name	last_name	gender		email_address	number_of_complaints
1	John	McKinley	M		<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0
2	Peter	Figaro	M			

# Data Definition Language

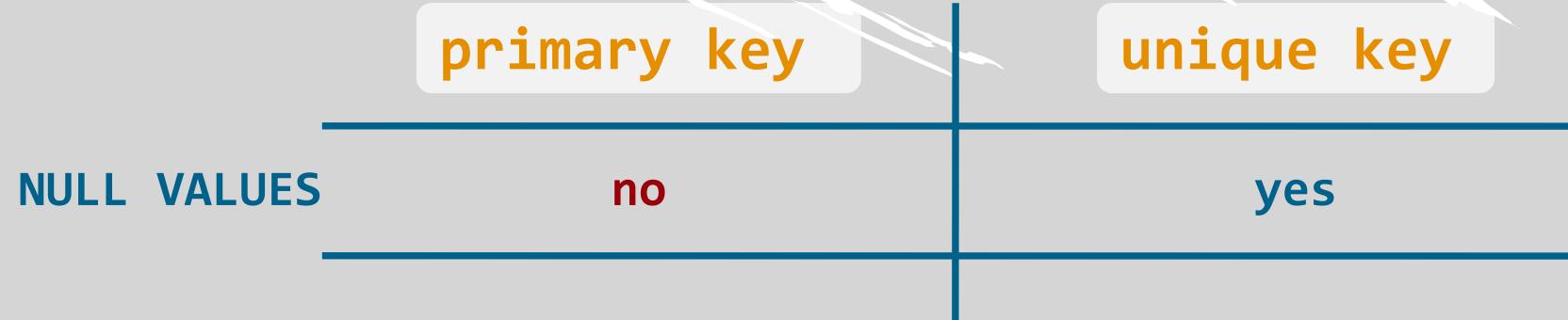
- **Data Definition Language (DDL)**

- CREATE
- ALTER
- DROP

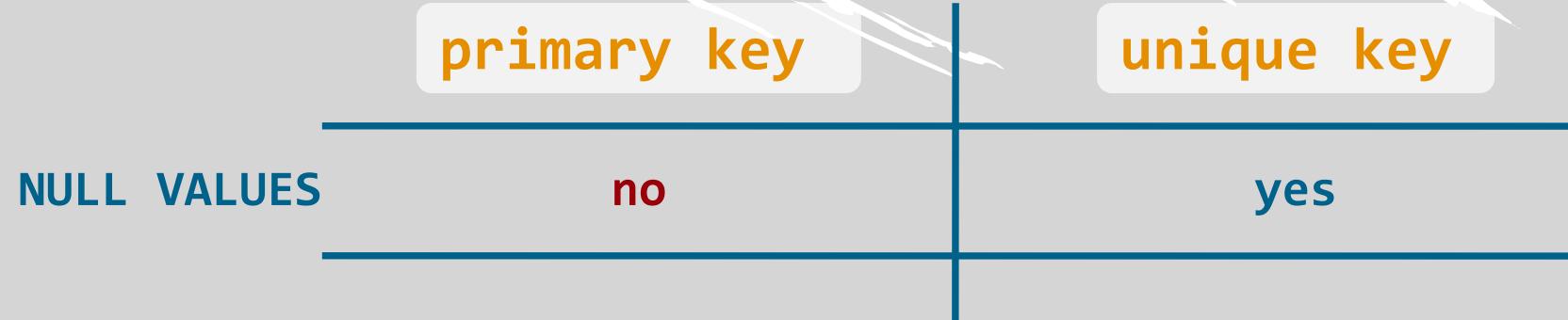
A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

# NOT NULL Constraint

# NOT NULL Constraint



# NOT NULL Constraint



the “not null” restriction is applied through the NOT NULL Constraint

# NOT NULL Constraint

	primary key	unique key
NULL VALUES	no	yes

the “not null” restriction is applied through the NOT NULL Constraint  
- when you insert values in the table, you cannot leave the respective field *empty*

# NOT NULL Constraint

	primary key	unique key
NULL VALUES	no	yes

the “not null” restriction is applied through the NOT NULL Constraint

- when you insert values in the table, you cannot leave the respective field *empty*
- if you leave it *empty*, MySQL will signal an error

ERROR

# NOT NULL Constraint

Companies		
company_id	headquarters_phone_number	company_name
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

# NOT NULL Constraint

Companies		
company_id	headquarters_phone_number	company_name
1	+1 (202) 555-0196	Company A
2	+1 (202) 555-0152	Company B
3	+1 (229) 853-9913	Company C
4	+1 (618) 369-7392	Company D

NOT NULL

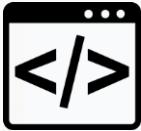
# NOT NULL Constraint



SQL

```
CREATE TABLE companies
(
    company_id INT AUTO_INCREMENT,
    headquarters_phone_number VARCHAR(255),
    company_name VARCHAR(255),
    PRIMARY KEY (company_id)
);
```

# NOT NULL Constraint



SQL

```
CREATE TABLE companies
(
    company_id INT AUTO_INCREMENT,
    headquarters_phone_number VARCHAR(255),
    company_name VARCHAR(255) NOT NULL,
    PRIMARY KEY (company_id)
);
```

# NOT NULL Constraint

- Don't confuse a NULL value with the value of 0 or with a "NONE" response!

*Think of a null value as a missing value.*



# NOT NULL Constraint

Customers						
customer_id	first_name	last_name	gender	email_address	number_of_complaints	
1	John	McKinley	M	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	
2	Elizabeth	McFarlane	F	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	M	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	F	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

**NOT NULL**  
the value  
of 0,  
**NOT NULL**

# NOT NULL Constraint

Customers						
customer_id	first_name	last_name	gender	email_address	number_of_complaints	
1	John	McKinley	M	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	NULL	
2	Elizabeth	McFarlane	F	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	
3	Kevin	Lawrence	M	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	
4	Catherine	Winnfield	F	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	

# NOT NULL Constraint

Customers							
customer_id	first_name	last_name	gender	email_address	number_of_complaints	feedback	
1	John	McKinley	M	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	I think...	
2	Elizabeth	McFarlane	F	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	Great service!	
3	Kevin	Lawrence	M	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	Great service!	
4	Catherine	Winnfield	F	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0	NONE	

NULL

# NOT NULL Constraint

Customers							
customer_id	first_name	last_name	gender	email_address	number_of_complaints	feedback	
1	John	McKinley	M	<a href="mailto:john.mackinley@365careers.com">john.mackinley@365careers.com</a>	0	<i>I think...</i>	
2	Elizabeth	McFarlane	F	<a href="mailto:e.mcfarlane@365careers.com">e.mcfarlane@365careers.com</a>	2	<i>Great service!</i>	
3	Kevin	Lawrence	M	<a href="mailto:kevin.lawrence@365careers.com">kevin.lawrence@365careers.com</a>	1	<i>Great service!</i>	
4	Catherine	Winnfield	F	<a href="mailto:c.winnfield@365careers.com">c.winnfield@365careers.com</a>	0		

NULL

Next:

Coding Techniques and Best Practices

Data Manipulation



# SQL Best Practices

A large, modern conference room is shown from a perspective looking down the length of the room. On the left, there are several rows of black office chairs facing towards the right. The room has a long, dark rectangular table in the center. At the far end of the room, there is a wall with three large, vertical, light-colored panels that appear to be touchscreens or projection screens. The ceiling is white with a grid of recessed lighting. The overall atmosphere is professional and tech-oriented.

# Coding Techniques and Best Practices

# Coding Techniques and Best Practices

complying with coding style is *crucial*

# Coding Techniques and Best Practices

complying with coding style is *crucial*

- you will always work in a team

# Coding Techniques and Best Practices

## clean code

code that is *focused* and *understandable*, which means it must be readable, logical, and changeable

# Coding Techniques and Best Practices

- good code is not the one computers understand; it is the one *humans* can understand

# Coding Techniques and Best Practices

- good code is not the one computers understand; it is the one humans can understand

code, in general, can be organized in several ways

# Coding Techniques and Best Practices

- good code is not the one computers understand; it is the one humans can understand

code, in general, can be organized in several ways



good practice implies you will choose the version that will be easiest to read and understand

# Coding Techniques and Best Practices

- good code is not the one computers understand; it is the one humans can understand

code, in general, can be organized in several ways



good practice implies you will choose the version that will be easiest to read and understand

assumption:

at your workplace, you will always type code cleanly -  
*as simple as possible, perfectly organized, maintaining a steady logical flow*

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects,*

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects, always chose shorter, meaningful names, conveying specific information*

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects, always chose shorter, meaningful names, conveying specific information*
  - ↓
    - pronounceable, where one word per concept has been picked*

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects,  
always chose shorter, meaningful names, conveying specific information*



*pronounceable, where one word per concept has been picked*

Sales			
purchase_number	date_of_purchase	customer_id	item_code
1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects, always chose shorter, meaningful names, conveying specific information*



*pronounceable, where one word per concept has been picked*

`customer_purchase_unique_number`

Sales				
	<code>purchase_number</code>	<code>date_of_purchase</code>	<code>customer_id</code>	<code>item_code</code>
<code>customer_purchase_unique_number</code>	1	9/3/2016	1	A_1
	2	12/2/2016	2	C_1
	3	4/15/2017	3	D_1
	4	5/24/2017	1	B_2
	5	5/25/2017	4	B_2
	6	6/6/2017	2	B_1
	7	6/10/2017	4	A_2
	8	6/10/2017	3	C_1
	9	7/20/2017	1	A_1
	10	8/11/2017	2	B_1

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects, always chose shorter, meaningful names, conveying specific information*



*pronounceable, where one word per concept has been picked*

**customer\_purchase\_unique\_number**

Sales				
	purchase_number	date_of_purchase	customer_id	item_code
	1	9/3/2016	1	A_1
	2	12/2/2016	2	C_1
	3	4/15/2017	3	D_1
	4	5/24/2017	1	B_2
	5	5/25/2017	4	B_2
	6	6/6/2017	2	B_1
	7	6/10/2017	4	A_2
	8	6/10/2017	3	C_1
	9	7/20/2017	1	A_1
	10	8/11/2017	2	B_1

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects,  
always chose shorter, meaningful names, conveying specific information*



*pronounceable, where one word per concept has been picked*

customer\_purchase\_unique\_number



Sales				
	purchase_number	date_of_purchase	customer_id	item_code
	1	9/3/2016	1	A_1
	2	12/2/2016	2	C_1
	3	4/15/2017	3	D_1
	4	5/24/2017	1	B_2
	5	5/25/2017	4	B_2
	6	6/6/2017	2	B_1
	7	6/10/2017	4	A_2
	8	6/10/2017	3	C_1
	9	7/20/2017	1	A_1
	10	8/11/2017	2	B_1

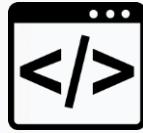
# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects, always chose shorter, meaningful names, conveying specific information*
  - ↓
    - pronounceable, where one word per concept has been picked*

# Coding Techniques and Best Practices

- *when assigning names to variables or SQL objects, always chose shorter, meaningful names, conveying specific information*
  - pronounceable, where one word per concept has been picked

# Coding Techniques and Best Practices



SQL

```
CREATE TABLE sales
(
    purchase_number INT,
    date_of_purchase DATE,
    customer_id VARCHAR(255),
    item_code VARCHAR(255),
    PRIMARY KEY (purcase_number)
);
```

# Coding Techniques and Best Practices

Sales				
<u>purchase_number</u>	<u>date_of_purchase</u>	<u>customer_id</u>	<u>item_code</u>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Coding Techniques and Best Practices

purchase\_number

Sales				
purchase_number	date_of_purchase	customer_id	item_code	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Coding Techniques and Best Practices

`purchase_number`

`PurchaseNumber`

Sales				
<code>purchase_number</code>	<code>date_of_purchase</code>	<code>customer_id</code>	<code>item_code</code>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Coding Techniques and Best Practices

`purchase_number`

`PurchaseNumber`

`purchase number`

Sales				
<code>purchase_number</code>	<code>date_of_purchase</code>	<code>customer_id</code>	<code>item_code</code>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Coding Techniques and Best Practices

`purchase_number`

`PurchaseNumber`

`purchasXnumber`

**ERROR**

Sales				
<code>purchase_number</code>	<code>date_of_purchase</code>	<code>customer_id</code>	<code>item_code</code>	
1	9/3/2016	1	A_1	
2	12/2/2016	2	C_1	
3	4/15/2017	3	D_1	
4	5/24/2017	1	B_2	
5	5/25/2017	4	B_2	
6	6/6/2017	2	B_1	
7	6/10/2017	4	A_2	
8	6/10/2017	3	C_1	
9	7/20/2017	1	A_1	
10	8/11/2017	2	B_1	

# Coding Techniques and Best Practices

- readability

# Coding Techniques and Best Practices

- **readability**

- horizontal and vertical organization of code

# Coding Techniques and Best Practices

- **readability**

- horizontal and vertical organization of code
- colour

# Coding Techniques and Best Practices



use ad-hoc software that re-organizes code and colours different words consistently

# Coding Techniques and Best Practices



use ad-hoc software that re-organizes code and colours different words consistently

- time is a factor

# Coding Techniques and Best Practices



use ad-hoc software that re-organizes code and colours different words consistently

- time is a factor
- unification of coding style is a top-priority

# Coding Techniques and Best Practices



use ad-hoc software that re-organizes code and colours different words consistently

- time is a factor
- unification of coding style is a top-priority

*it is unprofessional to merge code written in the same language but in a different style*

# Coding Techniques and Best Practices



use ad-hoc software that re-organizes code and colours different words consistently

# Coding Techniques and Best Practices



use ad-hoc software that re-organizes code and colours different words consistently



use the relevant analogical tool provided in Workbench

# Coding Techniques and Best Practices



use ad-hoc software that re-organizes code and colours different words consistently



use the relevant analogical tool provided in Workbench



intervene manually and adjust your code as you like

# Coding Techniques and Best Practices

## Comments

lines of text that Workbench will not run as code; they convey a message to someone who reads our code

# Coding Techniques and Best Practices

## Comments

lines of text that Workbench will not run as code; they convey a message to someone who reads our code

`/* ... */`      *(for Large comments)*

# Coding Techniques and Best Practices

## Comments

lines of text that Workbench will not run as code; they convey a message to someone who reads our code

`/* ... */`      *(for Large comments)*

`# or --`      *(for one-line comments)*

Next:

Loading the ‘employees’ database

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

# The SQL SELECT Statement



**SELECT... FROM...**

**SELECT... FROM...**

**So far:**

# SELECT... FROM...

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating SQL Databases and Tables
- Creating SQL Constraints

# SELECT... FROM...

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating SQL Databases and Tables
- Creating SQL Constraints

## In this section:

# SELECT... FROM...

## So far:

- Theory of Relational Databases
- SQL Theory
- Creating SQL Databases and Tables
- Creating SQL Constraints

## In this section:

- Data Manipulation

# SELECT... FROM...

- the SELECT statement

## SELECT... FROM...

- the SELECT statement

allows you to extract a fraction of the entire data set

# SELECT... FROM...

- **the SELECT statement**

allows you to extract a fraction of the entire data set

- used to retrieve data from database objects, like tables

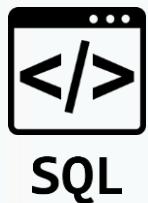
# SELECT... FROM...

- **the SELECT statement**

allows you to extract a fraction of the entire data set

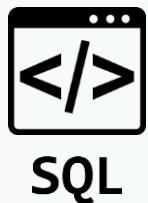
- used to retrieve data from database objects, like tables
- used to “*query data from a database*”

# SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

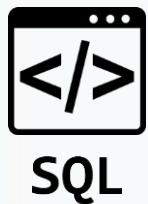
# SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

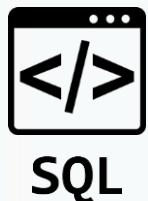
- when extracting information, SELECT goes with FROM

# SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

# SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

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

# SELECT... FROM...



SQL

```
SELECT * FROM employees;
```

\* - a wildcard character, means “all” and “everything”

A large conference room with rows of black office chairs facing a long rectangular table. The room has a modern design with a grid-patterned ceiling and floor-to-ceiling windows overlooking a landscape. The word "WHERE" is overlaid in white capital letters in the center of the image.

WHERE

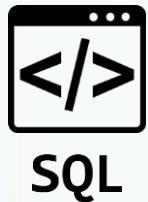
# WHERE



SQL

```
SELECT * FROM employees;
```

# WHERE



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

# WHERE

- the WHERE clause

## WHERE

- the WHERE clause

it will allow us to set a condition upon which we will specify what part of the data we want to retrieve from the database

## WHERE

- the WHERE clause

it will allow us to set a condition upon which we will specify what part of the data we want to retrieve from the database



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

## WHERE

- the WHERE clause

it will allow us to set a condition upon which we will specify what part of the data we want to retrieve from the database



```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition;
```



AND

AND

- = equal operator

**AND**

- **= equal operator**

in SQL, there are many other *Linking keywords and symbols*, called operators, that you can use with the WHERE clause

## AND

- **= equal operator**

in SQL, there are many other *Linking keywords and symbols*, called **operators**, that you can use with the WHERE clause

- AND
- OR
- IN            - NOT IN
- LIKE        - NOT LIKE
- BETWEEN... AND...

## AND

- = equal operator

in SQL, there are many other *Linking keywords and symbols*, called operators, that you can use with the WHERE clause

- AND
- OR
- IN            - NOT IN
- LIKE        - NOT LIKE
- BETWEEN... AND...
- EXISTS      - NOT EXISTS
- IS NULL     - IS NOT NULL
- comparison operators
- etc.

**AND**

**AND**

## AND

- **AND**

allows you to logically combine two statements in the condition code block

**AND**

- **AND**

allows you to logically combine two statements in the condition code block



**SQL**

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition_1 AND condition_2;
```

## AND

- **AND**

allows you to logically combine two statements in the condition code block



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition_1 AND condition_2;
```

- allows us to *narrow* the output we would like to extract from our data

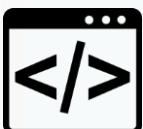


OR

OR

- AND

AND binds SQL to meet both conditions enlisted in the WHERE clause simultaneously



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition_1 AND condition_2;
```

**OR**

- **AND**

conditions set on *different columns*

**OR**

- **AND**

conditions set on *different columns*

- **OR**

conditions set on *the same column*

A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

# Operator Precedence

# Operator Precedence

So far:

# Operator Precedence

## So far:

- AND
- OR

# Operator Precedence

## So far:

- AND
- OR

## In this lesson:

# Operator Precedence

## So far:

- AND
- OR

## In this lesson:

- the *Logical order* with which you must comply when you use both operators in the same WHERE block

# Operator Precedence

- logical operator precedence

# Operator Precedence

- **logical operator precedence**

an SQL rule stating that in the execution of the query, the operator **AND** is applied first, while the operator **OR** is applied second

# Operator Precedence

- logical operator precedence

an SQL rule stating that in the execution of the query, the operator AND is applied first, while the operator OR is applied second

**AND > OR**

# Operator Precedence

- logical operator precedence

an SQL rule stating that in the execution of the query, the operator AND is applied first, while the operator OR is applied second

**AND > OR**

*regardless of the order in which you use these operators, SQL will always start by reading the conditions around the AND operator*

# Wildcard Characters

# Wildcard Characters

- wildcard characters

%

-

\*

you would need a wildcard character whenever you wished to put “anything” on its place

# Wildcard Characters

%

- a substitute for a sequence of characters

LIKE ('Mar%')

Mark, Martin, Margaret

\_

- helps you match a single character

LIKE ('Mar\_')

Mark, Mary, Marl

# Wildcard Characters

- \* will deliver a list of *all* columns in a table

```
SELECT * FROM employees;
```

- it can be used to count *all* rows of a table



**BETWEEN... AND...**

## BETWEEN... AND...

- **BETWEEN... AND...**

helps us designate the interval to which a given value belongs

# BETWEEN... AND...

**SELECT**

\*

**FROM**

employees

**WHERE**

hire\_date **BETWEEN** '1990-01-01' **AND** '2000-01-01';



# BETWEEN... AND...

**SELECT**

\*

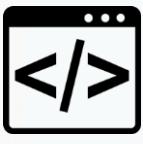
**FROM**

employees

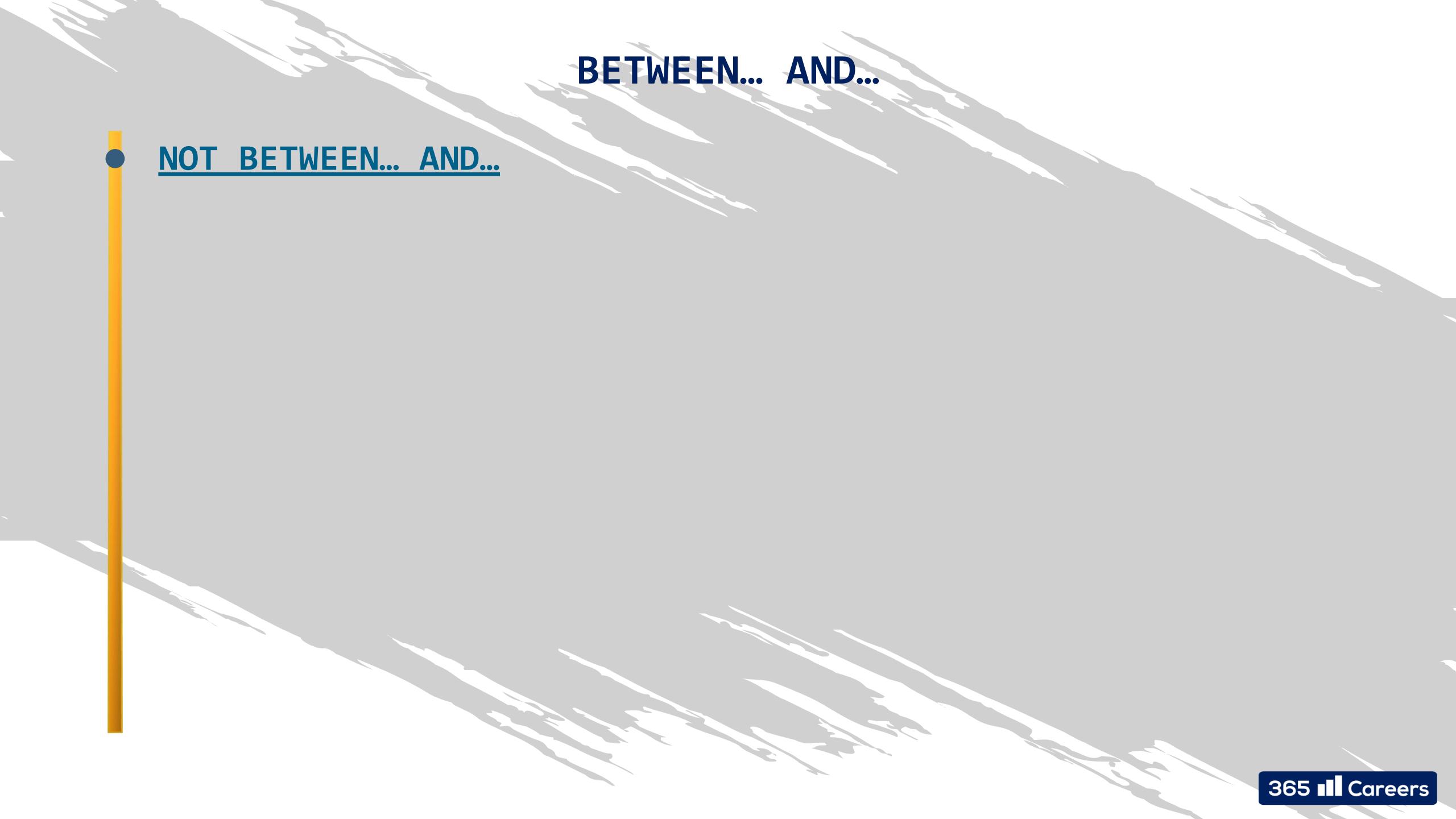
**WHERE**

hire\_date **BETWEEN** '1990-01-01' **AND** '2000-01-01';

'1990-01-01' AND '2000-01-01' *will be included in the retrieved list of records*



SQL



# BETWEEN... AND...

- **NOT BETWEEN... AND...**

## BETWEEN... AND...

- **NOT BETWEEN... AND...**

will refer to an interval composed of two parts:

## BETWEEN... AND...

- **NOT BETWEEN... AND...**

will refer to an interval composed of two parts:

- an interval below the first value indicated

## BETWEEN... AND...

- **NOT BETWEEN... AND...**

will refer to an interval composed of two parts:

- an interval below the first value indicated
- a second interval above the second value

# BETWEEN... AND...

**SELECT**

\*

**FROM**

employees

**WHERE**

hire\_date **NOT BETWEEN** '1990-01-01' **AND** '2000-01-01';



SQL

# BETWEEN... AND...

**SELECT**

\*

**FROM**

employees

**WHERE**

**hire\_date NOT BETWEEN '1990-01-01' AND '2000-01-01';**

- the **hire\_date** is *before* '1990-01-01'

# BETWEEN... AND...



```
SELECT
  *
FROM
  employees
WHERE
  hire_date NOT BETWEEN '1990-01-01' AND '2000-01-01';
```

- the `hire_date` is *before* '1990-01-01'  
or
- the `hire_date` is *after* '2000-01-01'

# BETWEEN... AND...

**SELECT**

\*

**FROM**

employees

**WHERE**

**hire\_date NOT BETWEEN '1990-01-01' AND '2000-01-01';**

'1990-01-01' AND '2000-01-01' are not included in the intervals



# BETWEEN... AND...

- **BETWEEN... AND...**

## BETWEEN... AND...

- **BETWEEN... AND...**

- not used only for date values

## BETWEEN... AND...

- **BETWEEN... AND...**

- not used only for date values
- could also be applied to strings and numbers

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of dark-colored office chairs arranged facing a large, dark rectangular conference table in the center. The room has a high ceiling with a grid of recessed lighting fixtures. Large windows along the back wall provide a view of an outdoor area with some greenery and a clear sky.

**IS NOT NULL / IS NULL**

# IS NOT NULL / IS NULL

- **IS NOT NULL**

## IS NOT NULL / IS NULL

- **IS NOT NULL**

used to extract values that are not null

# IS NOT NULL / IS NULL

- **IS NOT NULL**

used to extract values that are not null



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE column_name IS NOT NULL;
```

## IS NOT NULL / IS NULL

- **IS NULL**

used to extract values that are null



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE column_name IS NULL;
```

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of black office chairs facing towards the front. The front wall is a light-colored paneling. Above the paneling, there are several flat-screen monitors mounted on the wall, displaying various data or software interfaces. The ceiling is white with a grid of recessed lighting fixtures. The floor is a polished wood. The overall atmosphere is professional and tech-oriented.

# Other Comparison Operators

# Other Comparison Operators

So far:

# Other Comparison Operators

## So far:

- BETWEEN... AND...
- LIKE
- IS NOT NULL
- NOT LIKE
- IS NULL

# Other Comparison Operators

## So far:

- BETWEEN... AND...
- LIKE
- IS NOT NULL

## In this lecture:

- NOT LIKE
- IS NULL

# Other Comparison Operators

## So far:

- BETWEEN... AND...
  - LIKE
  - IS NOT NULL
- NOT LIKE
  - IS NULL

## In this lecture:

=, >, >=, =<, <, <>, !=

# Other Comparison Operators

SQL

=

equal to

>

greater than

>=

greater than or equal to

<

less than

<=

less than or equal to

# Other Comparison Operators

SQL

<>, !=

“Not Equal” operators

not equal, ≠

different from



**SELECT DISTINCT**

# SELECT DISTINCT

- the SELECT statement

## SELECT DISTINCT

- the SELECT statement  
can retrieve rows from a designated column, given some criteria

# SELECT DISTINCT

- **SELECT DISTINCT**

*selects all *distinct*, *different* data values*

# SELECT DISTINCT

- **SELECT DISTINCT**

selects all *distinct, different* data values



SQL

```
SELECT DISTINCT column_1, column_2,... column_n  
FROM table_name;
```

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are rows of black office chairs facing towards the front. The front wall features several large, dark rectangular panels, likely projection screens or television monitors. The room has a high ceiling with a grid of recessed lighting fixtures. The floor is a light-colored wood or laminate. The overall atmosphere is professional and spacious.

# Introduction to Aggregate Functions

# Introduction to Aggregate Functions

## aggregate functions

they are applied on *multiple rows of a single column* of a table and  
*return an output of a single value*

# Introduction to Aggregate Functions

- COUNT()

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

returns the maximum value from the entire list

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

returns the maximum value from the entire list

- **AVG()**

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

returns the maximum value from the entire list

- **AVG()**

calculates the average of all non-null values belonging to a certain column of a table

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

# Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- it is frequently used in combination with the reserved word **“DISTINCT”**

# Introduction to Aggregate Functions

- COUNT()



```
SELECT COUNT(column_name)  
FROM table_name;
```

# Introduction to Aggregate Functions

- COUNT()

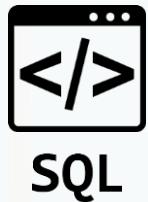


```
SELECT COUNT(column_name)  
FROM table_name;
```

the parentheses after COUNT() must start right after the keyword, not after a whitespace

# Introduction to Aggregate Functions

- **COUNT(DISTINCT )**



```
SELECT COUNT(DISTINCT column_name)  
FROM table_name;
```

# Introduction to Aggregate Functions

## aggregate functions

they are applied on *multiple rows of a single column* of a table and  
*return an output of a single value*

- they ignore NULL values unless told not to



**GROUP BY**

# GROUP BY

When working in SQL, results can be grouped according to a specific field or fields

# GROUP BY

- **GROUP BY**

When working in SQL, results can be grouped according to a specific field or fields

# GROUP BY

- **GROUP BY**

When working in SQL, results can be grouped according to a specific field or fields

- **GROUP BY** must be placed immediately after the **WHERE** conditions, if any, and just before the **ORDER BY** clause

# GROUP BY

- **GROUP BY**

When working in SQL, results can be grouped according to a specific field or fields

- **GROUP BY** must be placed immediately after the **WHERE** conditions, if any, and just before the **ORDER BY** clause
- **GROUP BY** is one of the most powerful and useful tools in SQL

# GROUP BY

- **GROUP BY**



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
ORDER BY column_name(s);
```

# GROUP BY

- **GROUP BY**

*in most cases, when you need an aggregate function, you must add a GROUP BY clause in your query, too*

*Always include the field you have grouped your results by in the SELECT statement!*



HAVING



# HAVING

- **HAVING**

# HAVING

- **HAVING**

- frequently implemented with **GROUP BY**

# HAVING

- **HAVING**

refines the output from records that do not satisfy a certain condition

- frequently implemented with **GROUP BY**

# HAVING



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s);
```

# HAVING



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s);
```

- HAVING is like WHERE but applied to the GROUP BY block

# HAVING

- **WHERE vs. HAVING**

after HAVING, you can have a condition with an aggregate function,  
while WHERE cannot use aggregate functions within its conditions



**WHERE vs HAVING**

# WHERE vs HAVING

• **WHERE**

## WHERE vs HAVING

- **WHERE**

allows us to set conditions that refer to subsets of *individual* rows

## WHERE vs HAVING

- **WHERE**

allows us to set conditions that refer to subsets of *individual* rows



applied *before* re-organizing the output into groups

# WHERE vs HAVING

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

# WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

re-organizing the  
output into groups  
**(GROUP BY)**

# WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

re-organizing the  
output into groups  
**(GROUP BY)**



the output can be further improved, or *filtered*

# WHERE vs HAVING

WHERE



HAVING

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

re-organizing the  
output into groups  
**(GROUP BY)**

# WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

HAVING



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
10	8/11/2017	2	B_1

re-organizing the  
output into groups  
**(GROUP BY)**

# WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

HAVING



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

ORDER BY...

1	9/3/2016	1	A_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
10	8/11/2017	2	B_1

re-organizing the  
output into groups  
**(GROUP BY)**

# WHERE vs HAVING

- **HAVING**

- you *cannot* have both an aggregated and a non-aggregated condition in the HAVING clause

# WHERE vs HAVING

*Aggregate functions - GROUP BY and HAVING*

# WHERE vs HAVING

*Aggregate functions - GROUP BY and HAVING*

*General conditions - WHERE*

# WHERE vs HAVING



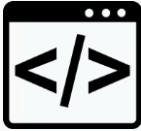
SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s);
```

A large conference room with rows of chairs facing a central table with laptops. The room has a modern design with glass walls and a wooden floor. The word "LIMIT" is overlaid in white text.

LIMIT

# LIMIT



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s)
LIMIT number ;
```

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

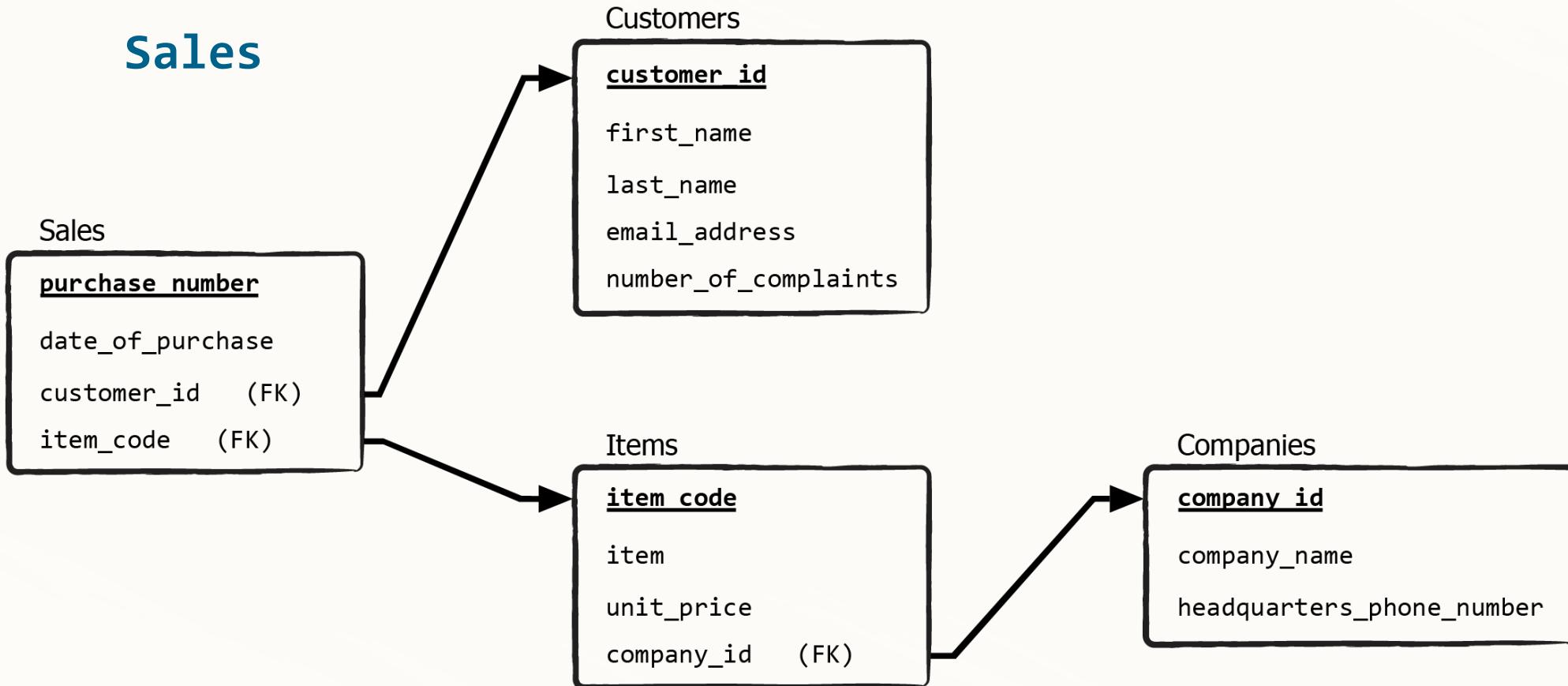
# The SQL INSERT Statement



# The INSERT Statement

# The INSERT Statement

## Sales



# The INSERT Statement

- The INSERT Statement



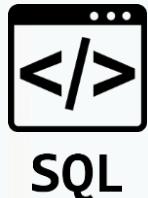
```
INSERT INTO table_name (column_1, column_2, ..., column_n)  
VALUES (value_1, value_2, ..., value_n);
```

A large conference room with rows of chairs facing a large screen. The room has a modern design with a grid pattern ceiling and large windows overlooking a landscape.

# Inserting Data INTO a New Table

# Inserting Data INTO a New Table

- INSERT INTO SELECT



```
INSERT INTO table_2 (column_1, column_2, ..., column_n)
SELECT column_1, column_2, ..., column_n
FROM table_1
WHERE condition;
```

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

# The SQL UPDATE Statement

A large conference room with rows of chairs facing a large screen. The room has a modern design with a grid pattern on the ceiling and large windows in the background.

# TCL's COMMIT and ROLLBACK

# Transaction Control Language

- **the COMMIT statement**

- saves the transaction in the database
- changes cannot be undone

- **the ROLLBACK clause**

- allows you to take a step back
- the last change(s) made will not count
- reverts to the last non-committed state

# TCL's COMMIT and ROLLBACK

- **the COMMIT statement**

- saves the transaction in the database
- changes cannot be undone

# TCL's COMMIT and ROLLBACK

- **the COMMIT statement**

- saves the transaction in the database
- changes cannot be undone

*used to save the state of the data in the database at the moment of its execution*

# TCL's COMMIT and ROLLBACK

- **the COMMIT statement**

- saves the transaction in the database
- changes cannot be undone

*used to save the state of the data in the database at the moment of its execution*

- **the ROLLBACK clause**

- allows you to take a step back
- the last change(s) made will not count
- reverts to the last non-committed state

# TCL's COMMIT and ROLLBACK

- **the COMMIT statement**

- saves the transaction in the database
- changes cannot be undone

*used to save the state of the data in the database at the moment of its execution*

- **the ROLLBACK clause**

- allows you to take a step back
- the last change(s) made will not count
- reverts to the last non-committed state

*it will refer to the state corresponding to the Last time you executed COMMIT*

# TCL's COMMIT and ROLLBACK

# TCL's COMMIT and ROLLBACK

**COMMIT;**

1

# TCL's COMMIT and ROLLBACK

COMMIT; COMMIT;

1      2

# TCL's COMMIT and ROLLBACK

**COMMIT; COMMIT;**

1      2      ...

# TCL's COMMIT and ROLLBACK

COMMIT; COMMIT; COMMIT;



# TCL's COMMIT and ROLLBACK



# TCL's COMMIT and ROLLBACK

- ROLLBACK will have an effect *on the Last execution you have performed*



# TCL's COMMIT and ROLLBACK

- ROLLBACK will have an effect *on the Last execution you have performed*



# TCL's COMMIT and ROLLBACK

- ROLLBACK will have an effect *on the Last execution you have performed*



# TCL's COMMIT and ROLLBACK

- ROLLBACK will have an effect *on the Last execution you have performed*



# TCL's COMMIT and ROLLBACK

- ROLLBACK will have an effect *on the Last execution you have performed*
- you cannot restore data to a state corresponding to an earlier COMMIT



A large, modern conference room with rows of chairs facing a central table. The room has large windows overlooking a city skyline.

# The UPDATE Statement

# The UPDATE Statement

- the UPDATE Statement

# The UPDATE Statement

- **the UPDATE Statement**

used to update the values of existing records in a table

# The UPDATE Statement

- the UPDATE Statement

used to update the values of existing records in a table



SQL

```
UPDATE table_name  
SET column_1 = value_1, column_2 = value_2 ...  
WHERE conditions;
```

# The UPDATE Statement

- the UPDATE Statement

used to update the values of existing records in a table



SQL

```
UPDATE table_name  
SET column_1 = value_1, column_2 = value_2 ...  
WHERE conditions;
```

- we do not have to update each value of the record of interest

# The UPDATE Statement

- the UPDATE Statement

used to update the values of existing records in a table



SQL

```
UPDATE table_name  
SET column_1 = value_1, column_2 = value_2 ...  
WHERE conditions;
```

- we do not have to update each value of the record of interest
- we can still say we have updated the specific record

# The UPDATE Statement

- the UPDATE Statement

used to update the values of existing records in a table



SQL

```
UPDATE table_name  
SET column_1 = value_1, column_2 = value_2 ...  
WHERE conditions;
```

# The UPDATE Statement

- the UPDATE Statement

used to update the values of existing records in a table



SQL

```
UPDATE table_name  
SET column_1 = value_1, column_2 = value_2 ...  
WHERE conditions;
```

- if you don't provide a WHERE *condition*, all rows of the table will be updated

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

# The SQL DELETE Statement



# The DELETE Statement

# The DELETE Statement

- the DELETE statement

removes records from a database



SQL

```
DELETE FROM table_name  
WHERE conditions;
```

# FOREIGN KEY Constraint

- **ON DELETE CASCADE**

*if a specific value from the parent table's primary key has been deleted, all the records from the child table referring to this value will be removed as well*



# **DROP vs TRUNCATE vs DELETE**

# DROP vs TRUNCATE vs DELETE

column_1
1
2
3
4
...
10

# DROP vs TRUNCATE vs DELETE

## DROP

column_1
1
2
3
4
...
10

# DROP vs TRUNCATE vs DELETE

## DROP

1  
2  
3  
4  
...  
10



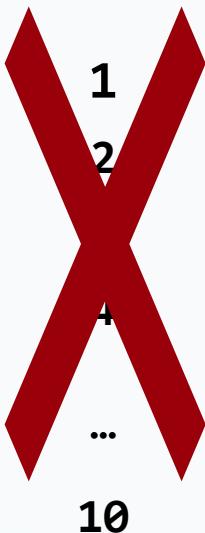
column_1



indexes  
constraints  
...  
...

# DROP vs TRUNCATE vs DELETE

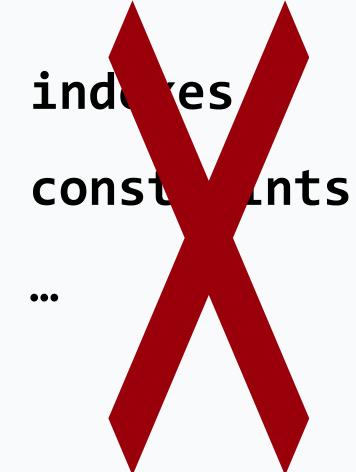
## DROP



+

column_1

+



# DROP vs TRUNCATE vs DELETE

- **DROP**

- you won't be able to roll back to its initial state, or to the last COMMIT statement

*use DROP TABLE only when you are sure you aren't going to use the table in question anymore*

# DROP vs TRUNCATE vs DELETE

## • TRUNCATE

column_1
1
2
3
4
...
10

# DROP vs TRUNCATE vs DELETE

- TRUNCATE ~ DELETE without WHERE

column_1
1
2
3
4
...
10

# DROP vs TRUNCATE vs DELETE

- TRUNCATE ~ DELETE without WHERE

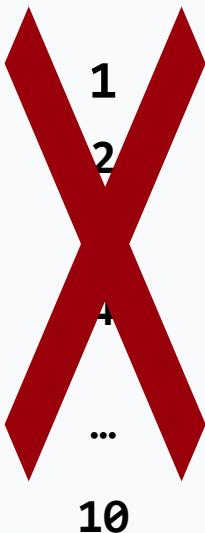
1  
2  
3  
4  
...  
10



column_1

# DROP vs TRUNCATE vs DELETE

- TRUNCATE ~ DELETE without WHERE



+

column_1

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

column_1

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

column_1
11

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

column_1
X

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

column_1
X 1

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

column_1
11
12

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

column_1
11
12

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE**

when truncating, auto-increment values will be reset

column_1
1
2
3
4
...
10

TRUNCATE

column_1
1
2
3
4
...
10

# DROP vs TRUNCATE vs DELETE

- **DELETE**

# DROP vs TRUNCATE vs DELETE

- **DELETE**

*removes records row by row*

# DROP vs TRUNCATE vs DELETE

- **DELETE**

removes records *row by row*



SQL

```
DELETE FROM table_name  
WHERE conditions;
```

# DROP vs TRUNCATE vs DELETE

- **DELETE**

removes records *row by row*



`DELETE FROM table_name  
WHERE conditions;`

SQL

- **TRUNCATE ~ DELETE without WHERE**

# DROP vs TRUNCATE vs DELETE

- TRUNCATE vs DELETE without WHERE

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE vs DELETE without WHERE**

- the SQL optimizer will implement different programmatic approaches when we are using TRUNCATE or DELETE

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE vs DELETE without WHERE**

- the SQL optimizer will implement different programmatic approaches when we are using TRUNCATE or DELETE



# DROP vs TRUNCATE vs DELETE

- **TRUNCATE vs DELETE without WHERE**

- the SQL optimizer will implement different programmatic approaches when we are using TRUNCATE or DELETE



TRUNCATE delivers the output much *quicker* than DELETE

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE vs DELETE without WHERE**

- the SQL optimizer will implement different programmatic approaches when we are using TRUNCATE or DELETE



TRUNCATE delivers the output much *quicker* than DELETE

*row by row*

*row by row*

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE vs DELETE without WHERE**

- the SQL optimizer will implement different programmatic approaches when we are using TRUNCATE or DELETE



- TRUNCATE delivers the output much *quicker* than DELETE**

~~row by row~~

*row by row*

# DROP vs TRUNCATE vs DELETE

- **TRUNCATE vs DELETE without WHERE**

- auto-increment values are *not* reset with DELETE

# DROP vs TRUNCATE vs DELETE

## TRUNCATE vs DELETE without WHERE

- auto-increment values are *not* reset with DELETE

column_1
1
2
3
4
...
10

# DROP vs TRUNCATE vs DELETE

## TRUNCATE vs DELETE without WHERE

- auto-increment values are *not* reset with DELETE

column_1
1
2
3
4
...
10

**DELETE**



# DROP vs TRUNCATE vs DELETE

## TRUNCATE vs DELETE without WHERE

- auto-increment values are *not* reset with DELETE

column_1
1
2
3
4
...
10

**DELETE**

column_1

# DROP vs TRUNCATE vs DELETE

## TRUNCATE vs DELETE without WHERE

- auto-increment values are *not* reset with DELETE

column_1
1
2
3
4
...
10

**DELETE**



column_1
11
12
13
14
...
20

**Next:**

Next:

SQL Functions

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

# MySQL Aggregate Functions

A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

**COUNT()**

# COUNT()

So far:

# COUNT()

## So far:

- Theory of Relational Databases
- SQL Theory
- Coding Techniques and Best Practices
- SELECT, INSERT, UPDATE, DELETE

# COUNT()

## So far:

- Theory of Relational Databases
- SQL Theory
- Coding Techniques and Best Practices
- SELECT, INSERT, UPDATE, DELETE

## Next:

# COUNT()

## So far:

- Theory of Relational Databases
- SQL Theory
- Coding Techniques and Best Practices
- SELECT, INSERT, UPDATE, DELETE

## Next:

- Aggregate Functions

# COUNT()

## aggregate functions

they gather data from *many* rows of a table, then aggregate it into a *single* value

# COUNT()

## aggregate functions

they gather data from *many* rows of a table, then aggregate it into a *single* value

## INPUT

# COUNT()

## aggregate functions

they gather data from *many* rows of a table, then aggregate it into a *single* value

## INPUT

the information contained  
in *multiple* rows

# COUNT()

## aggregate functions

they gather data from *many* rows of a table, then aggregate it into a *single* value

### INPUT

the information contained →  
in *multiple* rows

# COUNT()

## aggregate functions

they gather data from *many* rows of a table, then aggregate it into a *single* value

### INPUT

the information contained →  
in *multiple* rows

### OUTPUT

# COUNT()

## aggregate functions

they gather data from *many* rows of a table, then aggregate it into a *single* value

### INPUT

the information contained  
in *multiple* rows



### OUTPUT

the *single* value they  
provide

# COUNT()

- COUNT()

# COUNT()

COUNT()

SUM()

# COUNT()

COUNT()

SUM()

MIN()

# COUNT()

COUNT()

SUM()

MIN()

MAX()

# COUNT()

COUNT()

SUM()

MIN()

MAX()

AVG()

# COUNT()

COUNT()

SUM()

MIN()

MAX()

AVG()



# COUNT()

COUNT()

SUM()

MIN()

MAX()

AVG()

aggregate functions

# COUNT()

COUNT()

SUM()

MIN()

MAX()

AVG()

aggregate functions

=

summarizing functions

# COUNT()

- Why do these functions exist?

# COUNT()

- Why do these functions exist?

- they are a response to the information requirements of a company's different organizational levels

# COUNT()

- Why do these functions exist?

- they are a response to the information requirements of a company's different organizational levels
- top management executives are typically interested in *summarized figures* and rarely in detailed data

# COUNT()

- **COUNT()**

applicable to both *numeric* and *non-numeric* data

## COUNT()

- **COUNT(DISTINCT )**

helps us find the number of times unique values are encountered in a given column

## COUNT()

- aggregate functions *typically* ignore null values throughout the field to which they are applied

## COUNT()

- aggregate functions typically ignore null values throughout the field to which they are applied

## COUNT()

- aggregate functions typically ignore null values throughout the field to which they are applied



## COUNT()

- aggregate functions typically ignore null values throughout the field to which they are applied



only if you have indicated a *specific column name* within the parentheses

## COUNT()

- aggregate functions typically ignore null values throughout the field to which they are applied



only if you have indicated a *specific column name* within the parentheses

Alternatively:

## COUNT()

- aggregate functions typically ignore null values throughout the field to which they are applied



only if you have indicated a *specific column name* within the parentheses

Alternatively:

- COUNT(\*)

## COUNT()

- aggregate functions typically ignore null values throughout the field to which they are applied



only if you have indicated a *specific column name* within the parentheses

### Alternatively:

COUNT(\*)

\* returns the number of all rows of the table, **NULL** values included

# COUNT()

- COUNT()

# COUNT()

- **COUNT()**

the parentheses and the argument must be attached to the name of the aggregate function

# COUNT()

- **COUNT()**

the parentheses and the argument must be attached to the name of the aggregate function

- you shouldn't leave white space before opening the parentheses

# COUNT()

- **COUNT()**

the parentheses and the argument must be attached to the name of the aggregate function

- you shouldn't leave white space before opening the parentheses

- **COUNT()**

# COUNT()

- **COUNT()**

the parentheses and the argument must be attached to the name of the aggregate function

- you shouldn't leave white space before opening the parentheses

COUNT()

COUNT ()

# COUNT()

- **COUNT()**

the parentheses and the argument must be attached to the name of the aggregate function

- you shouldn't leave white space before opening the parentheses

COUNT()

COUNT\_()

# COUNT()

- COUNT()

the parentheses and the argument must be attached to the name of the aggregate function

- you shouldn't leave white space before opening the parentheses

COUNT()

~~COUNT\_()~~

A large conference room with rows of chairs facing a central table with laptops. The room has a modern design with large windows overlooking a landscape. The text "SUM()" is overlaid on the image.

SUM()

**SUM()**

**COUNT(\*)**

# SUM()

- COUNT(\*)

\* returns all rows of the table, **NULL values included**

**SUM()**

**COUNT(\*)**

\* returns all rows of the table, **NULL** values included

**SUM(\*)**

## SUM()

- COUNT(\*)

\* returns all rows of the table, **NULL values included**

- ~~SUM(\*)~~

# SUM()

- **COUNT(\*)**

\* returns all rows of the table, **NULL values included**

- ~~SUM(\*)~~

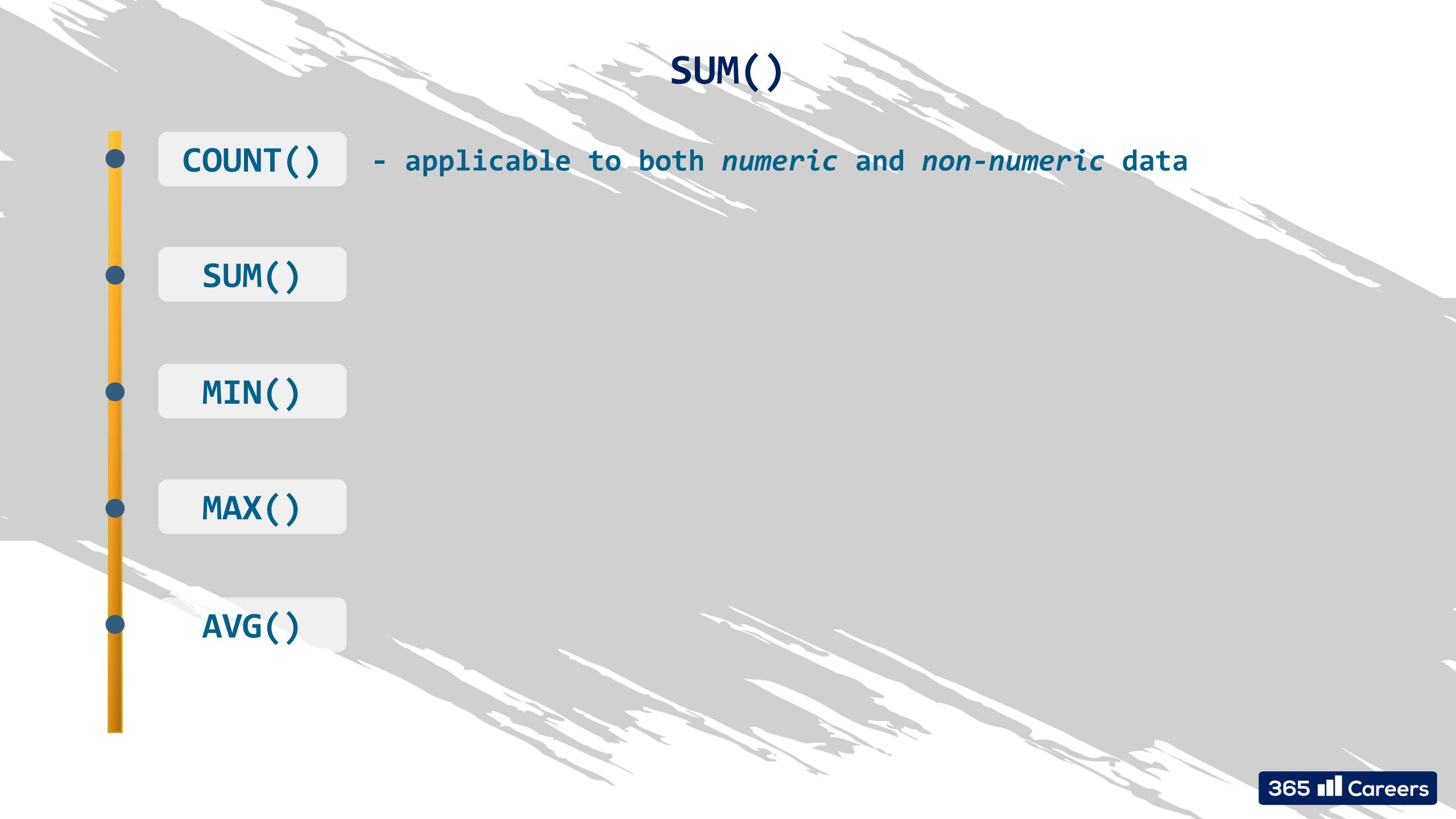
\* goes well with only the **COUNT()** function

**SUM()**

**COUNT()**

## SUM()

- COUNT() - applicable to both *numeric* and *non-numeric* data



# SUM()

COUNT()

- applicable to both *numeric* and *non-numeric* data

SUM()

MIN()

MAX()

AVG()

# SUM()

- COUNT()

- applicable to both *numeric* and *non-numeric* data

- SUM()

- MIN()

- MAX()

- AVG()



## SUM()

COUNT()

- applicable to both *numeric and non-numeric* data

SUM()

MIN()

MAX()

AVG()

- work *only with numeric* data

A large, modern conference room with a long rectangular table in the center. Rows of black office chairs are arranged facing the table. The room has floor-to-ceiling windows on one side, providing a view of an outdoor area. The ceiling is white with a grid of recessed lights. The overall atmosphere is professional and spacious.

# MIN() and MAX()

# MIN() and MAX()

- **MAX()**

returns the maximum value of a column

# MIN() and MAX()

- **MAX()**

returns the maximum value of a column

- **MIN()**

returns the minimum value of a column

A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

AVG()

# AVG()

- **AVG()**

extracts the average value of all non-null values in a field

**AVG()**

**COUNT()**

**SUM()**

**MIN()**

**MAX()**

**AVG()**

# AVG()

COUNT()

SUM()

MIN()

MAX()

AVG()

- aggregate functions can be applied to any *group* of data values within a certain column

# AVG()

COUNT()

SUM()

MIN()

MAX()

AVG()

- aggregate functions can be applied to any group of data values within a certain column

# AVG()

COUNT()

SUM()

MIN()

MAX()

AVG()

- aggregate functions can be applied to any group of data values within a certain column



## AVG()

COUNT()

SUM()

MIN()

MAX()

AVG()

- aggregate functions can be applied to any group of data values within a certain column

frequently used together with a GROUP BY clause

A large conference room with rows of black office chairs facing a long rectangular table. The room has a modern design with a grid-patterned ceiling and floor-to-ceiling windows overlooking a landscape. The word "ROUND()" is overlaid in white text in the center of the image.

# ROUND()

# ROUND()

- **ROUND(*#,decimal\_places*)**

# ROUND()

- **ROUND(*#,decimal\_places*)**

numeric, or math, function you can use

# ROUND()

- **ROUND(*#,decimal\_places*)**

numeric, or math, function you can use

- usually applied to the single values that aggregate functions return

A large, modern conference room with rows of chairs facing a large screen. The room has floor-to-ceiling windows overlooking a city skyline.

# COALESCE() - Preamble

## COALESCE() - Preamble

Here we will study something a bit more sophisticated.

IF NULL() and COALESCE() are among the advanced SQL functions in the toolkit of SQL professionals. They are used when null values are dispersed in your data table and you would like to substitute the null values with another value.

So, let's adjust the “Departments” duplicate in a way that suits the purposes of the next video, in which we will work with IF NULL() and COALESCE().

First, let's look at our table and see what we have there.

# COALESCE() - Preamble



SQL

```
SELECT * FROM departments_dup;
```

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

	dept_no	dept_name
	d001	Marketing
	d002	Finance
	d003	Human Resources
	d004	Production
	d005	Development
	d006	Quality Management
	d007	Sales
	d008	Research
	d009	Customer Service

Nine departments, with their department numbers and names provided. Ok!

# COALESCE() - Preamble

Currently, as shown in the DDL statement of this table, the “Department name” field is with a NOT NULL constraint, which naturally means we must insert a value in each of its rows.

DDL for employees.departments\_dup

```
1 ┌─ CREATE TABLE `departments_dup` (
2   └─ `dept_no` char(4) NOT NULL,
3   └─ `dept_name` varchar(40) NOT NULL
4 ) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

## COALESCE() - Preamble

Now, with the ALTER TABLE statement and the CHANGE COLUMN command, we will modify this constraint and allow null values to be registered in the “department name” column.



```
ALTER TABLE departments_dup  
CHANGE COLUMN dept_name dept_name VARCHAR(40) NULL;
```

## COALESCE() - Preamble

Right after that, we will insert into the department number column of this table a couple of data values – D-10 and D-11, the numbers of the next two potential departments in the “Departments Duplicate” table.

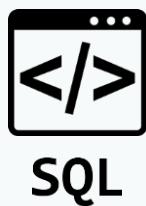


SQL

```
INSERT INTO departments_dup(dept_no) VALUES ('d010'), ('d011');
```

# COALESCE() - Preamble

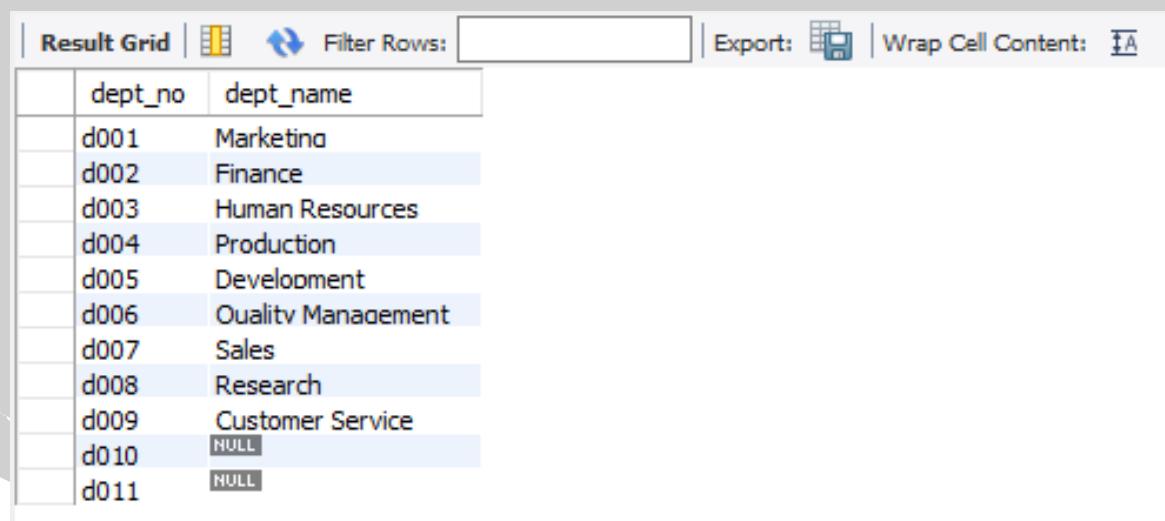
By running this SELECT query over here, you can see whether this operation was carried out successfully.



```
SELECT
    *
FROM
    departments_dup
ORDER BY dept_no ASC;
```

# COALESCE() - Preamble

We have the two new department numbers listed below, and in the “Department name” column we can see two null values. The latter happened because we allowed for null values to exist in this field, “Department name”. Thus, Workbench will indicate that a value in a cell is missing by attaching a “null” label to it. Great!



	dept_no	dept_name
d001		Marketing
d002		Finance
d003		Human Resources
d004		Production
d005		Development
d006		Quality Management
d007		Sales
d008		Research
d009		Customer Service
d010		NULL
d011		NULL

## COALESCE() - Preamble

The next adjustment we'll have to make is adding a third column called "Department manager". It will indicate the manager of the respective department. For now, we will leave it empty, and will add the NULL constraint. Finally, we will place it next to the "Department name" column by typing "AFTER "Department name".



SQL

```
ALTER TABLE employees.departments_dup  
ADD COLUMN dept_manager VARCHAR(255) NULL AFTER dept_name;
```

# COALESCE() - Preamble

Let's check the state of the "Departments duplicate" table now.



```
SELECT
  *
FROM
  departments_dup
ORDER BY dept_no ASC;
```

# COALESCE() - Preamble

Exactly as we wanted, right? The third column is completely empty and we have null values in the last two records. These are the “department name” and “manager” fields.

Result Grid | Filter Rows: [ ] | Export: | Wrap Cell Content: [A]

	dept_no	dept_name	dept_manager
	d001	Marketing	NULL
	d002	Finance	NULL
	d003	Human Resources	NULL
	d004	Production	NULL
	d005	Development	NULL
	d006	Quality Management	NULL
	d007	Sales	NULL
	d008	Research	NULL
	d009	Customer Service	NULL
	d010	NULL	NULL
	d011	NULL	NULL

# COALESCE() - Preamble

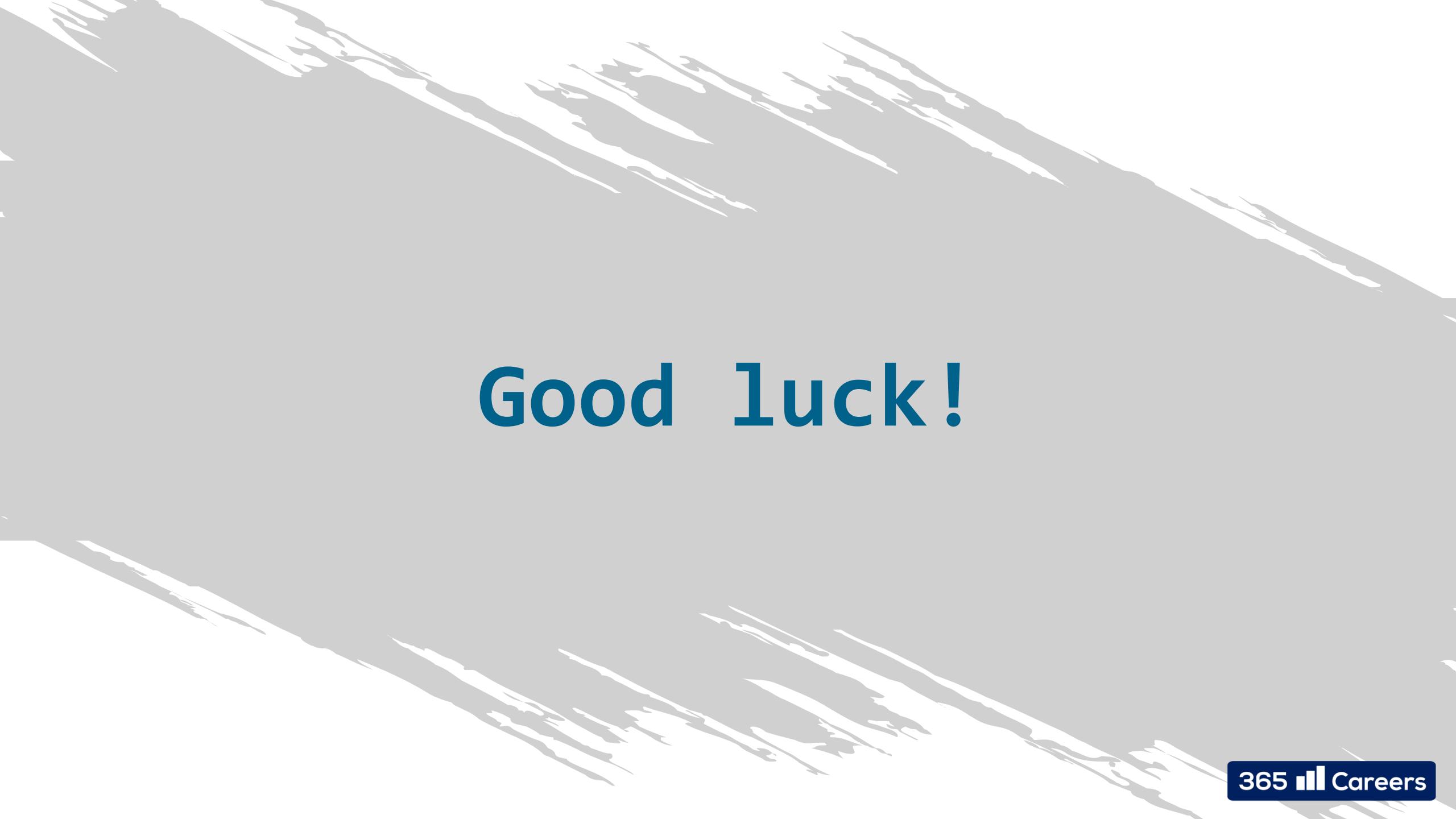
To save the “Departments duplicate” table in its current state, execute a COMMIT statement.



COMMIT;

SQL

Here we'll end the setup for the video about IF NULL() and COALESCE().



# Good luck!

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of dark-colored office chairs arranged facing a large, dark rectangular conference table in the center. The room has a high ceiling with a grid of recessed lighting fixtures. Large windows along the back wall provide a view of an outdoor area with some greenery and a clear sky.

# IFNULL() and COALESCE()

# IFNULL() and COALESCE()

- **IFNULL(expression\_1, expression\_2)**

## IFNULL() and COALESCE()

- **IFNULL(expression\_1, expression\_2)**

returns the first of the two indicated values if the data value found in the table is *not null*, and returns the second value if there is a *null* value

## IFNULL() and COALESCE()

- **IFNULL(expression\_1, expression\_2)**

returns the first of the two indicated values if the data value found in the table is *not null*, and returns the second value if there is a *null* value

- prints the returned value in the column of the output

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**  
allows you to insert N arguments in the parentheses

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**  
allows you to insert N arguments in the parentheses
  - think of COALESCE() as IFNULL() with more than two parameters

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**

allows you to insert N arguments in the parentheses

- think of COALESCE() as IFNULL() with more than two parameters
- COALESCE() will always return a *single* value of the ones we have within parentheses, and this value will be *the first non-null value* of this list, reading the values from left to right

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**
  - if COALESCE() has two arguments, it will work precisely like IFNULL()

## IFNULL() and COALESCE()

- **IFNULL()** and **COALESCE()** do not make any changes to the data set. They merely create an output where certain data values appear in place of NULL values.

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**

- we can have a single argument in a given function

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**
  - we can have a single argument in a given function
  - practitioners find this trick useful if some *hypothetical result* must be provided in a supplementary column

## IFNULL() and COALESCE()

- **COALESCE(expression\_1, expression\_2 ..., expression\_N)**

- we can have a single argument in a given function
- practitioners find this trick useful if some *hypothetical result* must be provided in a supplementary column
- COALESCE() can help you visualize a *prototype of the table's final version*

# **IFNULL() and COALESCE()**

**IFNULL()**

works with precisely *two* arguments

# **IFNULL() and COALESCE()**

**IFNULL()**

works with precisely *two* arguments

**COALESCE()**

can have *one, two, or more* arguments

**Next:**

Next:  
**Joins**

# SQL Joins

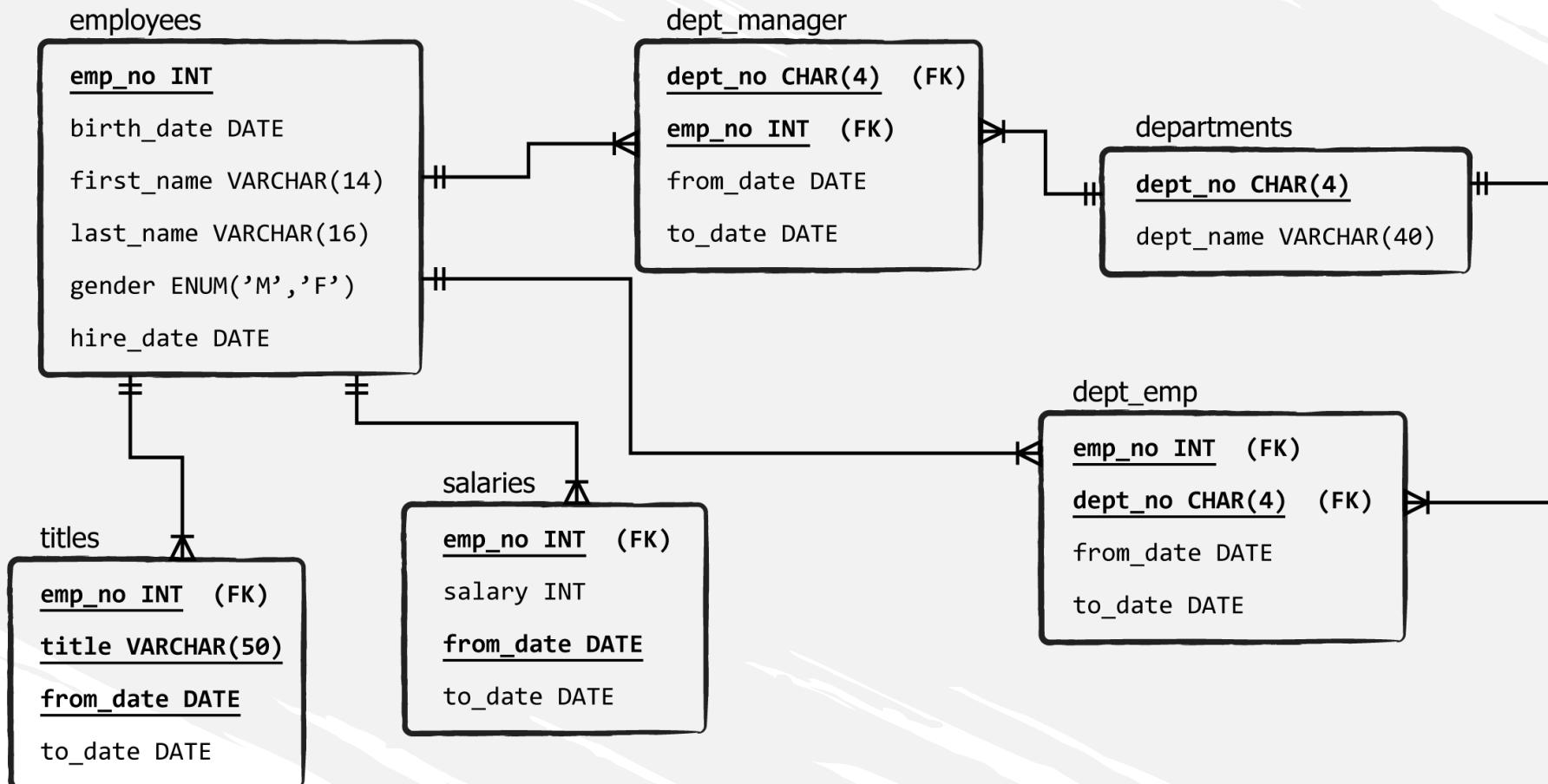
A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

# Introduction to Joins

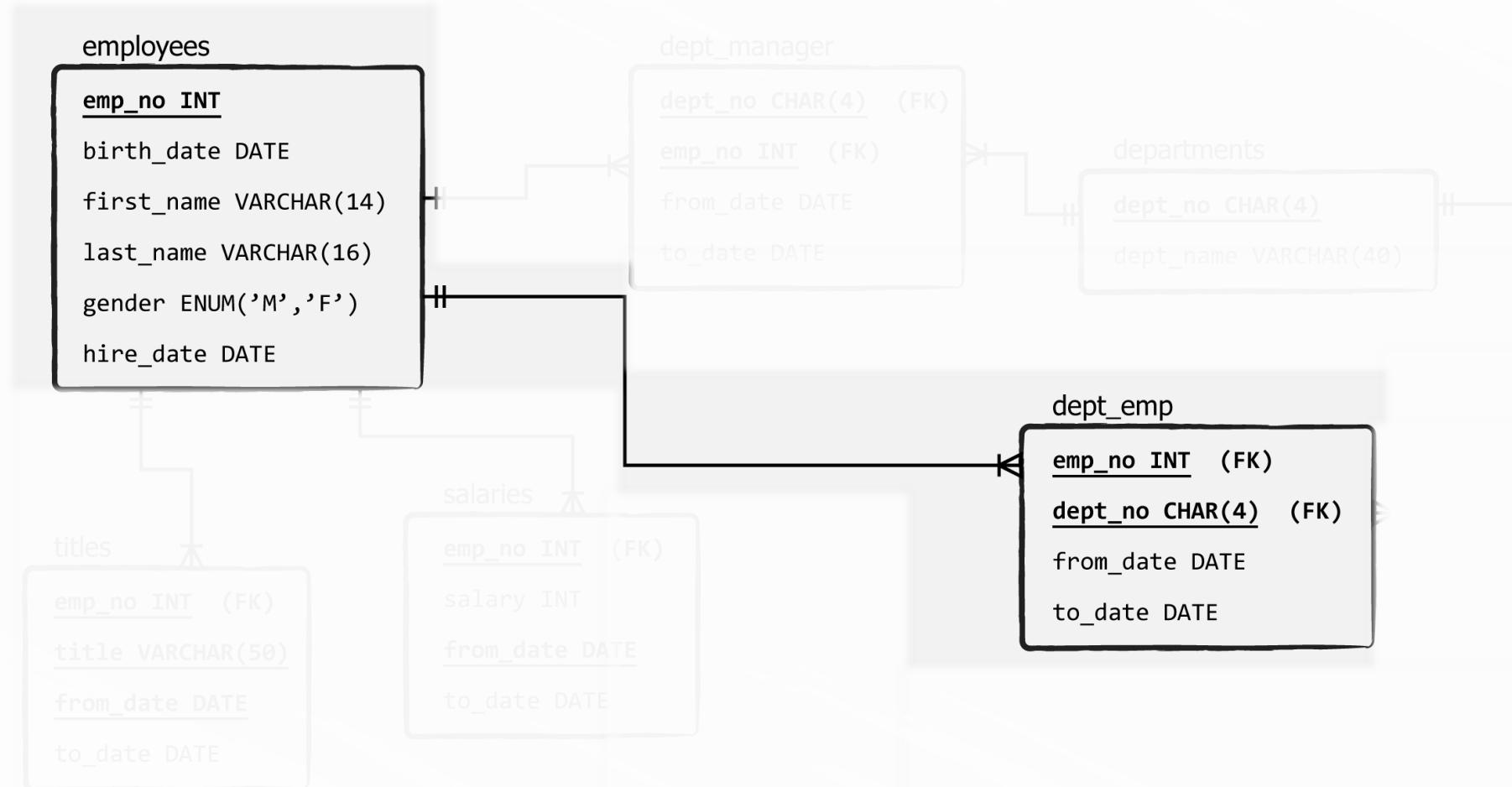
# Introduction to Joins

- **joins**  
the SQL tool that allow us to construct a relationship between objects

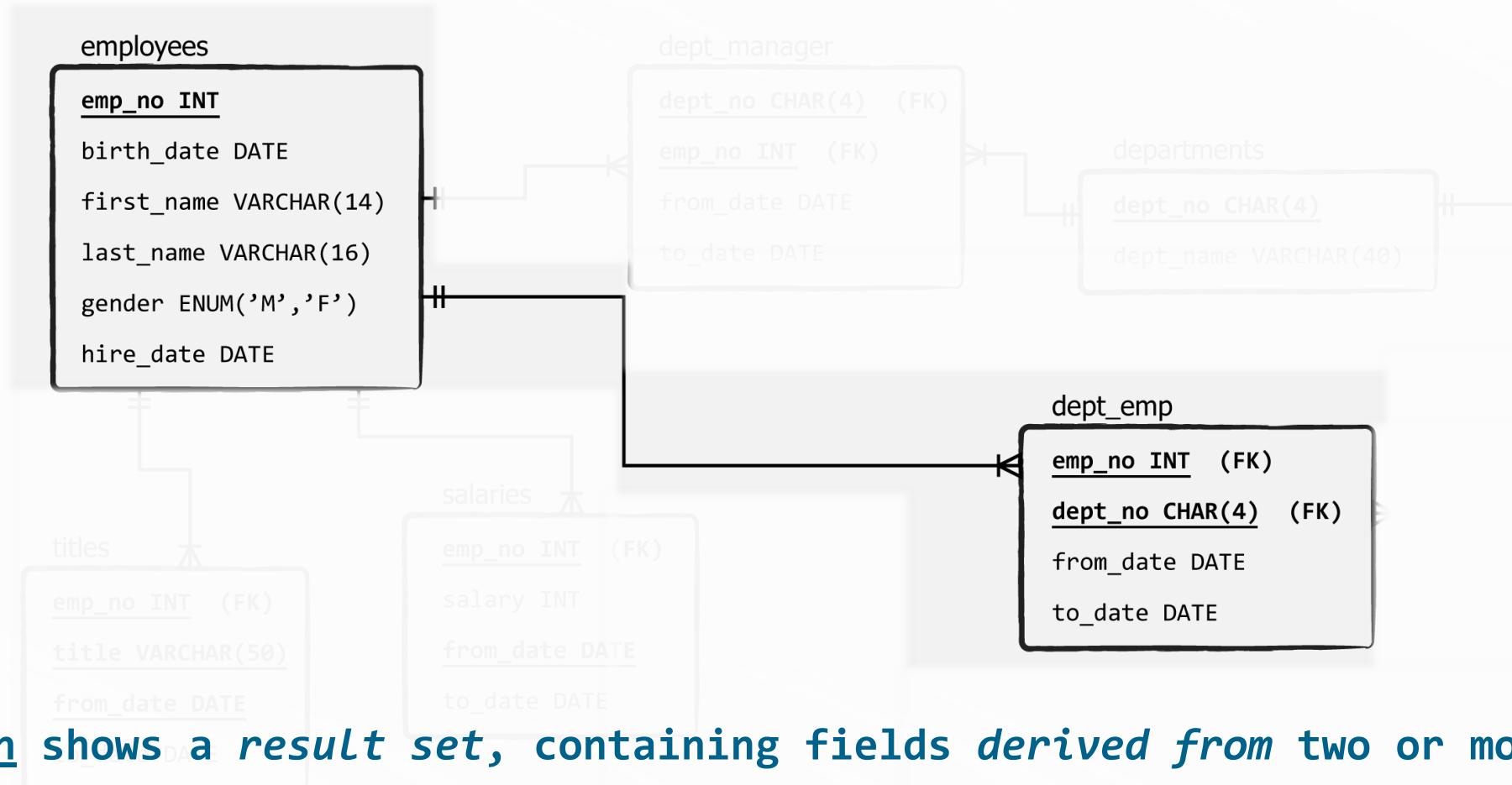
# Introduction to Joins



# Introduction to Joins



# Introduction to Joins



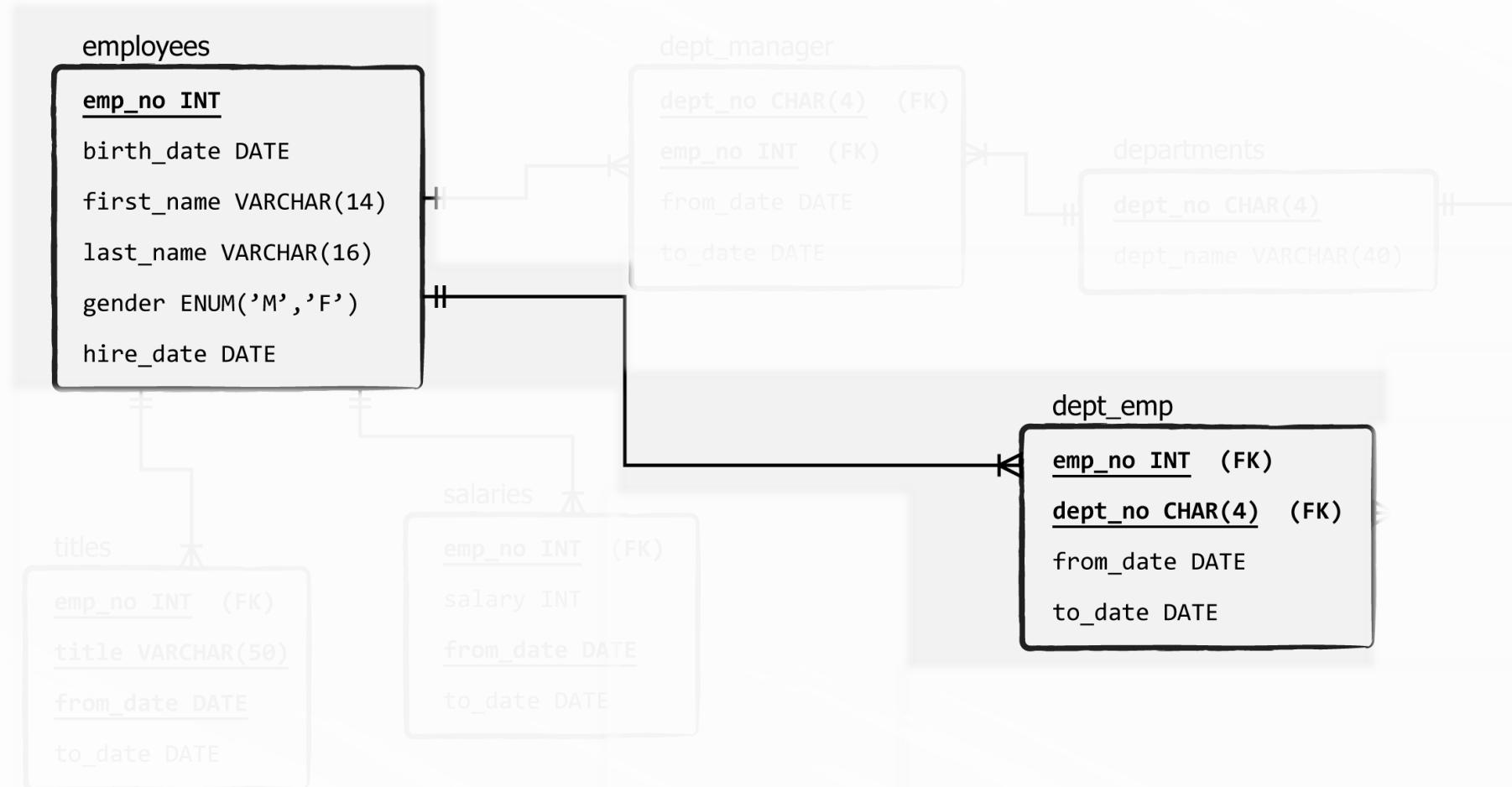
a join shows a *result set*, containing fields *derived from two or more tables*

# Introduction to Joins

- **joins**

- we must find a *related column* from the two tables that contains the same *type* of data

# Introduction to Joins

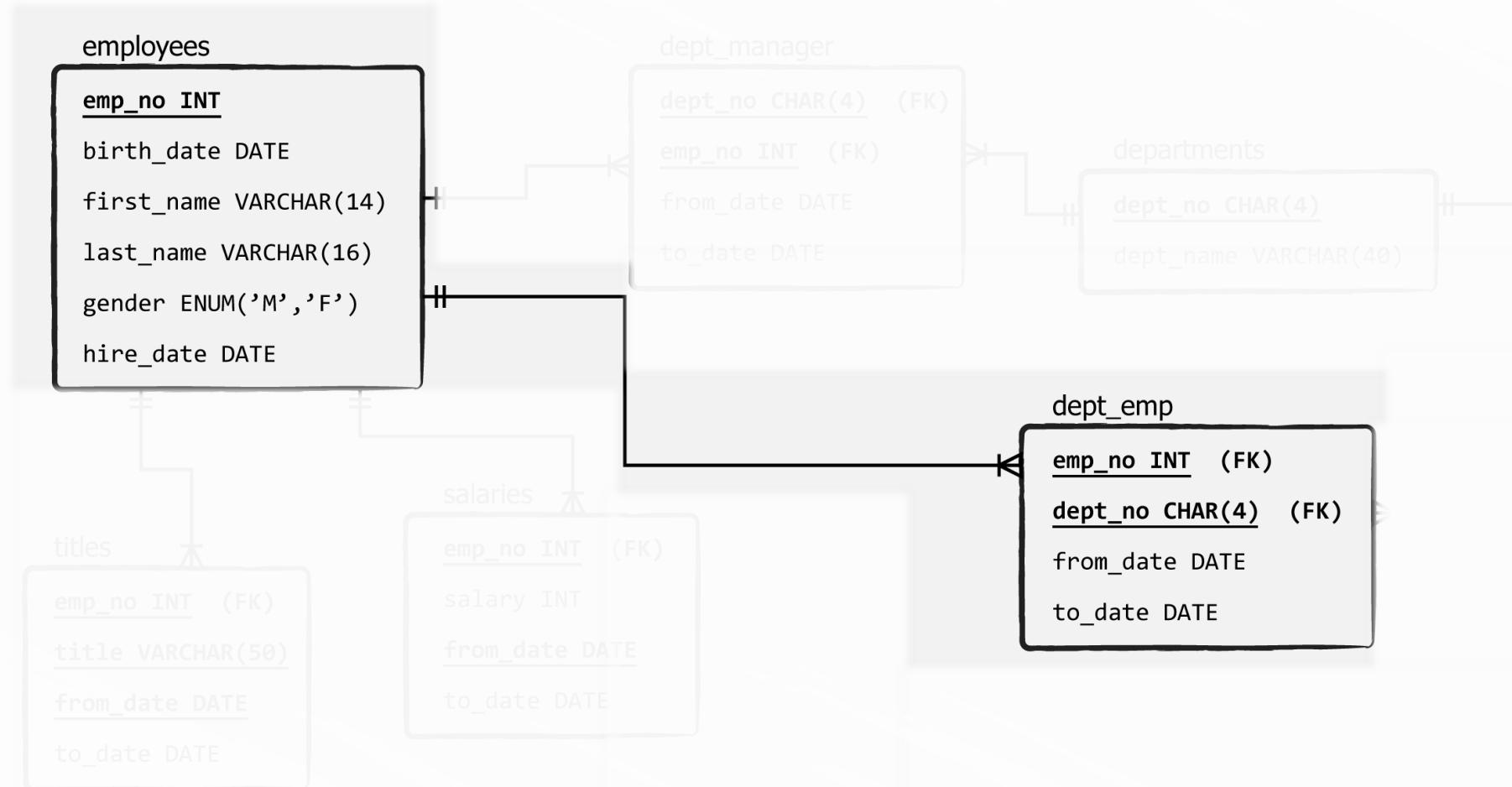


# Introduction to Joins

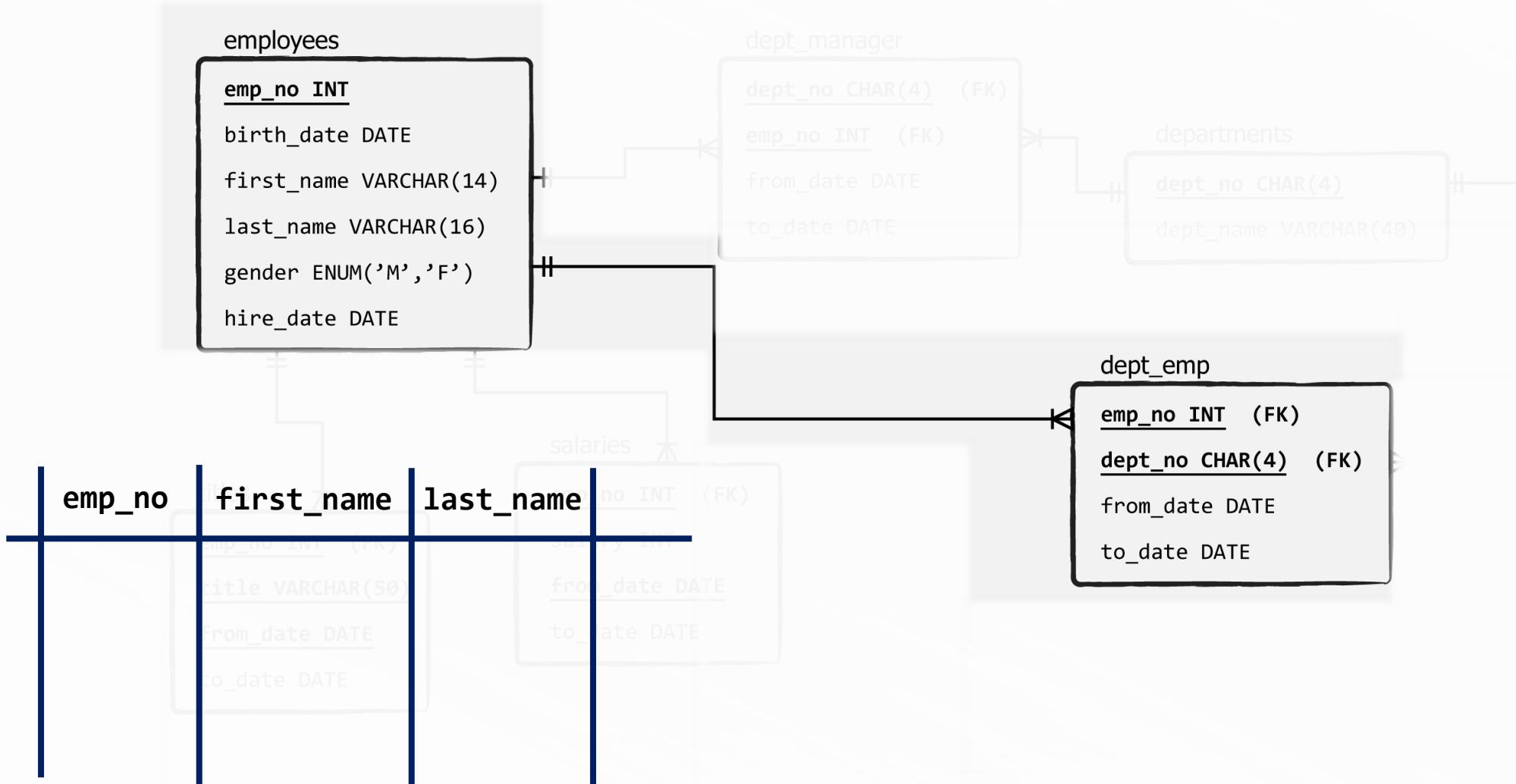
- **joins**

- we must find a *related column* from the two tables that contains the same *type* of data
- we will be free to add columns from these two tables to our output

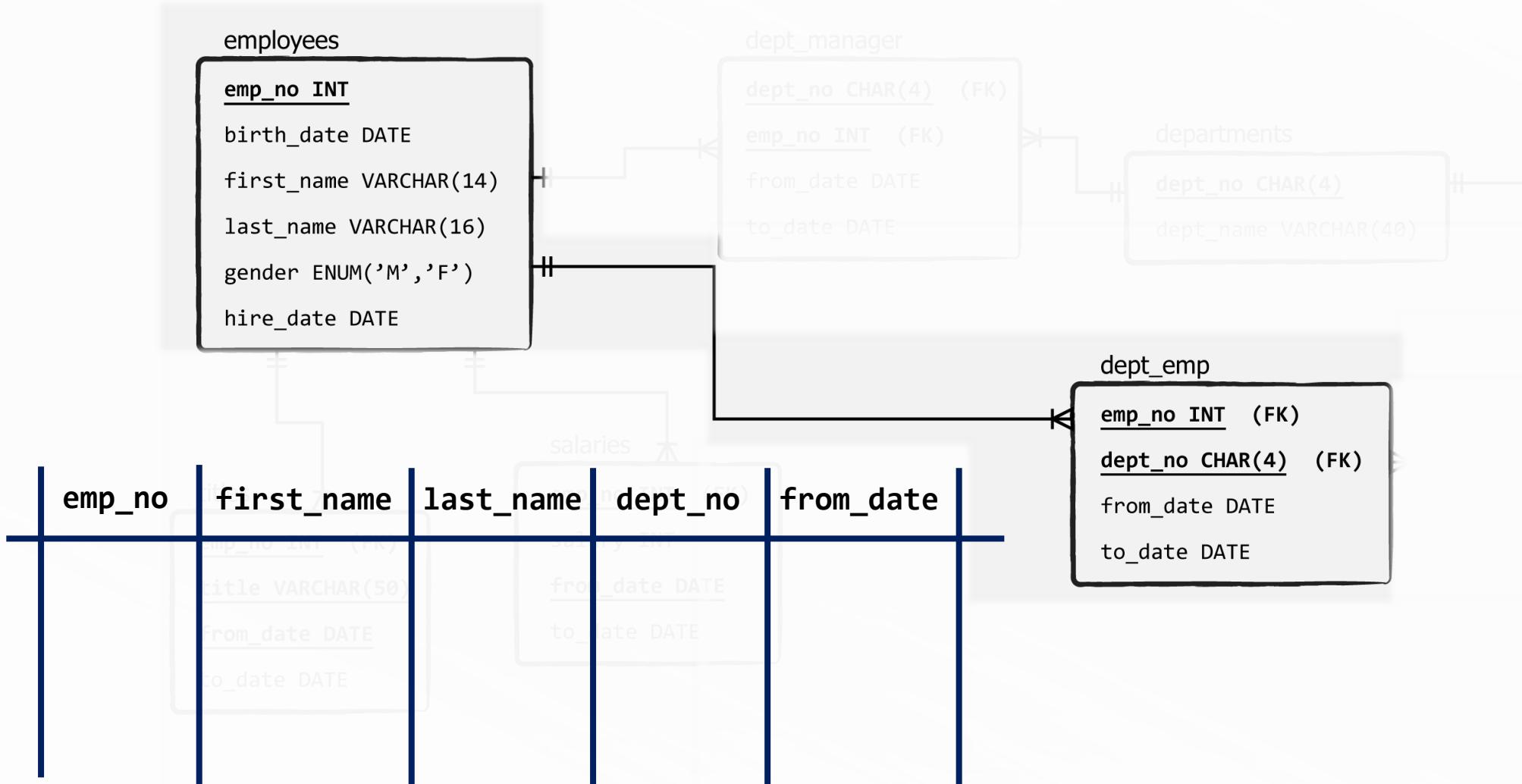
# Introduction to Joins



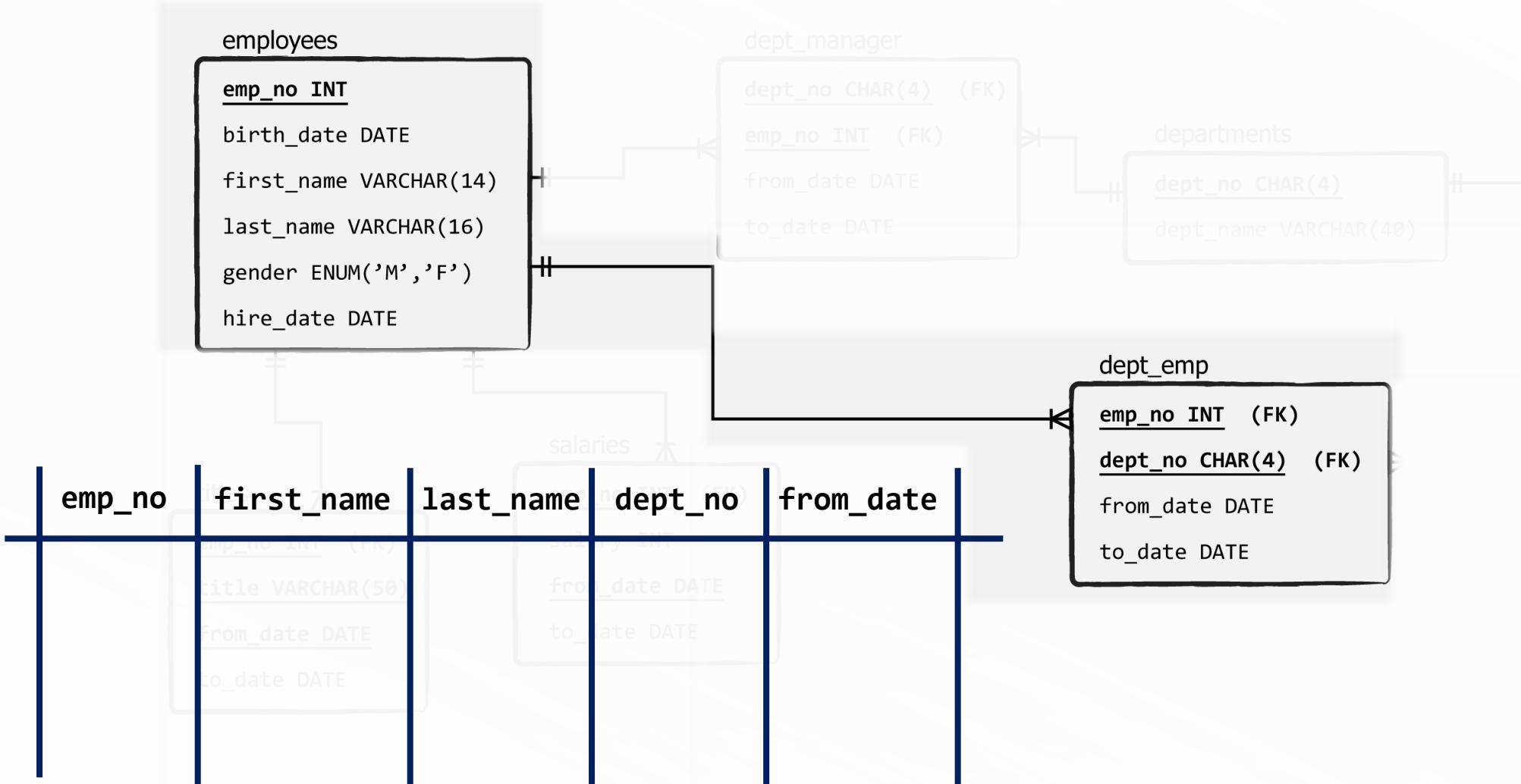
# Introduction to Joins



# Introduction to Joins



# Introduction to Joins

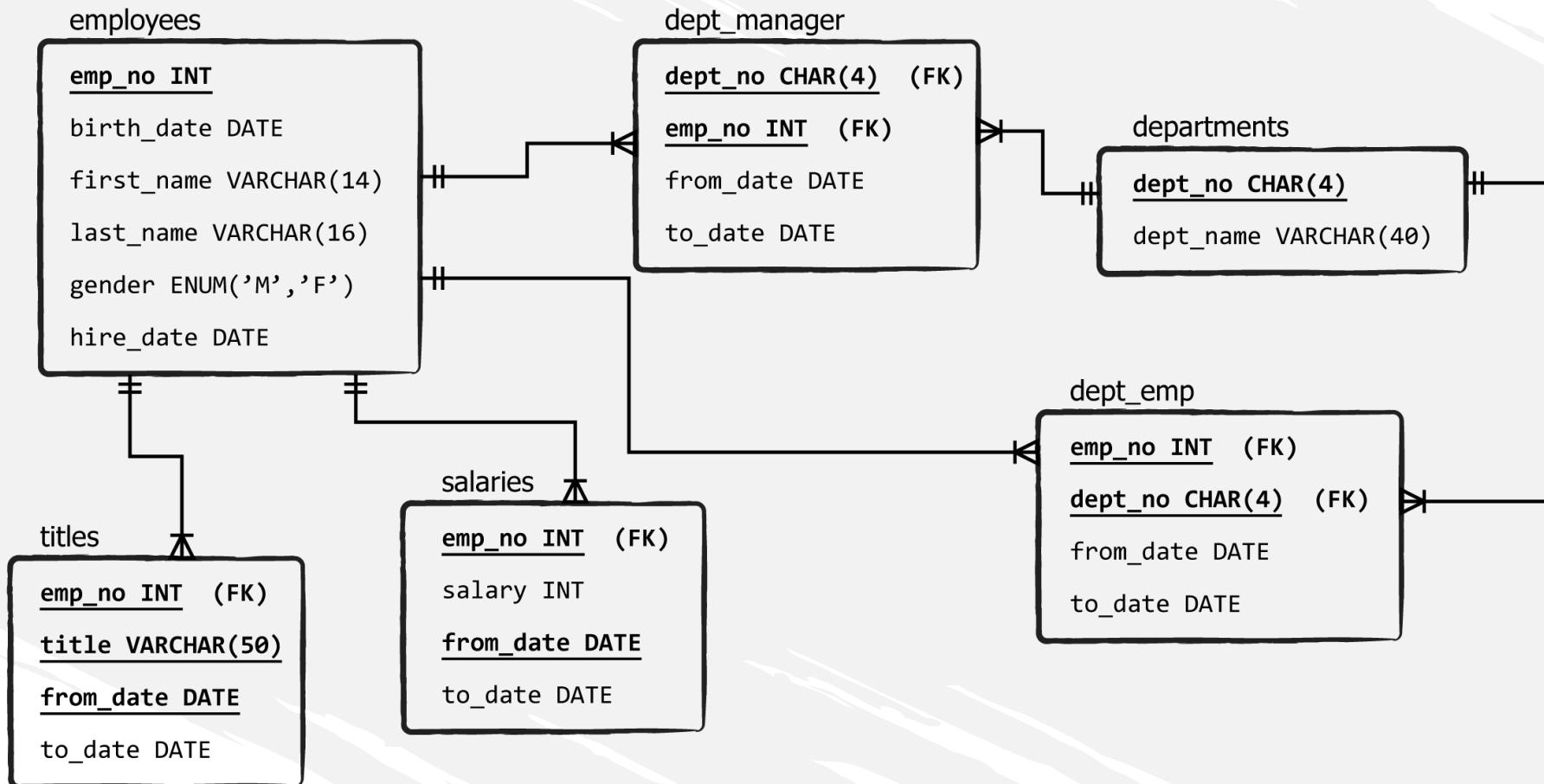


# Introduction to Joins

- **joins**

- the columns you use to relate tables must represent the same object, such as *id*
- the tables you are considering need not be logically adjacent

# Introduction to Joins



# Introduction to Joins

- We will use two duplicate tables:

- ‘departments\_dup’
- ‘dept\_manager\_dup’

Next:

**INNER JOIN**



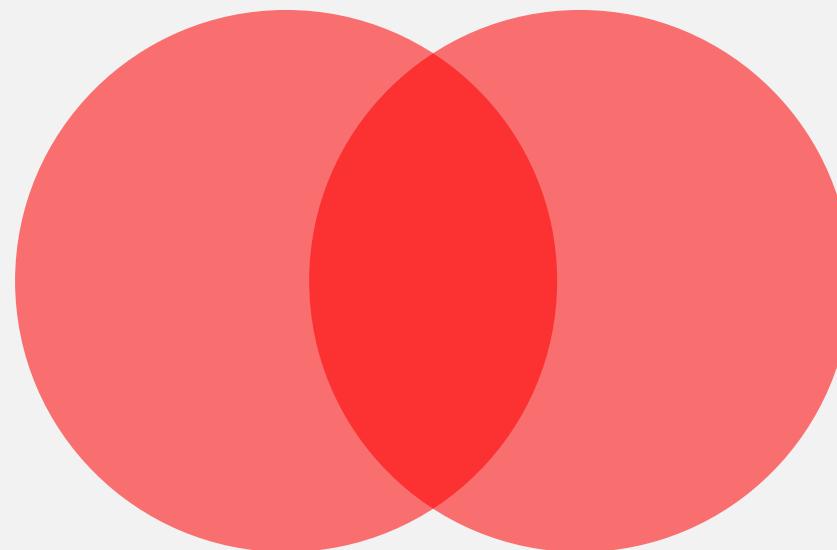
# INNER JOIN

# INNER JOIN

- **INNER JOIN**

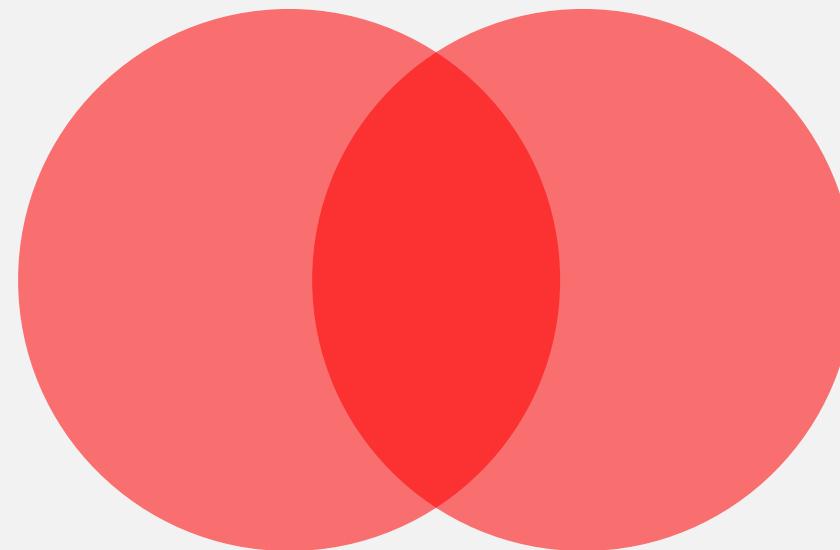
# INNER JOIN

- **INNER JOIN**



# INNER JOIN

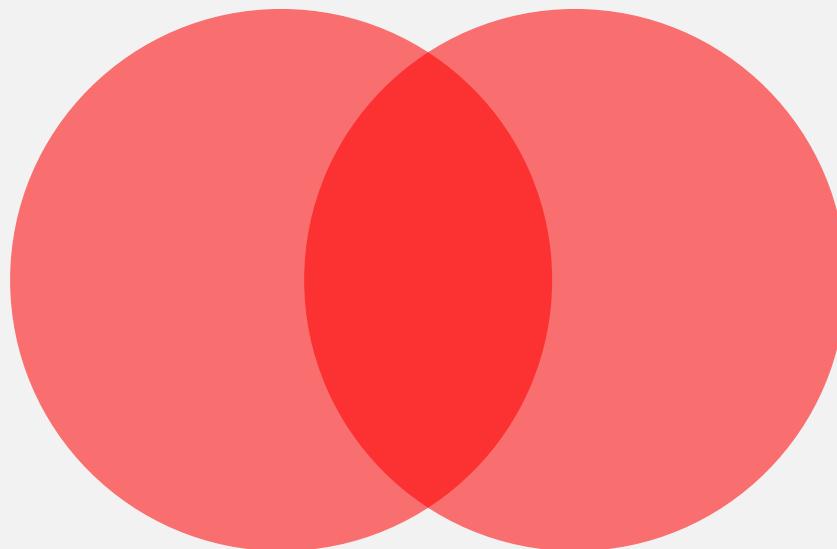
- INNER JOIN



Venn diagram

# INNER JOIN

- **INNER JOIN**



## Venn diagram

a mathematical tool representing all possible logical relations  
between a finite collection of sets

# INNER JOIN

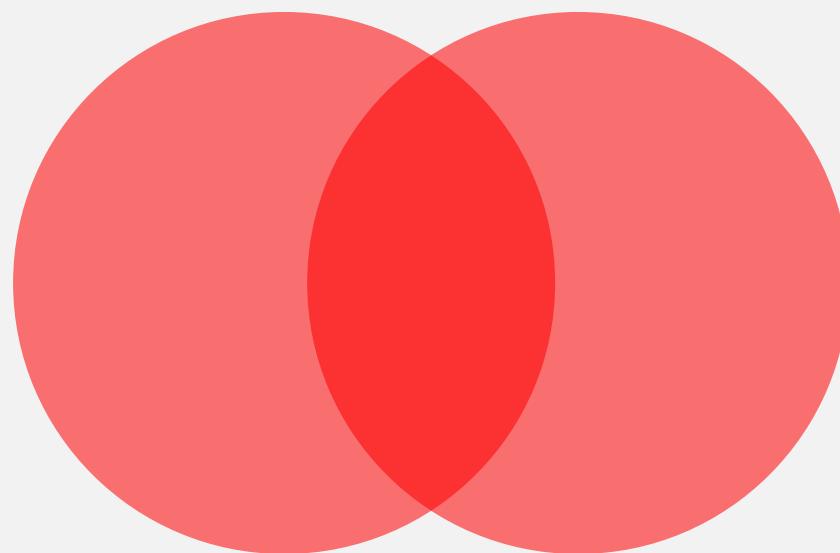
dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE



## Venn diagram

a mathematical tool representing all possible logical relations  
between a finite collection of sets

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

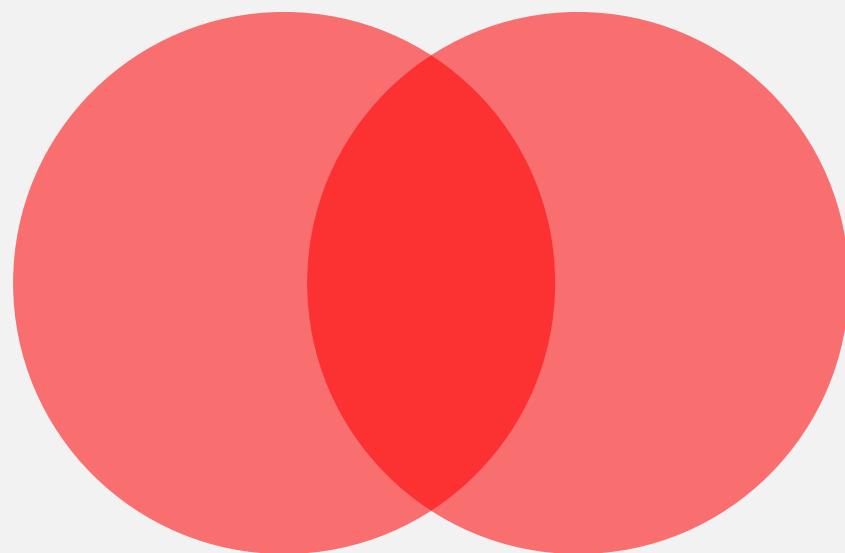
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



## Venn diagram

a mathematical tool representing all possible logical relations  
between a finite collection of sets

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

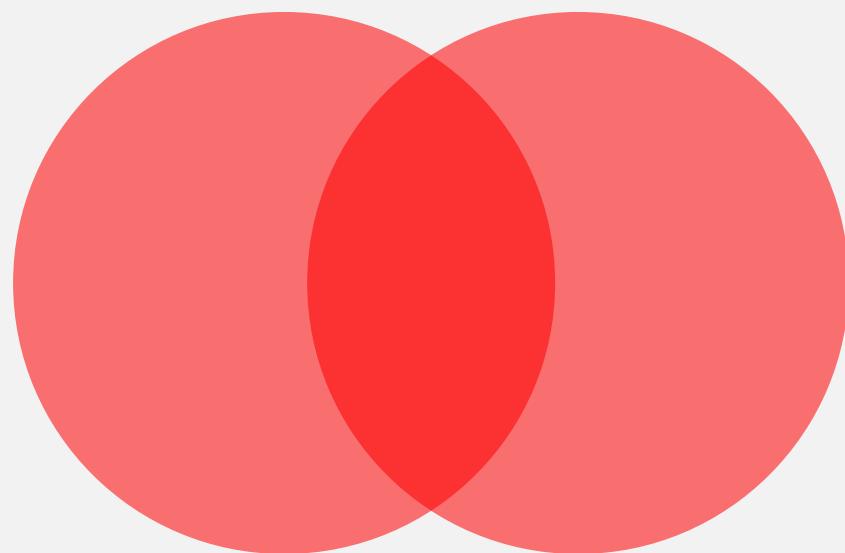
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



Which will be the related column here?

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

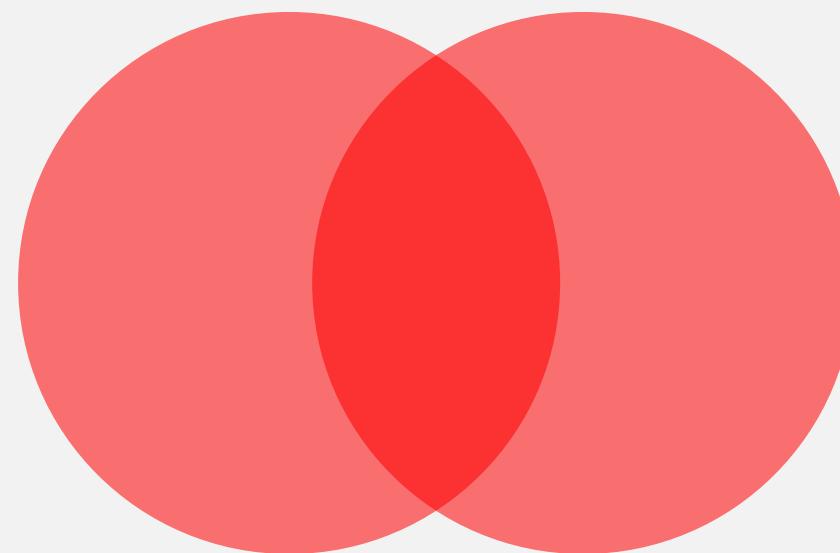
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



Related column: dept\_no

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

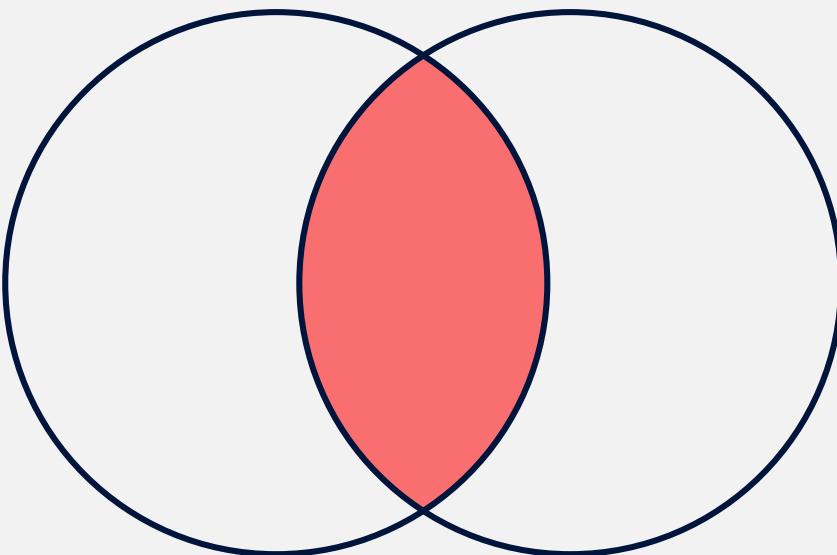
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



the area that belongs to both circles, which is filled with red, represents all records belonging to both the “Department Manager Duplicate” and the “Departments Duplicate” tables

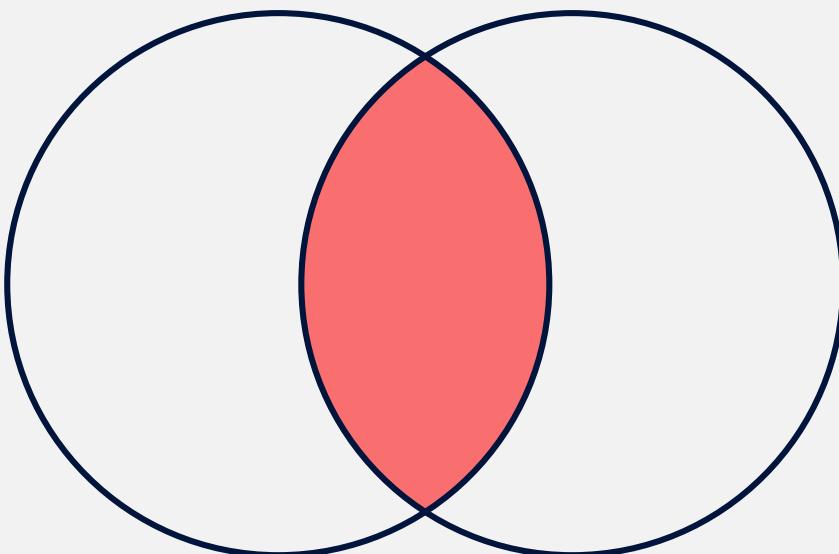
# INNER JOIN

dept\_manager\_dup

dept_no CHAR(4)
emp_no INT
from_date DATE
to_date DATE

departments\_dup

dept_no CHAR(4)
dept_name VARCHAR(40)



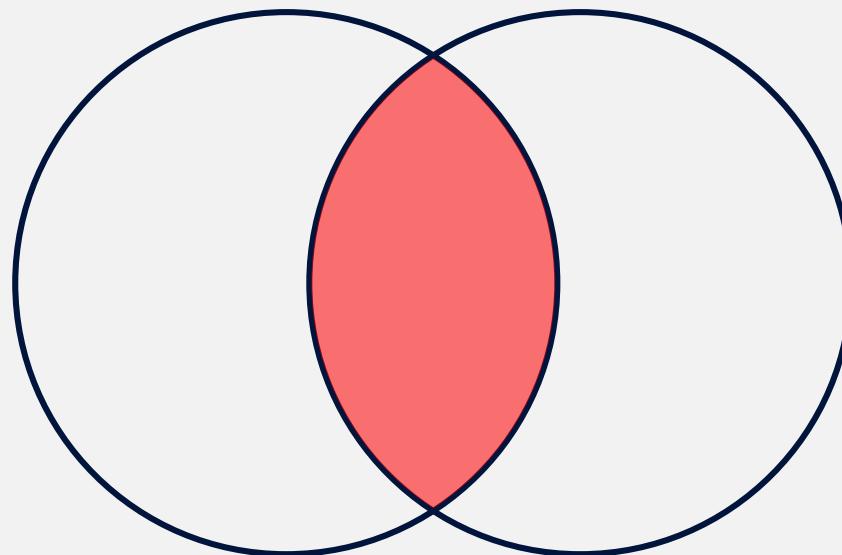
result set

the area that belongs to both circles, which is filled with red, represents all records belonging to *both* the “Department Manager Duplicate” and the “Departments Duplicate” tables

# INNER JOIN

- **INNER JOIN**

can help us extract this result set

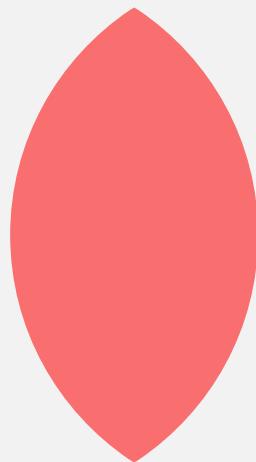


result set

# INNER JOIN

- **INNER JOIN**

can help us extract this result set



result set

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

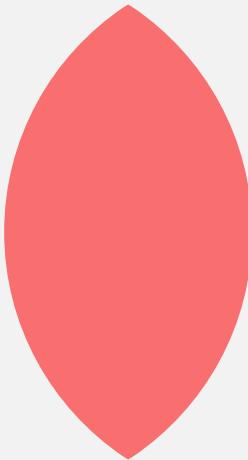
dept\_no CHAR(4)

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# INNER JOIN



**matching values = matching records**

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

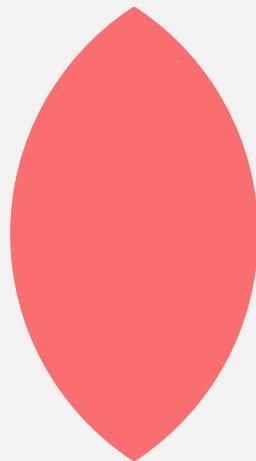
to\_date DATE

dept\_no CHAR(4)

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



**matching values = matching records**

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

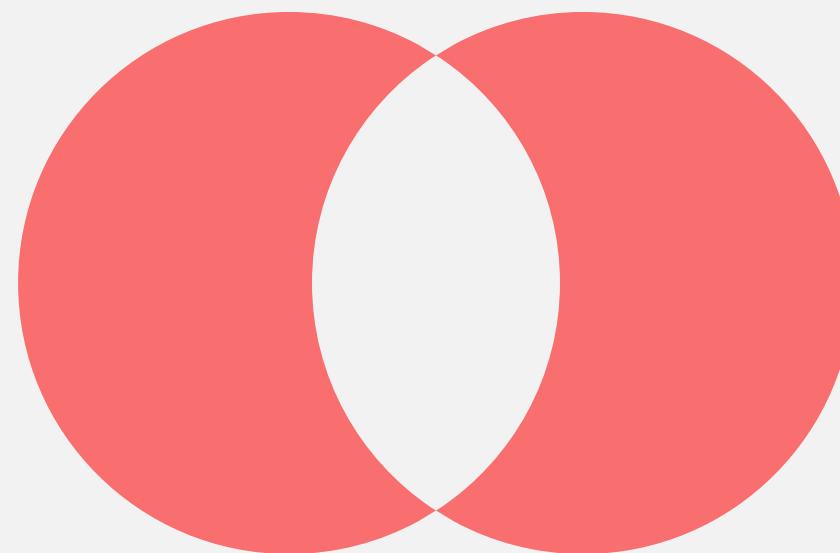
from\_date DATE

to\_date DATE

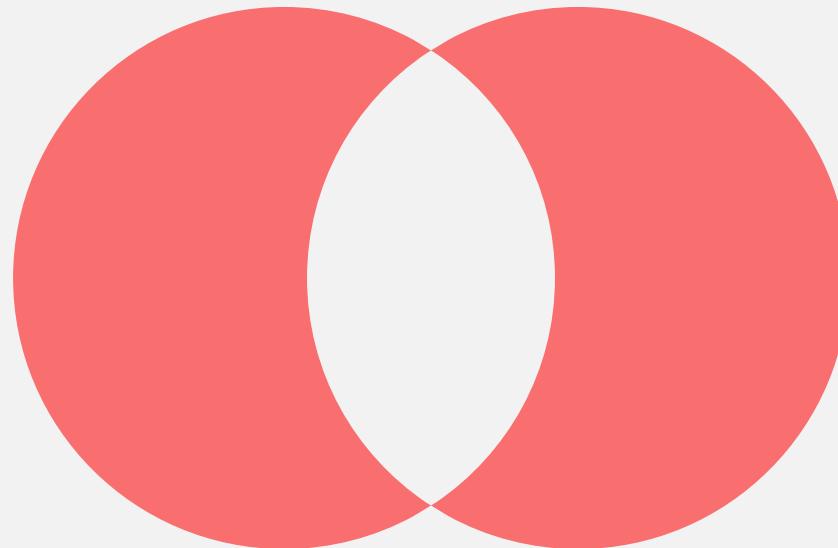
departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



# INNER JOIN



**non-matching values = non-matching records**

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

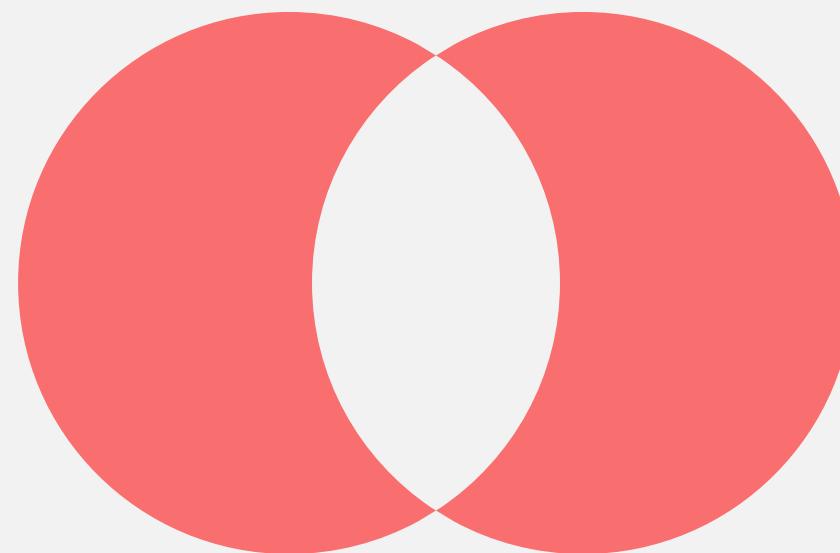
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



**non-matching values = non-matching records**

# INNER JOIN

## INNER JOIN



SQL

```
SELECT
    table_1.column_name(s), table_2.column_name(s)
FROM
    table_1
JOIN
    table_2 ON table_1.column_name = table_2.column_name;
```

# INNER JOIN

## INNER JOIN



SQL

```
SELECT
    t1.column_name, t1.column_name, ..., t2.column_name, ...
FROM
    table_1 t1
JOIN
    table_2 t2 ON t1.column_name = t2.column_name;
```

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

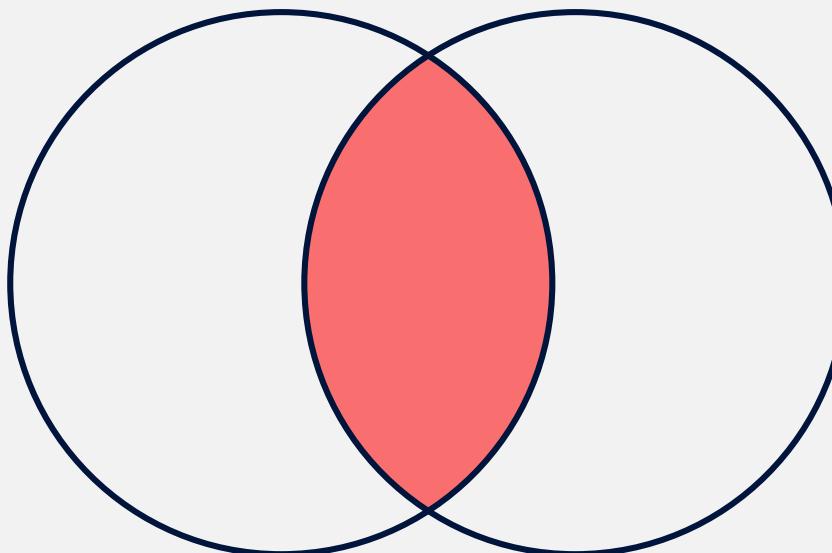
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



# INNER JOIN

M

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

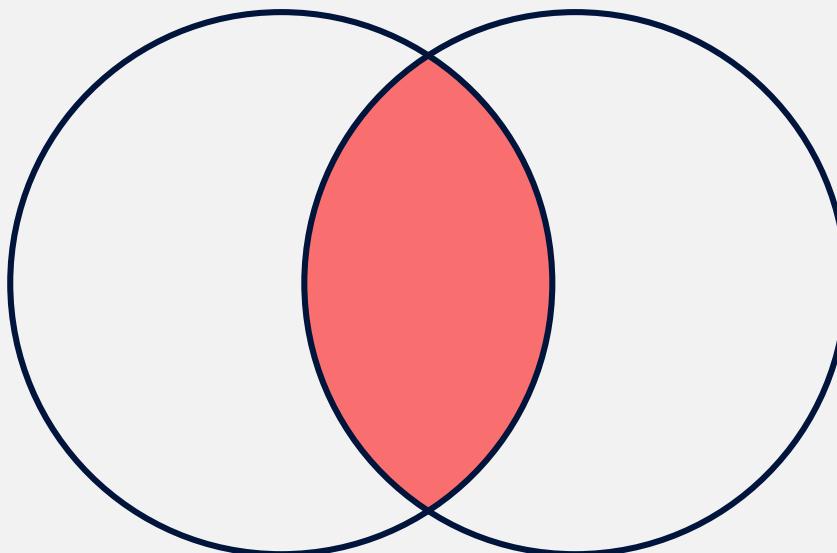
to\_date DATE

D

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



## INNER JOIN

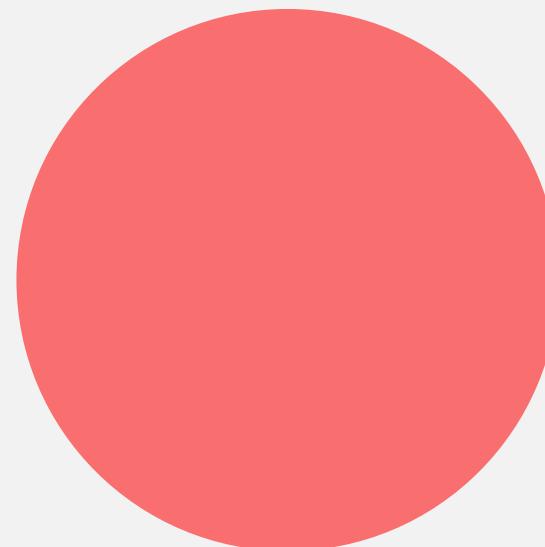
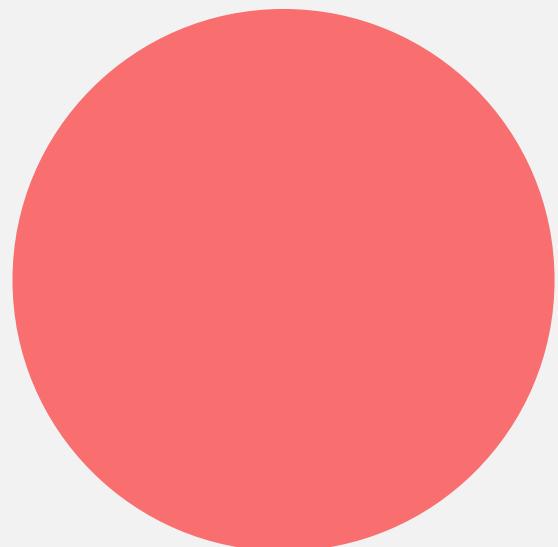
- inner joins extract only records in which the values in the related columns match. Null values, or values appearing in just one of the two tables and not appearing in the other, are not displayed
  - only non-null matching values are in play

## INNER JOIN

- And what if such *matching values* did not exist?

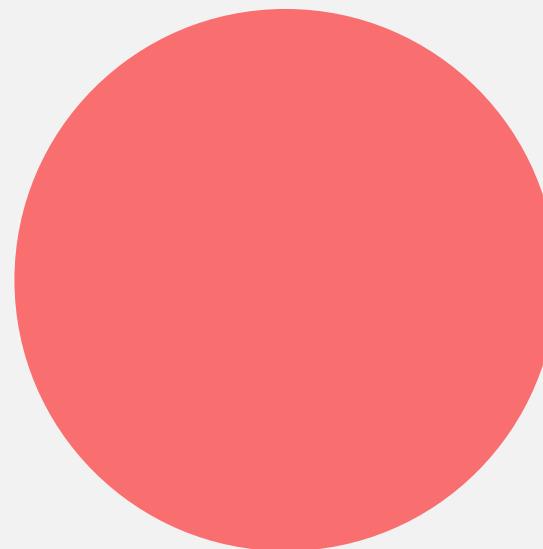
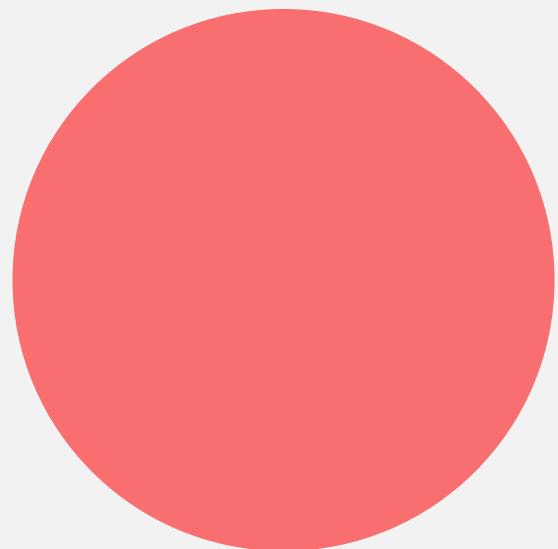
## INNER JOIN

- And what if such *matching values* did not exist?



## INNER JOIN

- And what if such *matching values* did not exist?



Simply, the result set will be empty. There will be no link between the two tables.

A large, modern conference room with rows of chairs facing a central table. The room has large windows overlooking a city skyline.

# Duplicate Records

# Duplicate Records

- duplicate records, also known as duplicate rows, are identical rows in an SQL table

# Duplicate Records

- duplicate records, also known as duplicate rows, are identical rows in an SQL table
  - for a pair of duplicate records, the values in each column coincide

# Duplicate Records

- duplicate rows are not always allowed in a database or a data table

# Duplicate Records

- duplicate rows are not always allowed in a database or a data table
  - they are sometimes encountered, especially in new, raw, or uncontrolled data

# Duplicate Records

- duplicate rows are not always allowed in a database or a data table
  - they are sometimes encountered, especially in new, raw, or uncontrolled data

here's how to handle duplicates:

**GROUP BY the field that differs most among records!**

A large conference room with rows of chairs facing a central table. The room has a modern design with a glass partition and a large window overlooking a landscape. The text "LEFT JOIN" is overlaid on the image.

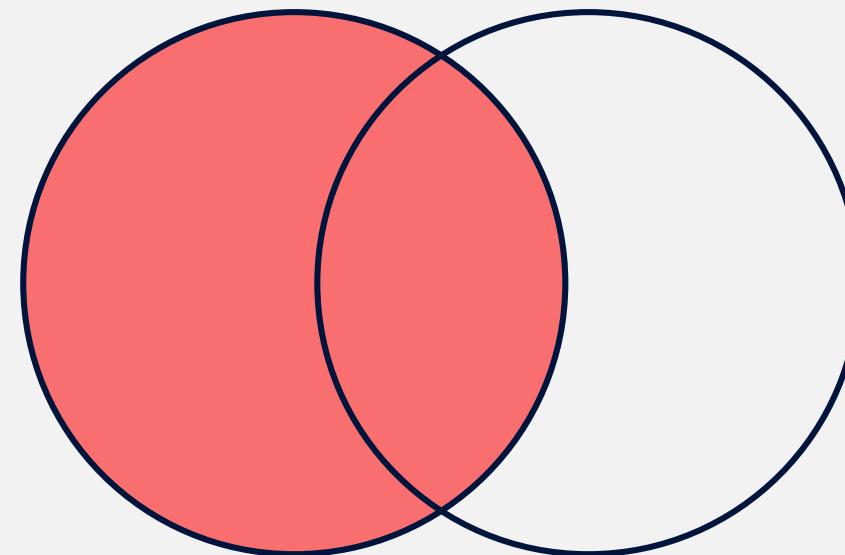
# LEFT JOIN

# LEFT JOIN

- LEFT JOIN

# LEFT JOIN

- **LEFT JOIN**



# LEFT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

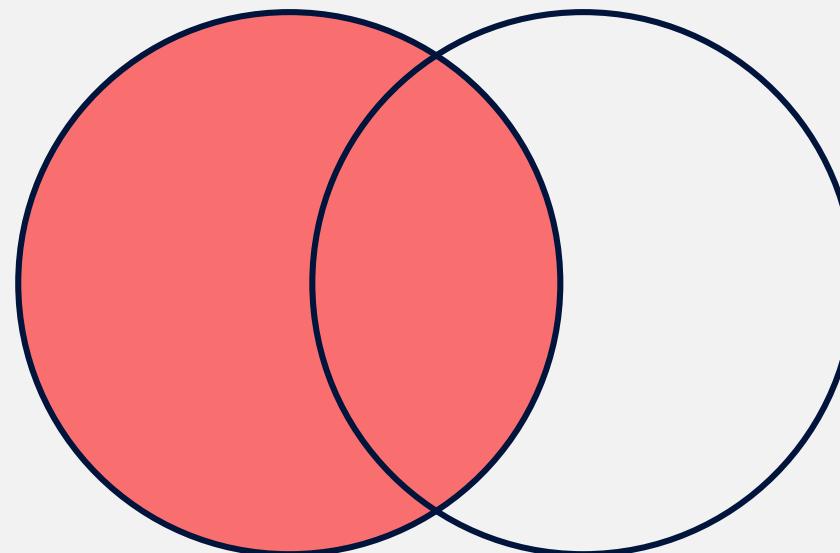
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



# LEFT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

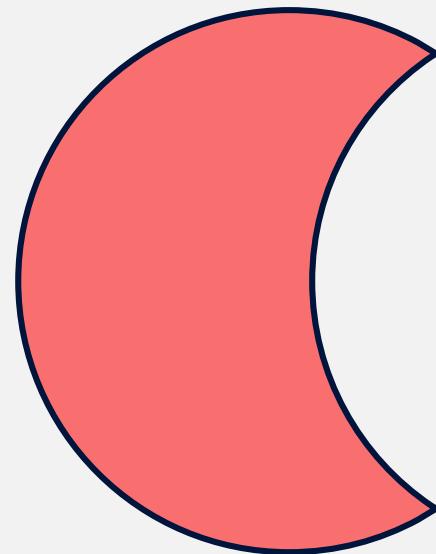
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



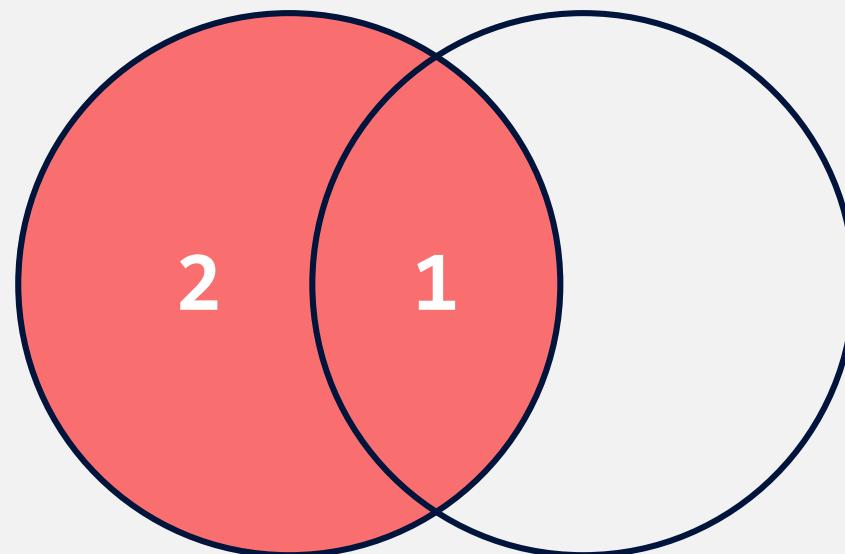
# LEFT JOIN

dept\_manager\_dup

```
dept_no CHAR(4)  
emp_no INT  
from_date DATE  
to_date DATE
```

departments\_dup

```
dept_no CHAR(4)  
dept_name VARCHAR(40)
```



- 1) all matching values of the two tables +
- 2) all values from the left table that match no values from the right table

# LEFT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

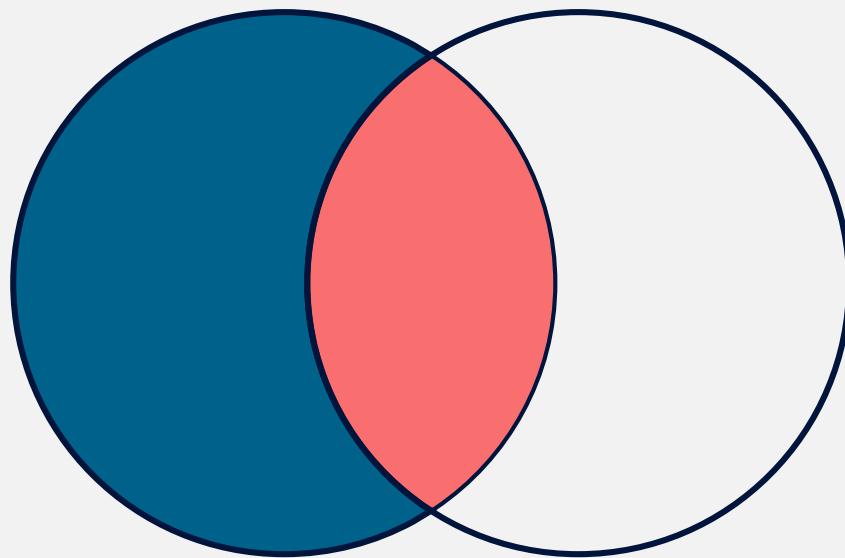
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



all matching values of the two tables +  
all values from the left table that match no values from the right table

# LEFT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

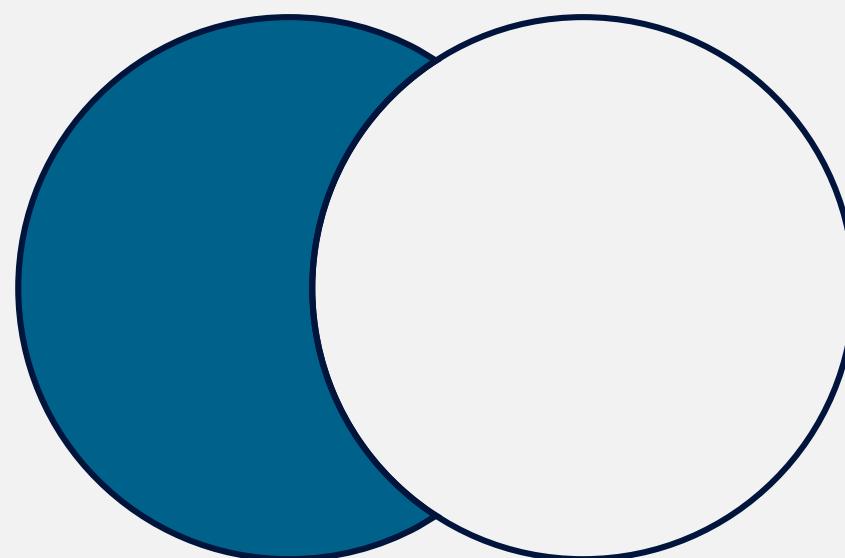
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



~~all matching values of the two tables +~~

all values from the left table that match no values from the right table

# LEFT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

~~all matching values of the two tables +~~

all values from the left table that match no values from the right table

# LEFT JOIN

- **LEFT JOIN**

when working with left joins, the order in which you join tables *matters*

# LEFT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

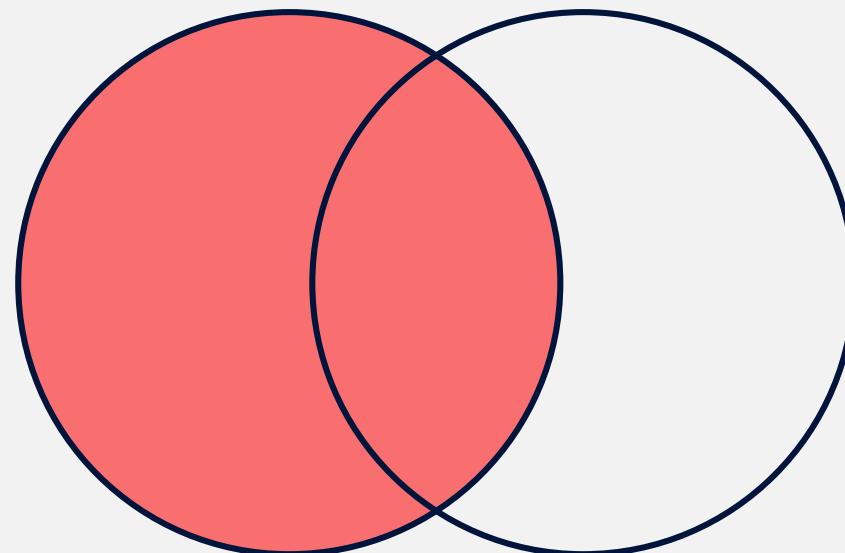
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

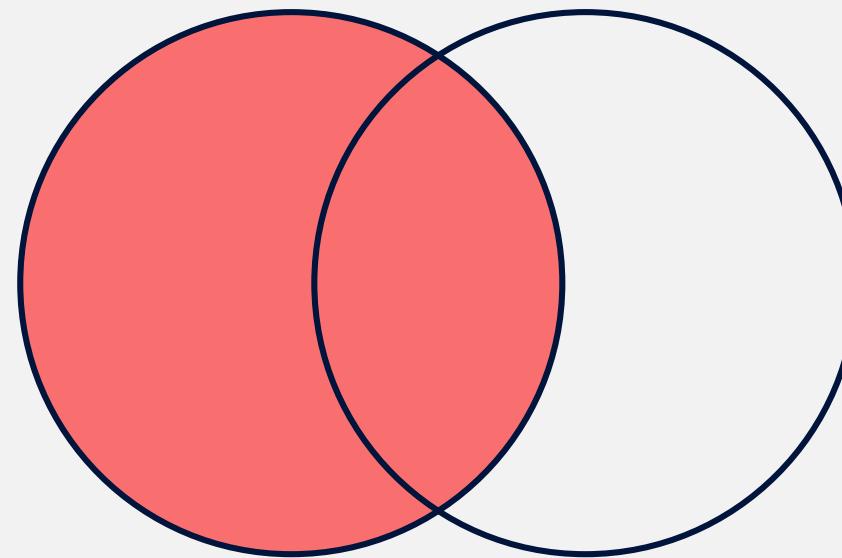


# LEFT JOIN

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

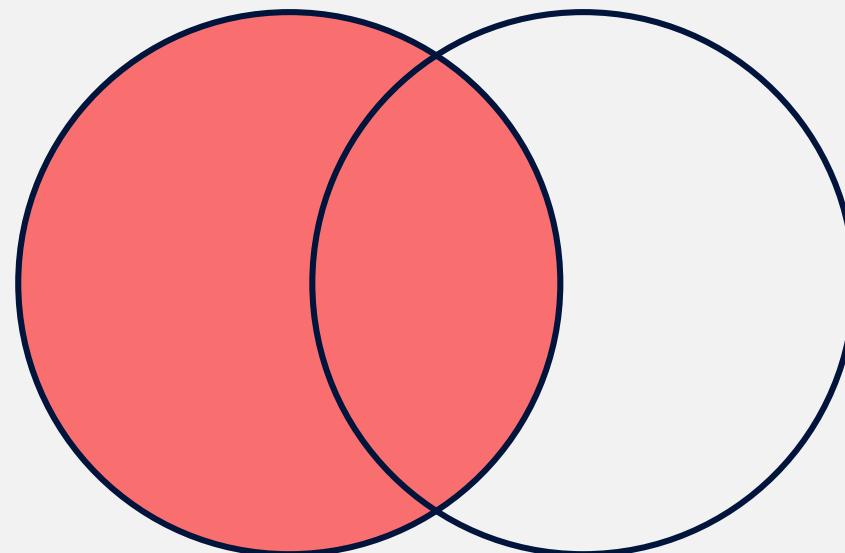
to\_date DATE

departments\_dup

```
dept_no CHAR(4)  
dept_name VARCHAR(40)
```

dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL

# LEFT JOIN

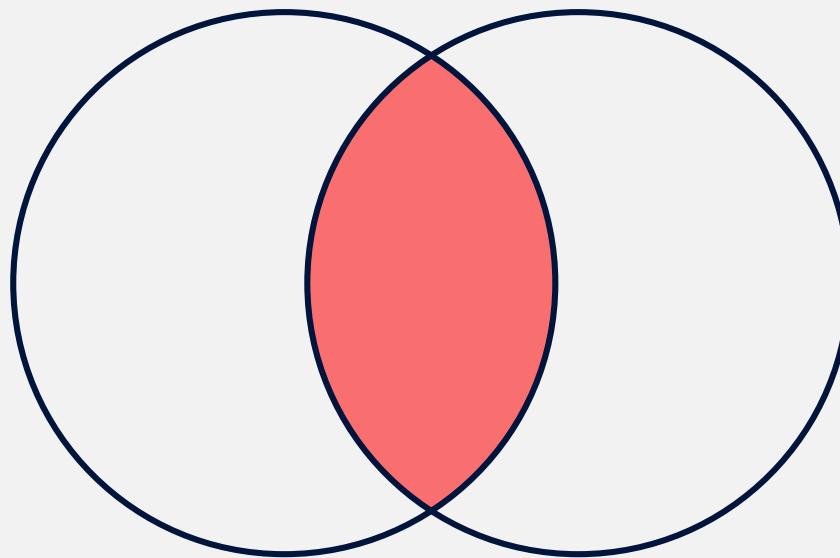


dept\_manager\_dup

```
dept_no CHAR(4)  
emp_no INT  
from_date DATE  
to_date DATE
```

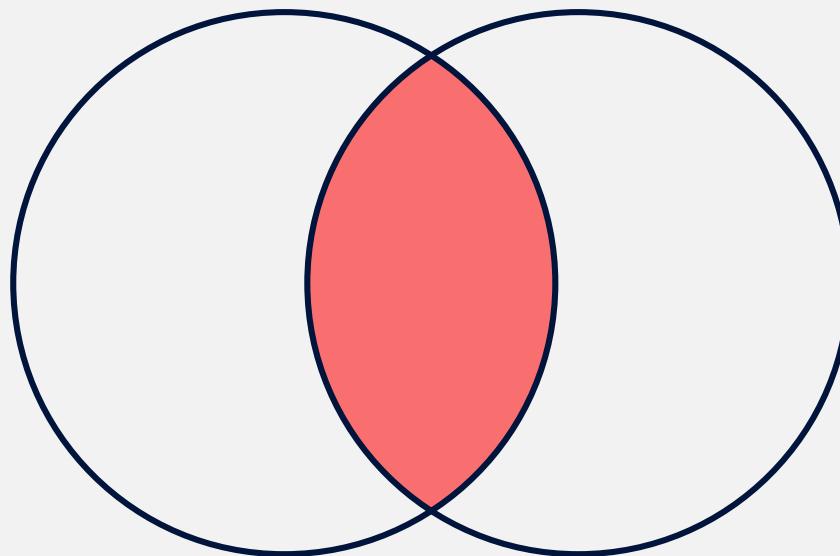
emp_no	dept_no	from_date	to_date
999904	NULL	2017-01-01	NULL
999905	NULL	2017-01-01	NULL
999906	NULL	2017-01-01	NULL
999907	NULL	2017-01-01	NULL
110085	d002	1985-01-01	1989-12-17
110114	d002	1989-12-17	9999-01-01
110183	d003	1985-01-01	1992-03-21
110228	d003	1992-03-21	9999-01-01
110303	d004	1985-01-01	1988-09-09
110344	d004	1988-09-09	1992-08-02
110386	d004	1992-08-02	1996-08-30
110420	d004	1996-08-30	9999-01-01
110511	d005	1985-01-01	1992-04-25
110567	d005	1992-04-25	9999-01-01
110725	d006	1985-01-01	1989-05-06
110765	d006	1989-05-06	1991-09-12
110800	d006	1991-09-12	1994-06-28
110854	d006	1994-06-28	9999-01-01
111035	d007	1985-01-01	1991-03-07
111133	d007	1991-03-07	9999-01-01
111400	d008	1985-01-01	1991-04-08
111534	d008	1991-04-08	9999-01-01
111692	d009	1985-01-01	1988-10-17
111784	d009	1988-10-17	1992-09-08
111877	d009	1992-09-08	1996-01-03
111939	d009	1996-01-03	9999-01-01

# LEFT JOIN



INNER join

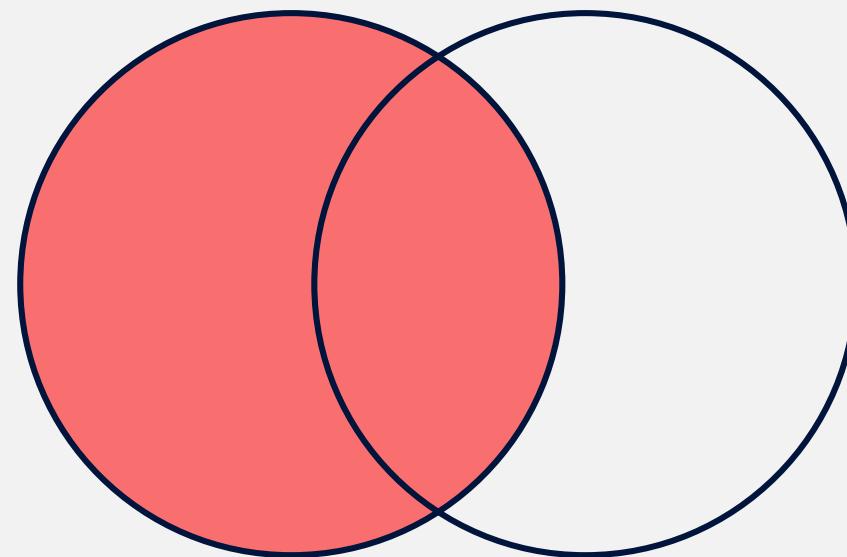
# LEFT JOIN



INNER join

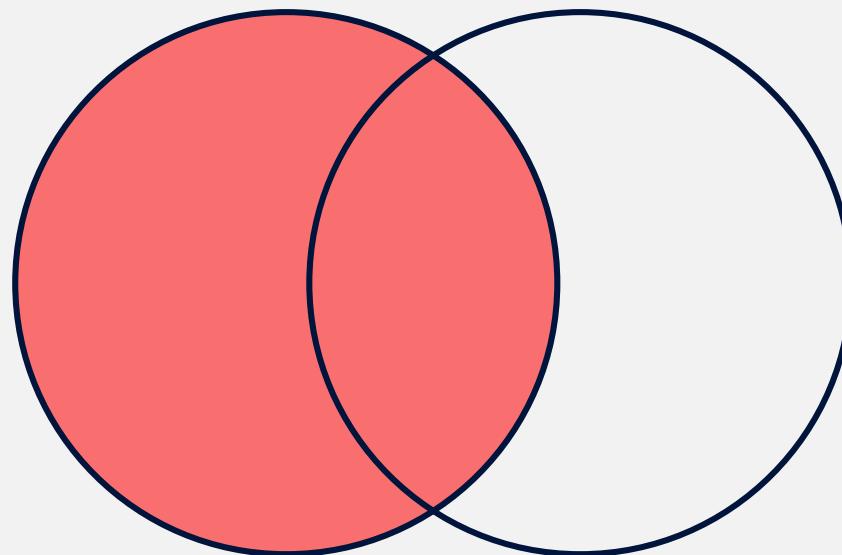
the result set is in the *inner* part of the Venn diagram

# LEFT JOIN



LEFT join

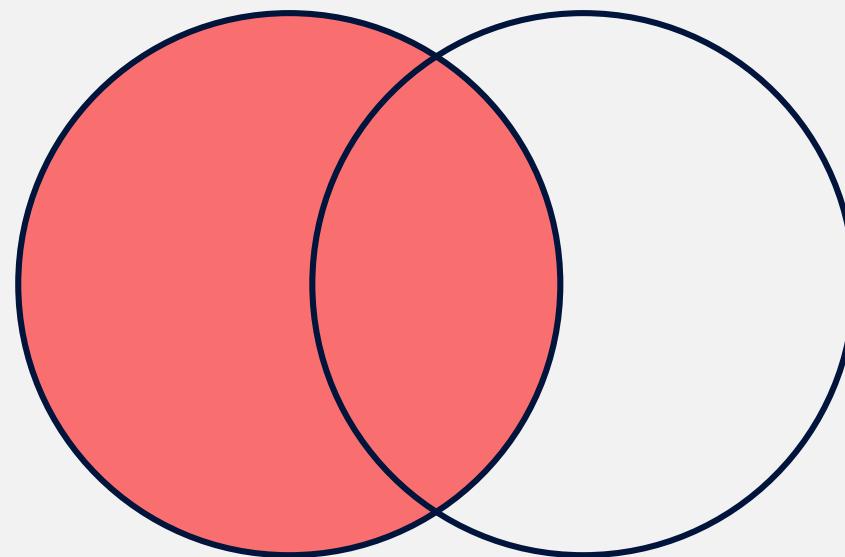
# LEFT JOIN



LEFT join

in the output obtained you have data from the  
*outer* part of the Venn diagram too

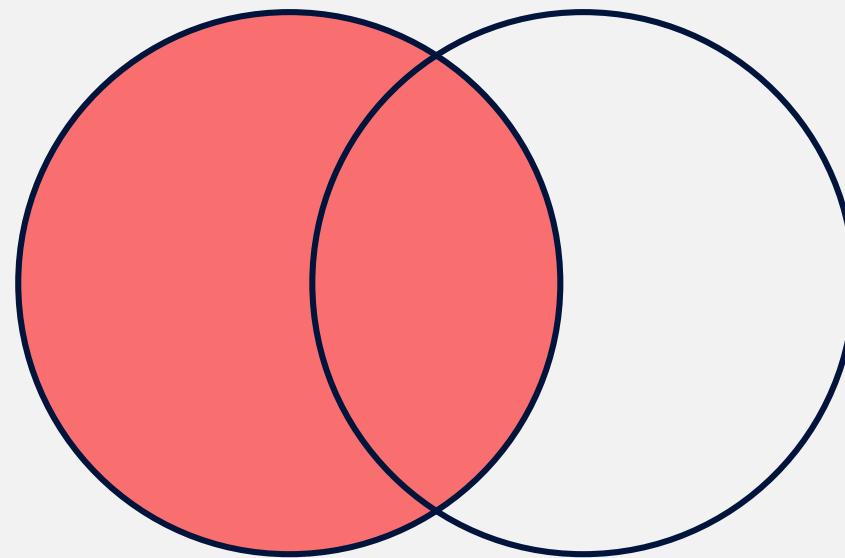
# LEFT JOIN



LEFT join = LEFT OUTER join

in the output obtained you have data from the  
*outer* part of the Venn diagram too

# LEFT JOIN



LEFT join = LEFT OUTER join

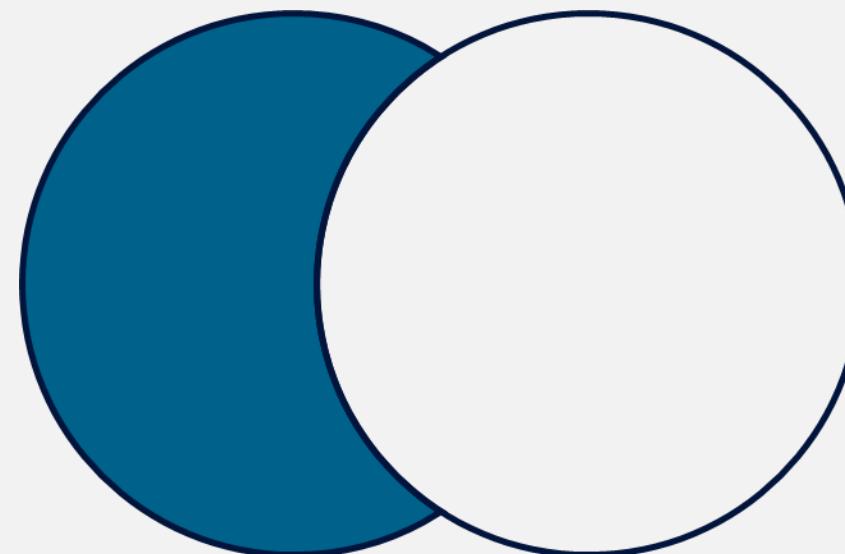
if you are using a left join, it will always be an  
OUTER type of join

## LEFT JOIN

- left joins can deliver a list with all records from the left table that do not match any rows from the right table

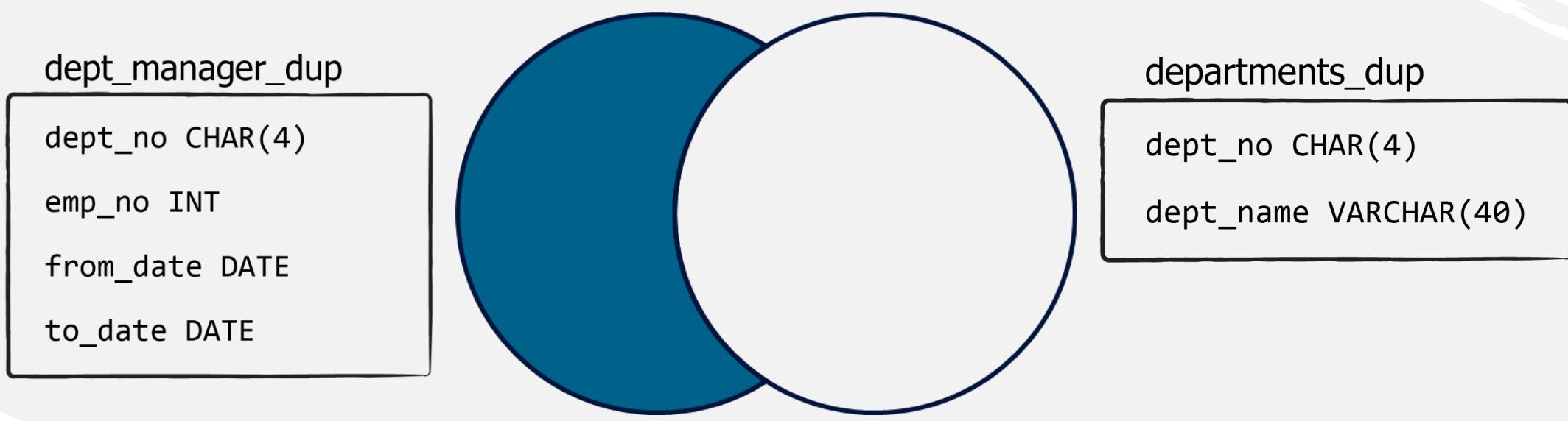
## LEFT JOIN

- left joins can deliver a list with all records from the left table that do not match any rows from the right table



# LEFT JOIN

left joins can deliver a list with all records from the left table that do not match any rows from the right table



# LEFT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



SQL

```
SELECT
    t1.column_name, t1.column_name, ..., t2.column_name, ...
FROM
    table_1 t1
JOIN
    table_2 t2 ON t1.column_name = t2.column_name
WHERE
    column_name ... IS NULL;
```

A large conference room with rows of black office chairs facing a long rectangular table. The room has a modern design with a grid-patterned ceiling and floor-to-ceiling windows on one wall, offering a view of an outdoor area.

**RIGHT JOIN**

# RIGHT JOIN

- **RIGHT JOIN**

## RIGHT JOIN

- **RIGHT JOIN**

their functionality is identical to LEFT JOINS, with the only difference being that the direction of the operation is inverted

## RIGHT JOIN

- **RIGHT JOIN** = **RIGHT OUTER JOIN**

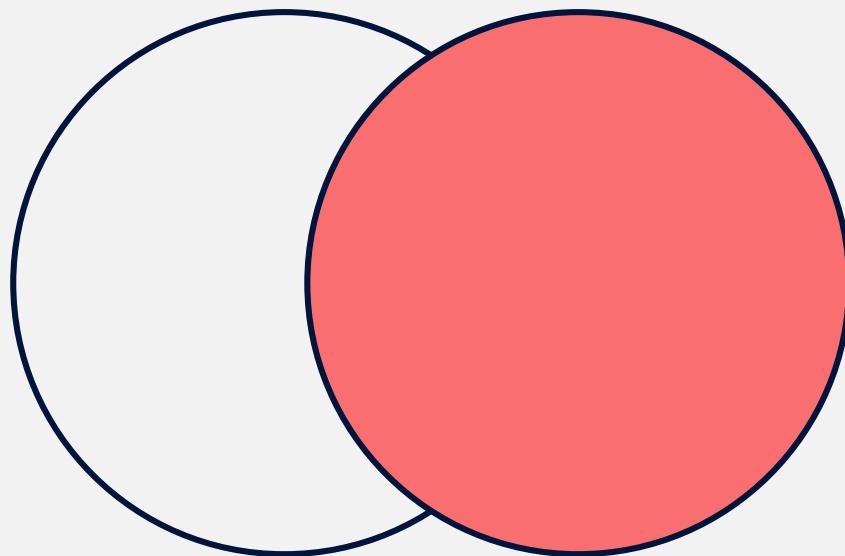
their functionality is identical to LEFT JOINS, with the only difference being that the direction of the operation is inverted

# RIGHT JOIN

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

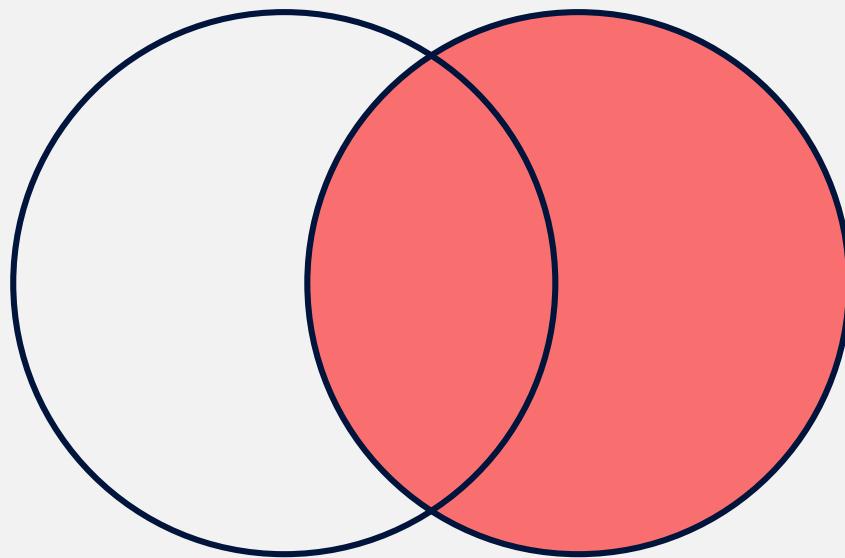
to\_date DATE

# RIGHT JOIN

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

# RIGHT JOIN

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

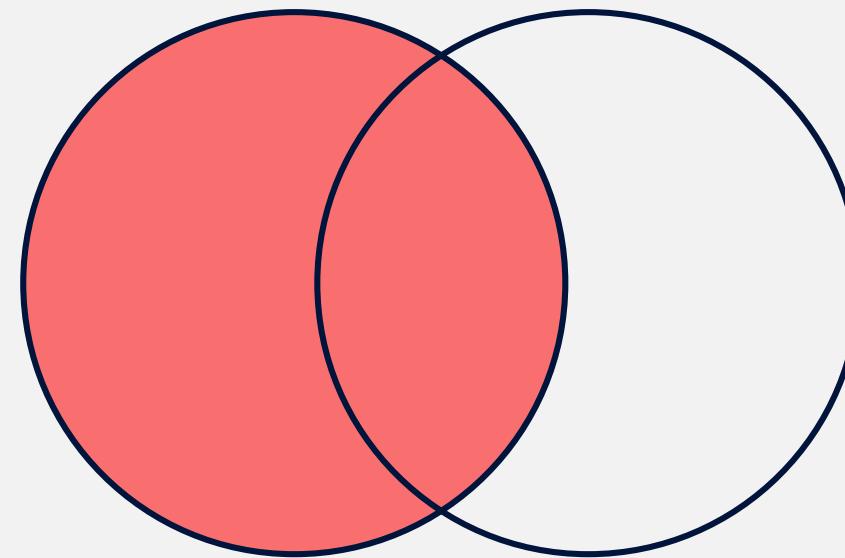
dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

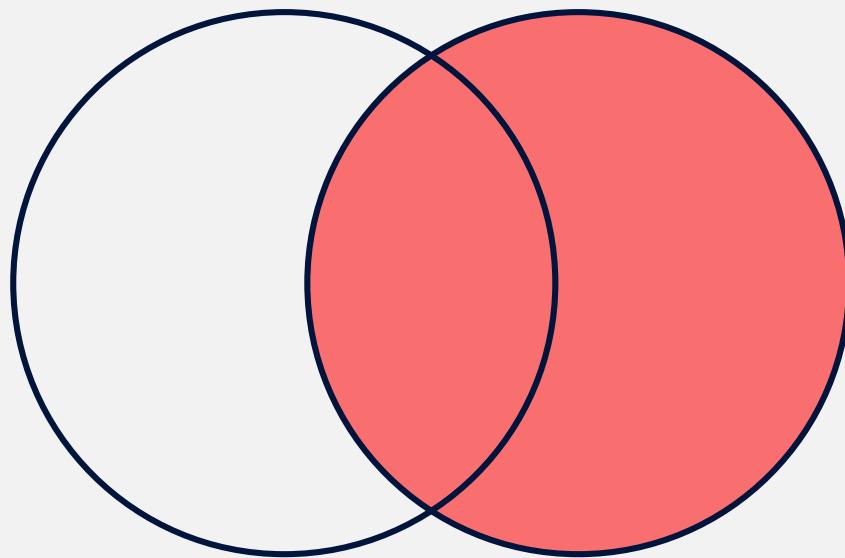
to\_date DATE



# RIGHT JOIN

departments\_dup

```
dept_no CHAR(4)  
dept_name VARCHAR(40)
```



dept\_manager\_dup

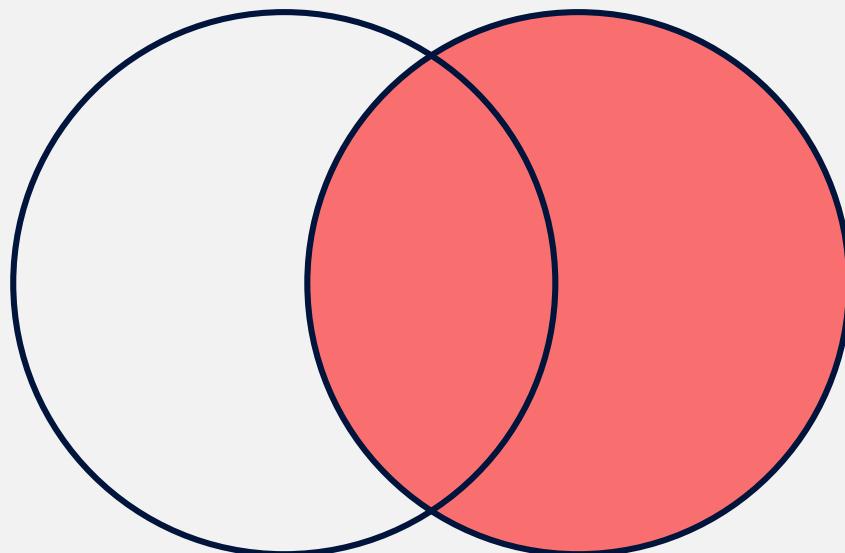
```
dept_no CHAR(4)  
emp_no INT  
from_date DATE  
to_date DATE
```

departments\_dup

```
dept_no CHAR(4)  
dept_name VARCHAR(40)
```

dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL

## RIGHT JOIN



dept\_manager\_dup

```
dept_no CHAR(4)  
emp_no INT  
from_date DATE  
to_date DATE
```

emp_no	dept_no	from_date	to_date
999904	NULL	2017-01-01	NULL
999905	NULL	2017-01-01	NULL
999906	NULL	2017-01-01	NULL
999907	NULL	2017-01-01	NULL
110085	d002	1985-01-01	1989-12-17
110114	d002	1989-12-17	9999-01-01
110183	d003	1985-01-01	1992-03-21
110228	d003	1992-03-21	9999-01-01
110303	d004	1985-01-01	1988-09-09
110344	d004	1988-09-09	1992-08-02
110386	d004	1992-08-02	1996-08-30
110420	d004	1996-08-30	9999-01-01
110511	d005	1985-01-01	1992-04-25
110567	d005	1992-04-25	9999-01-01
110725	d006	1985-01-01	1989-05-06
110765	d006	1989-05-06	1991-09-12
110800	d006	1991-09-12	1994-06-28
110854	d006	1994-06-28	9999-01-01
111035	d007	1985-01-01	1991-03-07
111133	d007	1991-03-07	9999-01-01
111400	d008	1985-01-01	1991-04-08
111534	d008	1991-04-08	9999-01-01
111692	d009	1985-01-01	1988-10-17
111784	d009	1988-10-17	1992-09-08
111877	d009	1992-09-08	1996-01-03
111939	d009	1996-01-03	9999-01-01

## RIGHT JOIN

- Whether we run a RIGHT JOIN or a LEFT JOIN with an *inverted tables order*, we will obtain the same output, right?

## RIGHT JOIN

- Whether we run a RIGHT JOIN or a LEFT JOIN with an *inverted* tables order, we will obtain the same output, right?

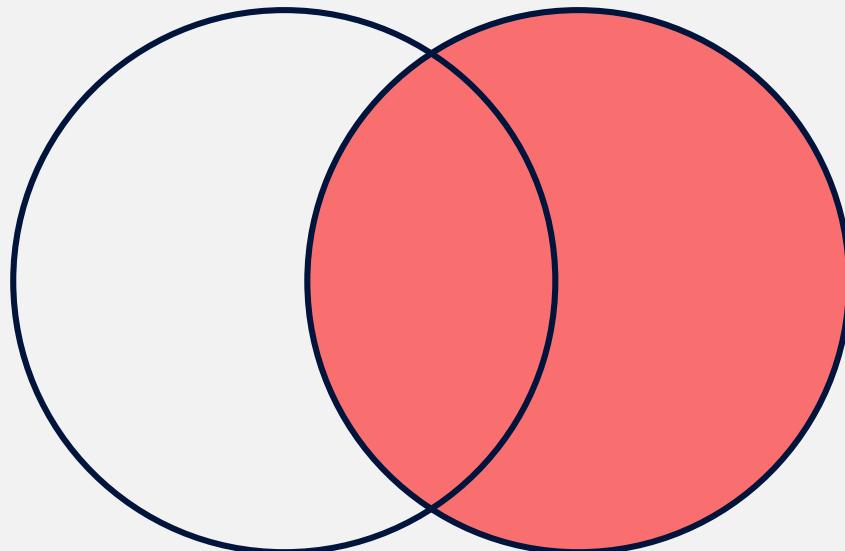
*Yes, we will!*

departments\_dup

```
dept_no CHAR(4)  
dept_name VARCHAR(40)
```

dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL

# RIGHT JOIN



dept\_manager\_dup

```
dept_no CHAR(4)  
emp_no INT  
from_date DATE  
to_date DATE
```

emp_no	dept_no	from_date	to_date
999904	NULL	2017-01-01	NULL
999905	NULL	2017-01-01	NULL
999906	NULL	2017-01-01	NULL
999907	NULL	2017-01-01	NULL
110085	d002	1985-01-01	1989-12-17
110114	d002	1989-12-17	9999-01-01
110183	d003	1985-01-01	1992-03-21
110228	d003	1992-03-21	9999-01-01
110303	d004	1985-01-01	1988-09-09
110344	d004	1988-09-09	1992-08-02
110386	d004	1992-08-02	1996-08-30
110420	d004	1996-08-30	9999-01-01
110511	d005	1985-01-01	1992-04-25
110567	d005	1992-04-25	9999-01-01
110725	d006	1985-01-01	1989-05-06
110765	d006	1989-05-06	1991-09-12
110800	d006	1991-09-12	1994-06-28
110854	d006	1994-06-28	9999-01-01
111035	d007	1985-01-01	1991-03-07
111133	d007	1991-03-07	9999-01-01
111400	d008	1985-01-01	1991-04-08
111534	d008	1991-04-08	9999-01-01
111692	d009	1985-01-01	1988-10-17
111784	d009	1988-10-17	1992-09-08
111877	d009	1992-09-08	1996-01-03
111939	d009	1996-01-03	9999-01-01

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

emp_no	dept_no	from_date	to_date
999904	NULL	2017-01-01	NULL
999905	NULL	2017-01-01	NULL
999906	NULL	2017-01-01	NULL
999907	NULL	2017-01-01	NULL
110085	d002	1985-01-01	1989-12-17
110114	d002	1989-12-17	9999-01-01
110183	d003	1985-01-01	1992-03-21
110228	d003	1992-03-21	9999-01-01
110303	d004	1985-01-01	1988-09-09
110344	d004	1988-09-09	1992-08-02
110386	d004	1992-08-02	1996-08-30
110420	d004	1996-08-30	9999-01-01
110511	d005	1985-01-01	1992-04-25
110567	d005	1992-04-25	9999-01-01
110725	d006	1985-01-01	1989-05-06
110765	d006	1989-05-06	1991-09-12
110800	d006	1991-09-12	1994-06-28
110854	d006	1994-06-28	9999-01-01
111035	d007	1985-01-01	1991-03-07
111133	d007	1991-03-07	9999-01-01
111400	d008	1985-01-01	1991-04-08
111534	d008	1991-04-08	9999-01-01
111692	d009	1985-01-01	1988-10-17
111784	d009	1988-10-17	1992-09-08
111877	d009	1992-09-08	1996-01-03
111939	d009	1996-01-03	9999-01-01

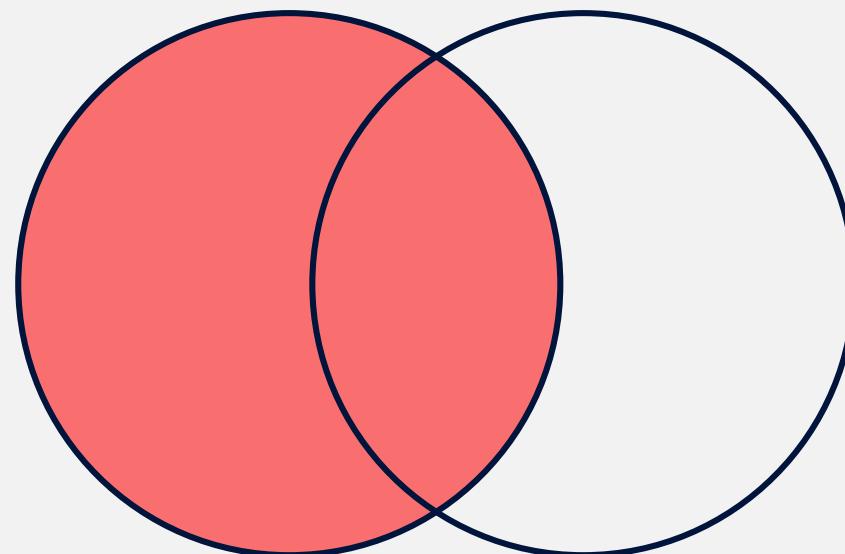
# RIGHT JOIN

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL



# RIGHT JOIN

dept\_manager\_dup

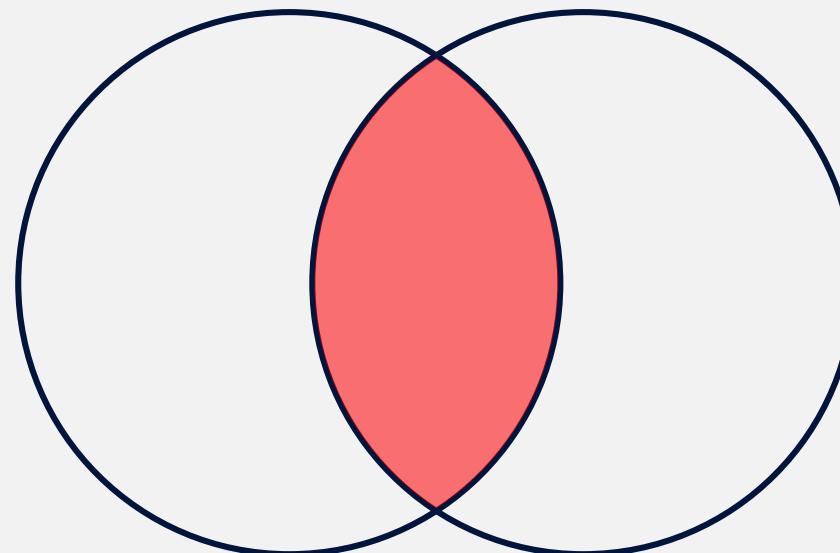
dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

dept\_no CHAR(4)



departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# RIGHT JOIN

dept\_manager\_dup

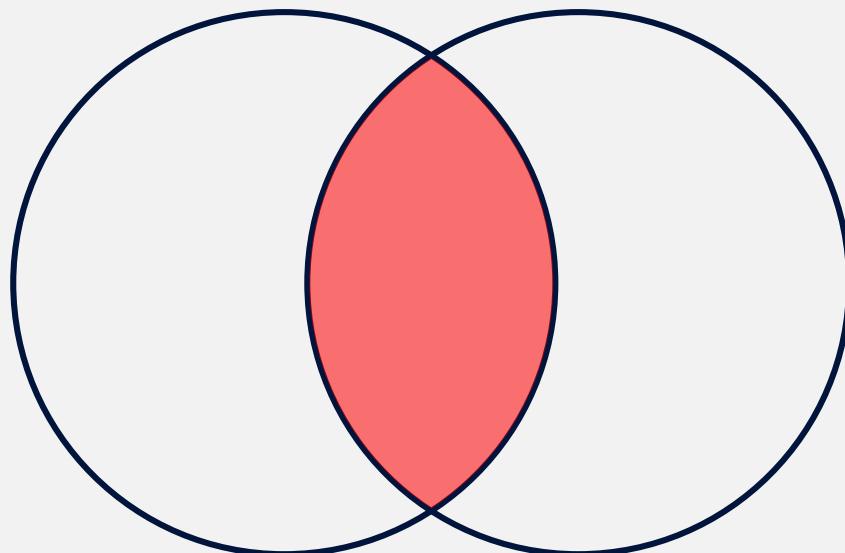
dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

dept\_no CHAR(4)



departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

matching column = linking column

# RIGHT JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

dept\_no CHAR(4)

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

matching column = linking column

# RIGHT JOIN

- **RIGHT JOIN**

# RIGHT JOIN

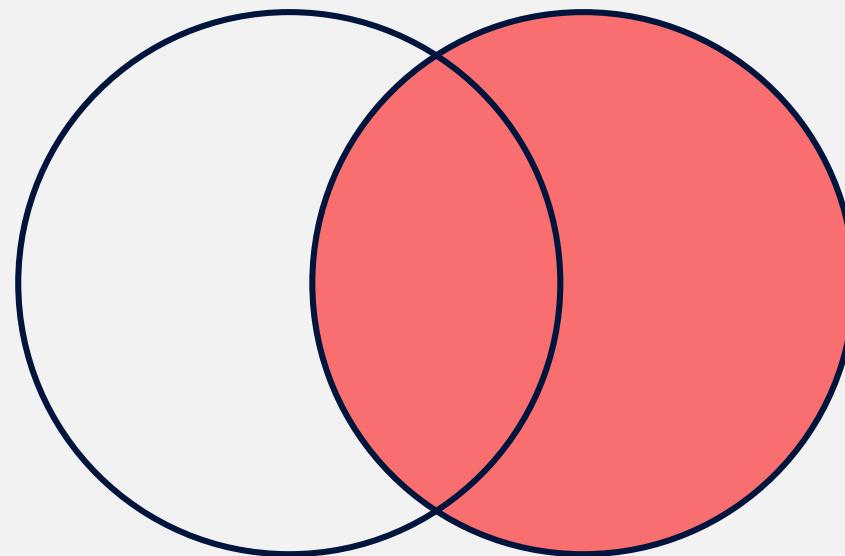
- **RIGHT JOIN**

when applying a **RIGHT JOIN**, all the records from the right table will be included in the result set

# RIGHT JOIN

- **RIGHT JOIN**

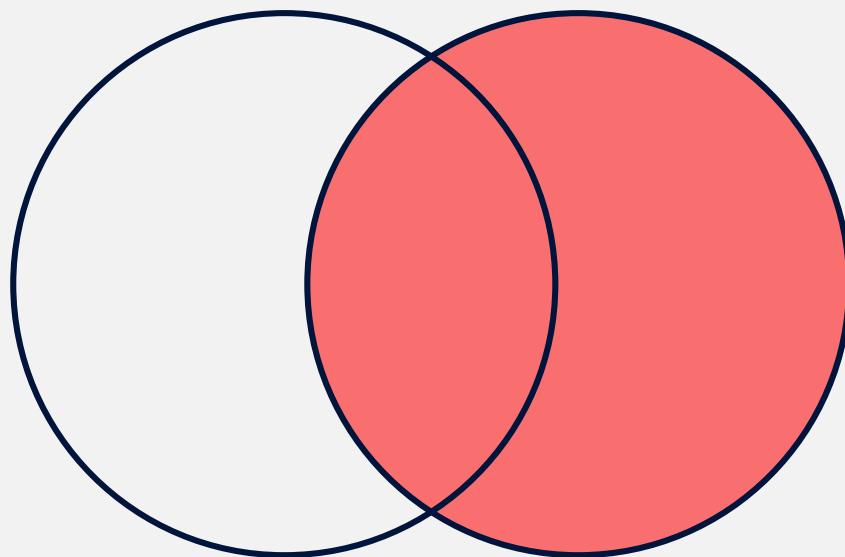
when applying a **RIGHT JOIN**, all the records from the right table will be included in the result set



# RIGHT JOIN

- **RIGHT JOIN**

when applying a **RIGHT JOIN**, all the records from the right table will be included in the result set

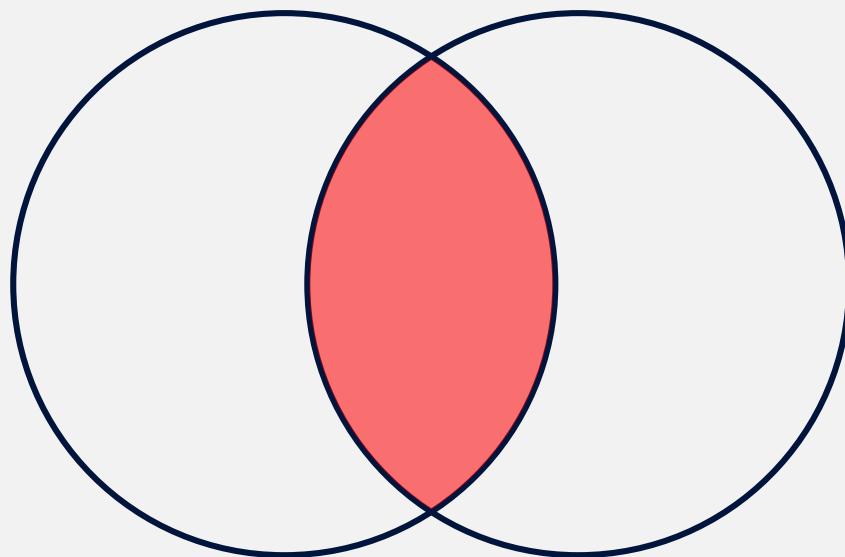


values from the left table will be included only if their *linking column* contains a value coinciding, or *matching*, with a value from the *linking column* of the right table

# RIGHT JOIN

- **RIGHT JOIN**

when applying a **RIGHT JOIN**, all the records from the right table will be included in the result set



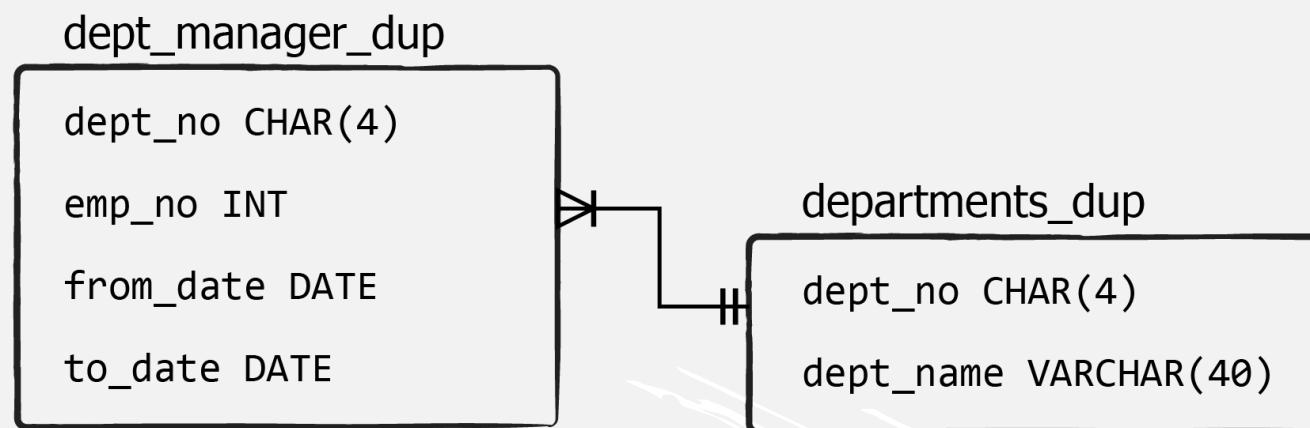
values from the left table will be included only if their *linking column* contains a value coinciding, or *matching*, with a value from the *linking column* of the right table

## RIGHT JOIN

- LEFT and RIGHT joins are perfect examples of one-to-many relationships

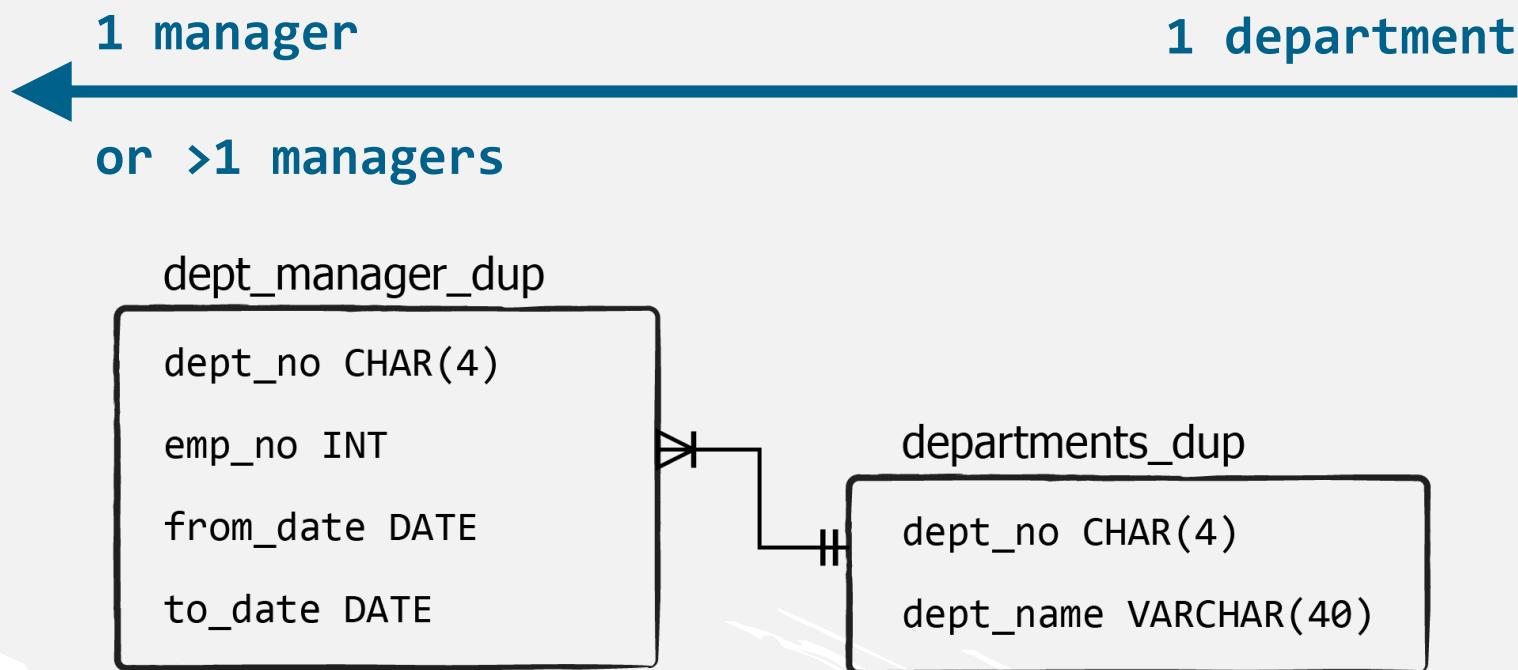
# RIGHT JOIN

- LEFT and RIGHT joins are perfect examples of one-to-many relationships



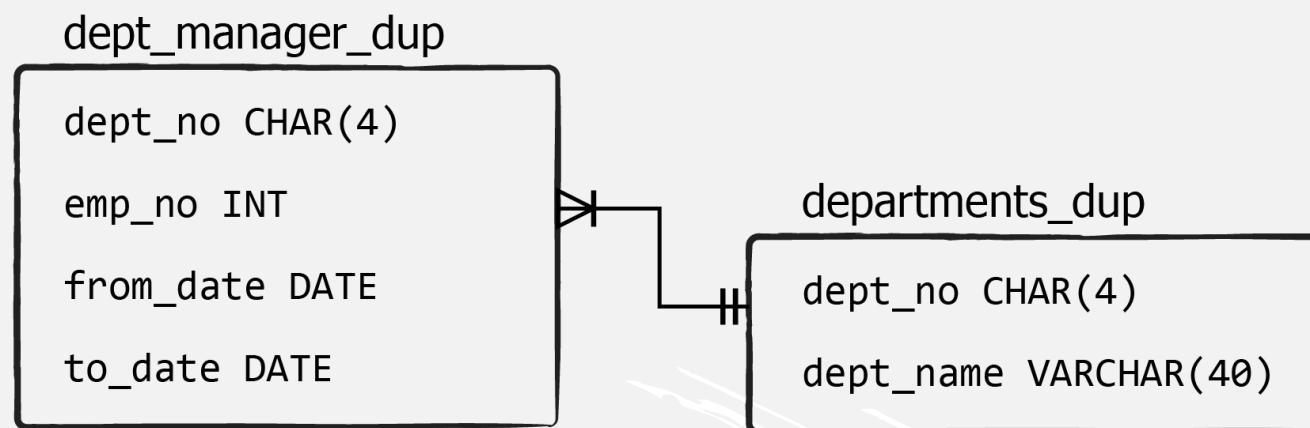
# RIGHT JOIN

- LEFT and RIGHT joins are perfect examples of one-to-many relationships



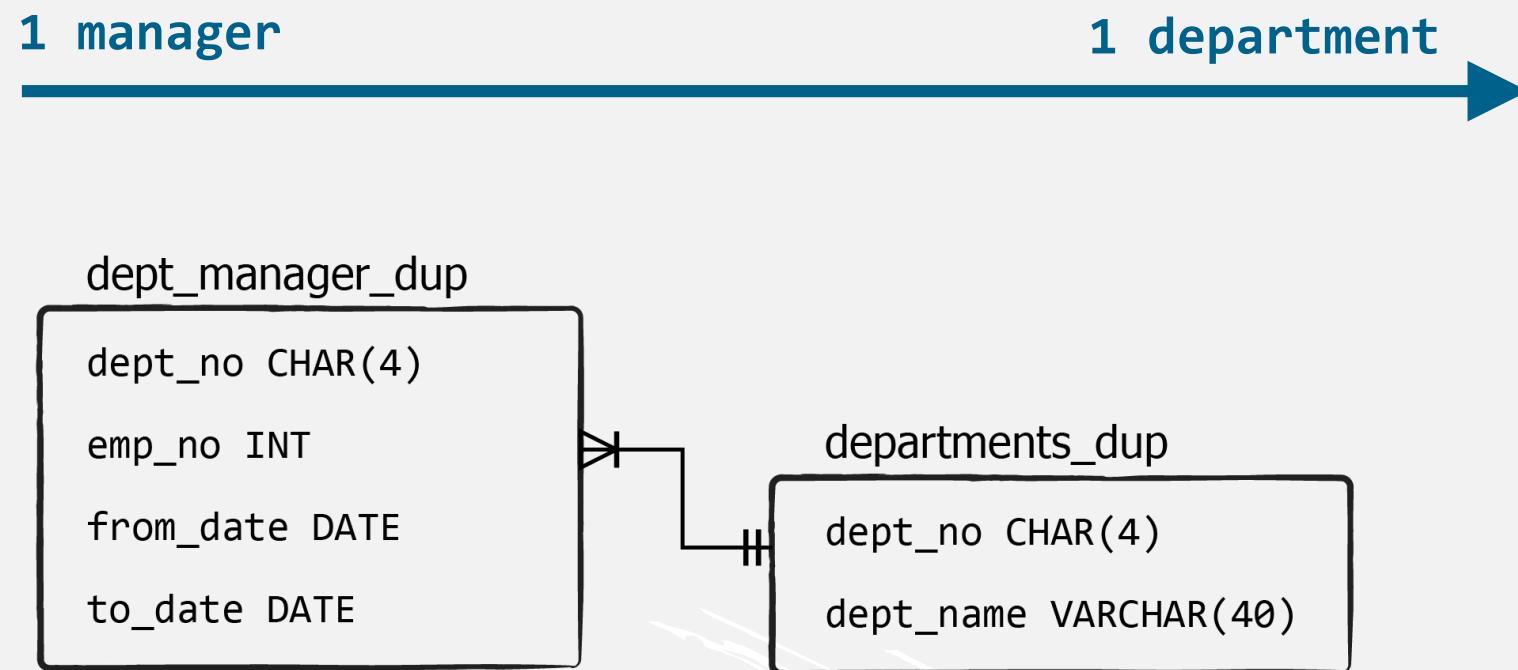
# RIGHT JOIN

- LEFT and RIGHT joins are perfect examples of one-to-many relationships



# RIGHT JOIN

- LEFT and RIGHT joins are perfect examples of one-to-many relationships



A large, modern conference room is shown from a perspective looking down the length of the room. On the left, there are several rows of modern office chairs facing towards the right. The room has a long, dark rectangular table in the center. At the far end of the room, there is a wall with three large, vertical projection screens. The ceiling is white with a grid of recessed lighting. The overall atmosphere is professional and spacious.

# The New and the Old Join Syntax

# The New and the Old Join Syntax

- Relational Database Essentials
- MySQL Constraints
- SELECT, INSERT, UPDATE, DELETE
- MySQL Aggregate Functions
- INNER JOIN, LEFT JOIN, RIGHT JOIN

# The New and the Old Join Syntax

Next:

- Relational Database Essentials
- MySQL Constraints
- SELECT, INSERT, UPDATE, DELETE
- MySQL Aggregate Functions
- INNER JOIN, LEFT JOIN, RIGHT JOIN

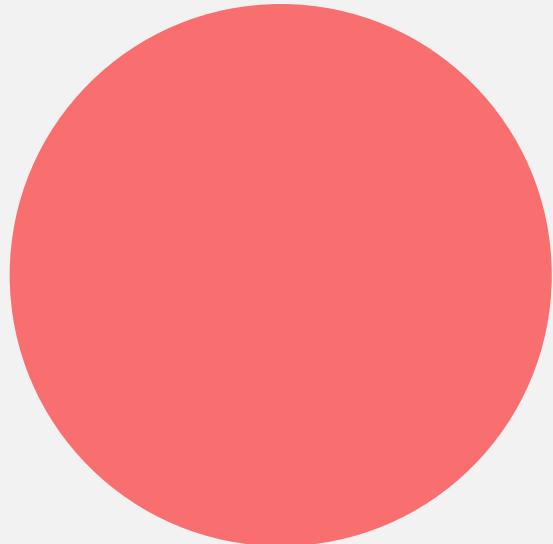
# The New and the Old Join Syntax

- Relational Database Essentials
- MySQL Constraints
- SELECT, INSERT, UPDATE, DELETE
- MySQL Aggregate Functions
- INNER JOIN, LEFT JOIN, RIGHT JOIN

## Next:

- Subqueries, Self-joins
- Stored Routines
- Advanced SQL Tools

# The New and the Old Join Syntax



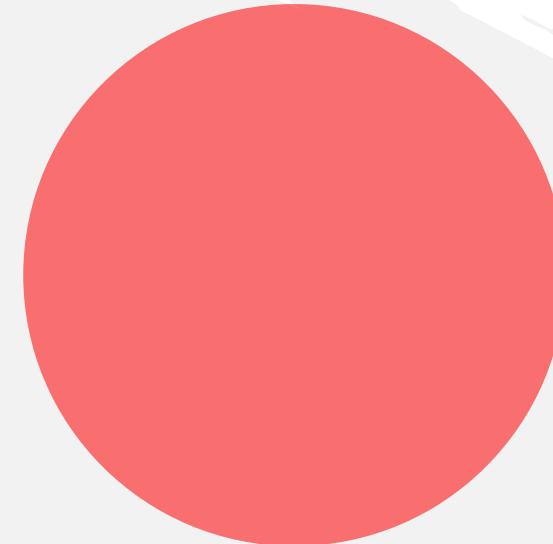
dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE



departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# The New and the Old Join Syntax

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

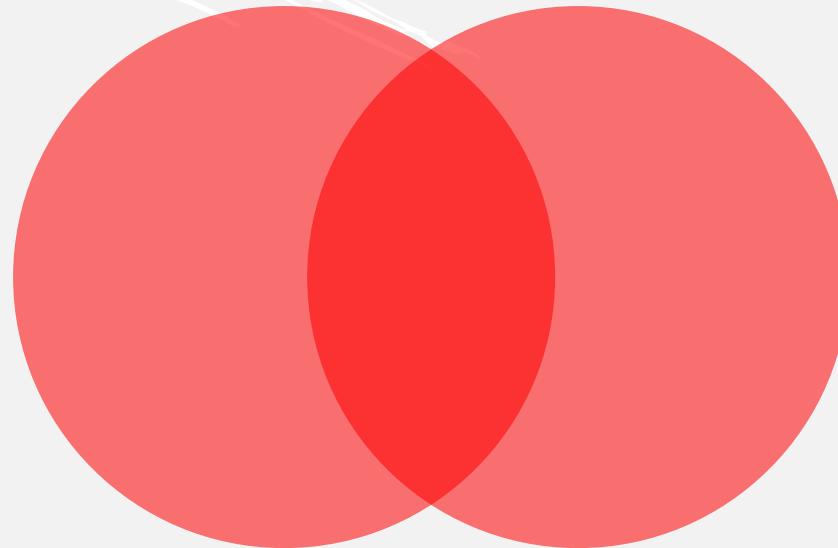
to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# The New and the Old Join Syntax



dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

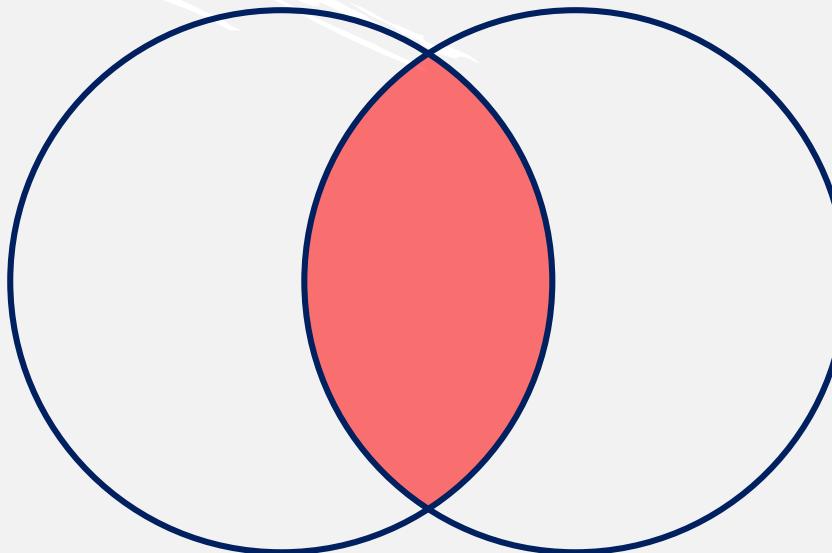
to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# The New and the Old Join Syntax



dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

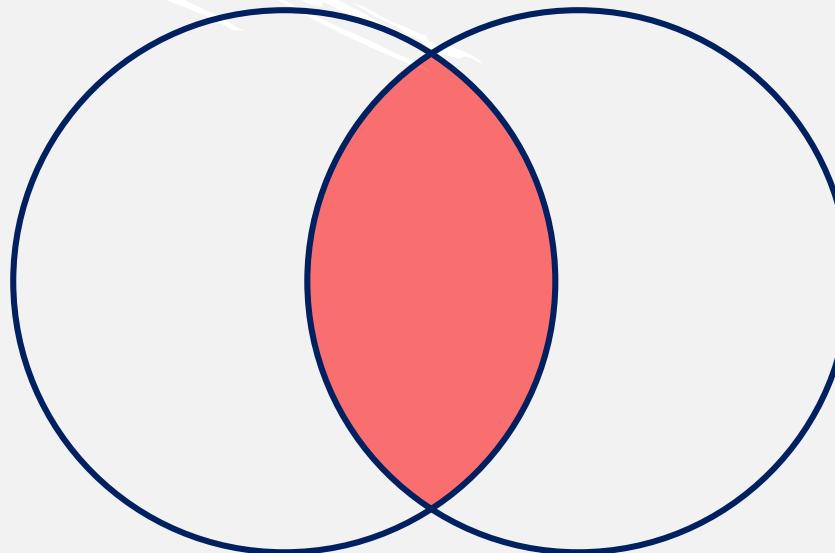
to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# The New and the Old Join Syntax



dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

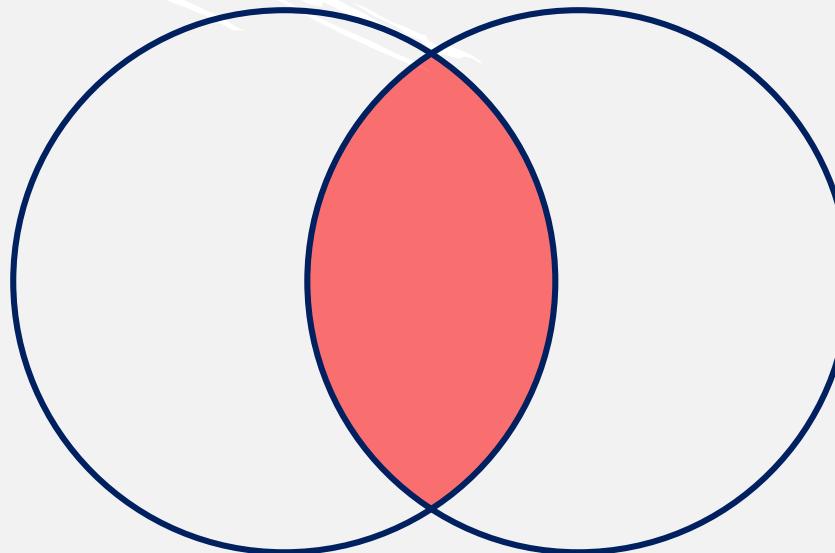
to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# The New and the Old Join Syntax



dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

connection points

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

# The New and the Old Join Syntax

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

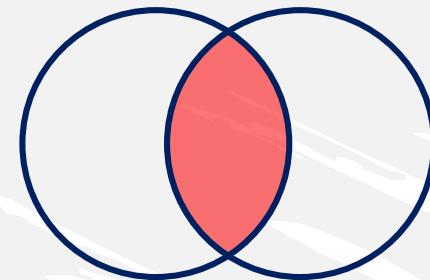
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



connection points

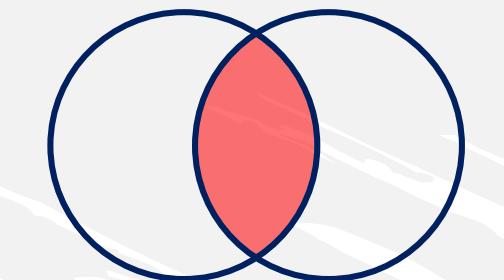
# The New and the Old Join Syntax

emp_no	dept_no	from_date	to_date
999904	NULL	2017-01-01	NULL
999905	NULL	2017-01-01	NULL
999906	NULL	2017-01-01	NULL
999907	NULL	2017-01-01	NULL
110085	d002	1985-01-01	1989-12-17
110114	d002	1989-12-17	9999-01-01
110183	d003	1985-01-01	1992-03-21
110228	d003	1992-03-21	9999-01-01
110303	d004	1985-01-01	1988-09-09
110344	d004	1988-09-09	1992-08-02
110386	d004	1992-08-02	1996-08-30
110420	d004	1996-08-30	9999-01-01
110511	d005	1985-01-01	1992-04-25
110567	d005	1992-04-25	9999-01-01
110725	d006	1985-01-01	1989-05-06
110765	d006	1989-05-06	1991-09-12
110800	d006	1991-09-12	1994-06-28
110854	d006	1994-06-28	9999-01-01
111035	d007	1985-01-01	1991-03-07
111133	d007	1991-03-07	9999-01-01
111400	d008	1985-01-01	1991-04-08
111534	d008	1991-04-08	9999-01-01
111692	d009	1985-01-01	1988-10-17
111784	d009	1988-10-17	1992-09-08
111877	d009	1992-09-08	1996-01-03
111939	d009	1996-01-03	9999-01-01

dept\_manager\_dup

```
dept_no CHAR(4)
emp_no INT
from_date DATE
to_date DATE
```

dept_no	emp_no	dept_name
d003	110228	Human Resources
d003	110183	Human Resources
d004	110344	Production
d004	110420	Production
d004	110303	Production
d004	110386	Production
d005	110567	Development
d005	110511	Development
d006	110800	Quality Management
d006	110765	Quality Management
d006	110854	Quality Management
d006	110725	Quality Management
d007	111035	Sales
d007	111133	Sales
d008	111400	Research
d008	111534	Research
d009	111784	Customer Service
d009	111939	Customer Service
d009	111692	Customer Service
d009	111877	Customer Service



connection points

dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL

departments\_dup

```
dept_no CHAR(4)
dept_name VARCHAR(40)
```

# The New and the Old Join Syntax

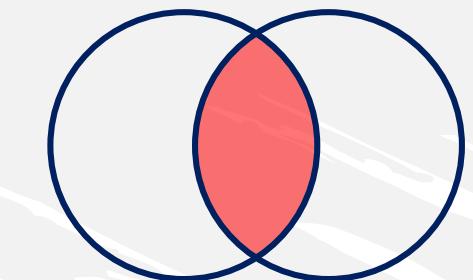
emp_no	dept_no	from_date	to_date
999904	NULL	X	2017-01-01
999905	NULL	X	2017-01-01
999906	NULL	X	2017-01-01
999907	NULL	X	2017-01-01
110085	d002	X	1985-01-01
110114	d002	X	1989-12-17
110183	d003	✓	1985-01-01
110228	d003	✓	1992-03-21
110303	d004	✓	1985-01-01
110344	d004	✓	1988-09-09
110386	d004	✓	1992-08-02
110420	d004	✓	1996-08-30
110511	d005	✓	1985-01-01
110567	d005	✓	1992-04-25
110725	d006	✓	1985-01-01
110765	d006	✓	1989-05-06
110800	d006	✓	1991-09-12
110854	d006	✓	1994-06-28
111035	d007	✓	1985-01-01
111133	d007	✓	1991-03-07
111400	d008	✓	1985-01-01
111534	d008	✓	1991-04-08
111692	d009	✓	1985-01-01
111784	d009	✓	1988-10-17
111877	d009	✓	1992-09-08
111939	d009	✓	1996-01-03

dept\_manager\_dup

```
dept_no CHAR(4)
emp_no INT
from_date DATE
to_date DATE
```

dept_no	emp_no	dept_name
d003	110228	Human Resources
d003	110183	Human Resources
d004	110344	Production
d004	110420	Production
d004	110303	Production
d004	110386	Production
d005	110567	Development
d005	110511	Development
d006	110800	Quality Management
d006	110765	Quality Management
d006	110854	Quality Management
d006	110725	Quality Management
d007	111035	Sales
d007	111133	Sales
d008	111400	Research
d008	111534	Research
d009	111784	Customer Service
d009	111939	Customer Service
d009	111692	Customer Service
d009	111877	Customer Service

dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL



connection points

departments\_dup

```
dept_no CHAR(4)
dept_name VARCHAR(40)
```

# The New and the Old Join Syntax

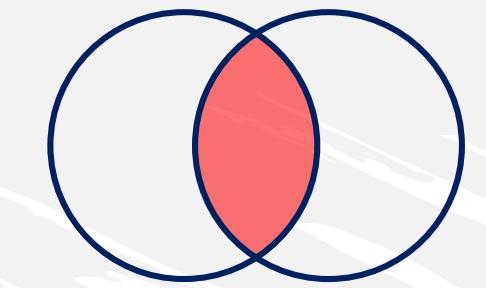
emp_no	dept_no	from_date	to_date
999904	NULL	X	2017-01-01
999905	NULL	X	2017-01-01
999906	NULL	X	2017-01-01
999907	NULL	X	2017-01-01
110085	d002	X	1985-01-01
110114	d002	X	1989-12-17
110183	d003	✓	1985-01-01
110228	d003	✓	1992-03-21
110303	d004	✓	1985-01-01
110344	d004	✓	1988-09-09
110386	d004	✓	1992-08-02
110420	d004	✓	1996-08-30
110511	d005	✓	1985-01-01
110567	d005	✓	1992-04-25
110725	d006	✓	1985-01-01
110765	d006	✓	1989-05-06
110800	d006	✓	1991-09-12
110854	d006	✓	1994-06-28
111035	d007	✓	1985-01-01
111133	d007	✓	1991-03-07
111400	d008	✓	1985-01-01
111534	d008	✓	1991-04-08
111692	d009	✓	1985-01-01
111784	d009	✓	1988-10-17
111877	d009	✓	1992-09-08
111939	d009	✓	1996-01-03

dept\_manager\_dup

```
dept_no CHAR(4)
emp_no INT
from_date DATE
to_date DATE
```

dept_no	emp_no	dept_name
d003	110228	Human Resources
d003	110183	Human Resources
d004	110344	Production
d004	110420	Production
d004	110303	Production
d004	110386	Production
d005	110567	Development
d005	110511	Development
d006	110800	Quality Management
d006	110765	Quality Management
d006	110854	Quality Management
d006	110725	Quality Management
d007	111035	Sales
d007	111133	Sales
d008	111400	Research
d008	111534	Research
d009	111784	Customer Service
d009	111939	Customer Service
d009	111692	Customer Service
d009	111877	Customer Service

dept_no	dept_name
NULL	X Public Relations
d001	X Marketing
d003	✓ Human Resources
d004	✓ Production
d005	✓ Development
d006	✓ Quality Management
d007	✓ Sales
d008	✓ Research
d009	✓ Customer Service
d010	X NULL
d011	X NULL



departments\_dup

```
dept_no CHAR(4)
dept_name VARCHAR(40)
```

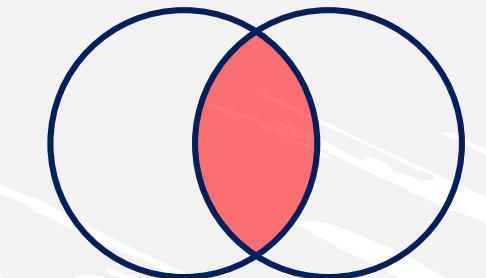
# The New and the Old Join Syntax

emp_no	dept_no	from_date	to_date
999904	NULL	2017-01-01	NULL
999905	NULL	2017-01-01	NULL
999906	NULL	2017-01-01	NULL
999907	NULL	2017-01-01	NULL
110085	d002	1985-01-01	1989-12-17
110114	d002	1989-12-17	9999-01-01
110183	d003	1985-01-01	1992-03-21
110228	d003	1992-03-21	9999-01-01
110303	d004	1985-01-01	1988-09-09
110344	d004	1988-09-09	1992-08-02
110386	d004	1992-08-02	1996-08-30
110420	d004	1996-08-30	9999-01-01
110511	d005	1985-01-01	1992-04-25
110567	d005	1992-04-25	9999-01-01
110725	d006	1985-01-01	1989-05-06
110765	d006	1989-05-06	1991-09-12
110800	d006	1991-09-12	1994-06-28
110854	d006	1994-06-28	9999-01-01
111035	d007	1985-01-01	1991-03-07
111133	d007	1991-03-07	9999-01-01
111400	d008	1985-01-01	1991-04-08
111534	d008	1991-04-08	9999-01-01
111692	d009	1985-01-01	1988-10-17
111784	d009	1988-10-17	1992-09-08
111877	d009	1992-09-08	1996-01-03
111939	d009	1996-01-03	9999-01-01

dept\_manager\_dup

```
dept_no CHAR(4)
emp_no INT
from_date DATE
to_date DATE
```

dept_no	emp_no	dept_name
d003	110228	Human Resources
d003	110183	Human Resources
d004	110344	Production
d004	110420	Production
d004	110303	Production
d004	110386	Production
d005	110567	Development
d005	110511	Development
d006	110800	Quality Management
d006	110765	Quality Management
d006	110854	Quality Management
d006	110725	Quality Management
d007	111035	Sales
d007	111133	Sales
d008	111400	Research
d008	111534	Research
d009	111784	Customer Service
d009	111939	Customer Service
d009	111692	Customer Service
d009	111877	Customer Service



connection points

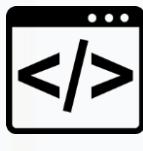
dept_no	dept_name
NULL	Public Relations
d001	Marketing
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	NULL
d011	NULL

departments\_dup

```
dept_no CHAR(4)
dept_name VARCHAR(40)
```

# The New and the Old Join Syntax

- WHERE (*the Old Join Syntax*)



SQL

```
SELECT
    t1.column_name, t1.column_name, ..., t2.column_name, ...
FROM
    table_1 t1,
    table_2 t2
WHERE
    t1.column_name = t2.column_name;
```

# The New and the Old Join Syntax

- JOIN or WHERE?

# The New and the Old Join Syntax

- **JOIN or WHERE?**

- the retrieved output is *identical*

# The New and the Old Join Syntax

- **JOIN or WHERE?**

- the retrieved output is *identical*
- using WHERE is more *time-consuming*

# The New and the Old Join Syntax

- **JOIN or WHERE?**

- the retrieved output is *identical*
- using WHERE is more *time-consuming*
- the WHERE syntax is perceived as *morally old* and is rarely employed by professionals

# The New and the Old Join Syntax

- **JOIN or WHERE?**

- the retrieved output is *identical*
- using WHERE is more *time-consuming*
- the WHERE syntax is perceived as *morally old* and is rarely employed by professionals
- the JOIN syntax allows you to modify the connection between tables easily

# The New and the Old Join Syntax

JOIN (*the New Join Syntax*) vs WHERE (*the Old Join Syntax*)

A large conference room with rows of chairs facing a front wall with monitors. The room has a modern design with a grid ceiling and large windows. The text "JOIN and WHERE Used Together" is overlaid on the image.

**JOIN and WHERE Used Together**

# JOIN and WHERE Used Together

- JOIN + WHERE

# JOIN and WHERE Used Together

- **JOIN + WHERE**

## JOIN:

- used for connecting the “employees” and “salaries” tables

# JOIN and WHERE Used Together

- **JOIN + WHERE**

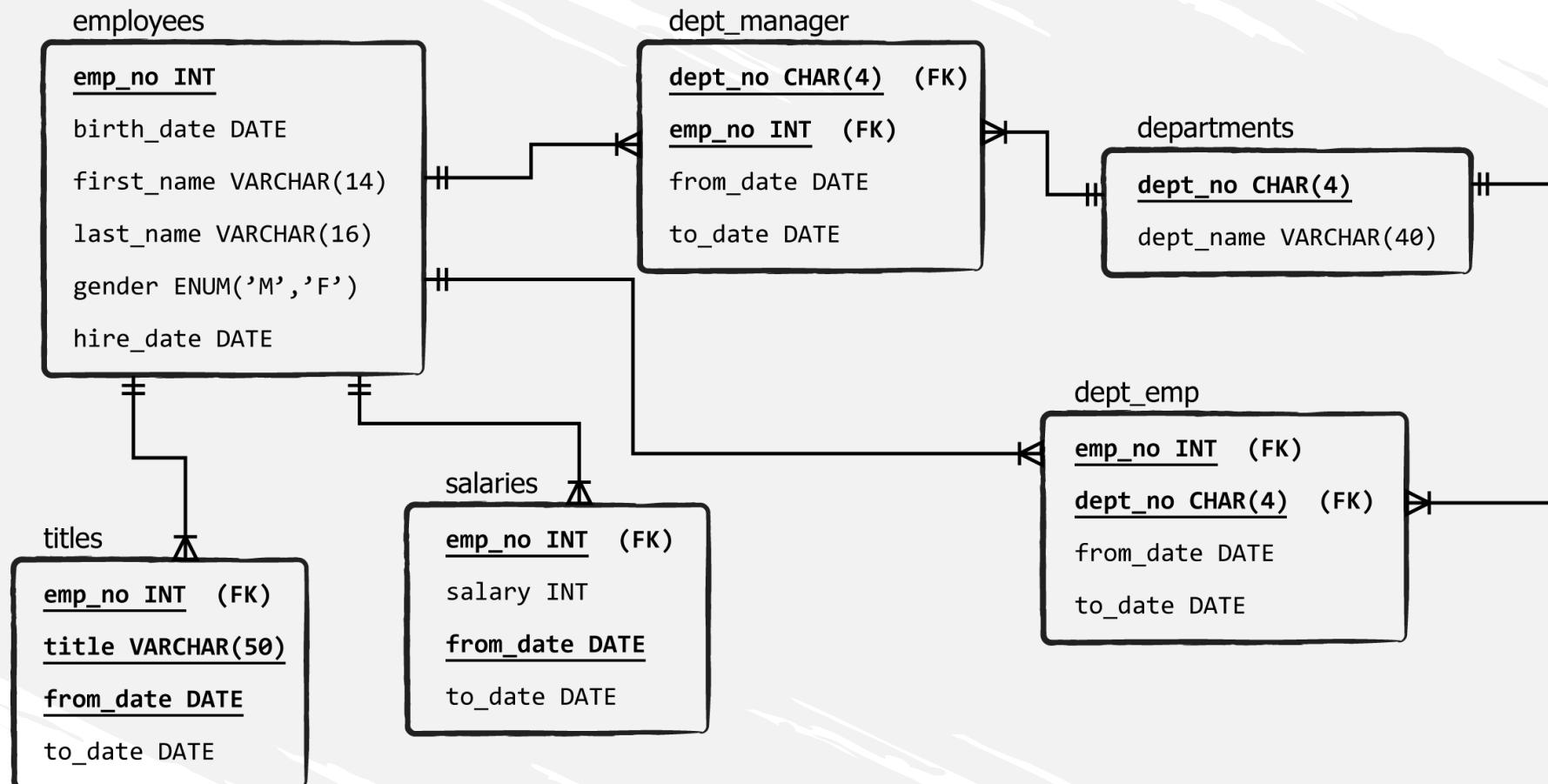
**JOIN:**

- used for connecting the “employees” and “salaries” tables

**WHERE:**

- used to define the condition or conditions that will determine which will be the connecting points between the two tables

# JOIN and WHERE Used Together



**FRAGILE**

**HANDLE WITH CARE**



**JOIN More Than Two Tables in SQL**

# JOIN More Than Two Tables in SQL

- when creating a query that joins multiple tables, you must back it with *strong intuition* and a *crystal-clear idea* of how you would like the tables to be connected

# JOIN More Than Two Tables in SQL

first_name	last_name

# JOIN More Than Two Tables in SQL

first_name	last_name	hire_date

# JOIN More Than Two Tables in SQL

first_name	last_name	hire_date	from_date

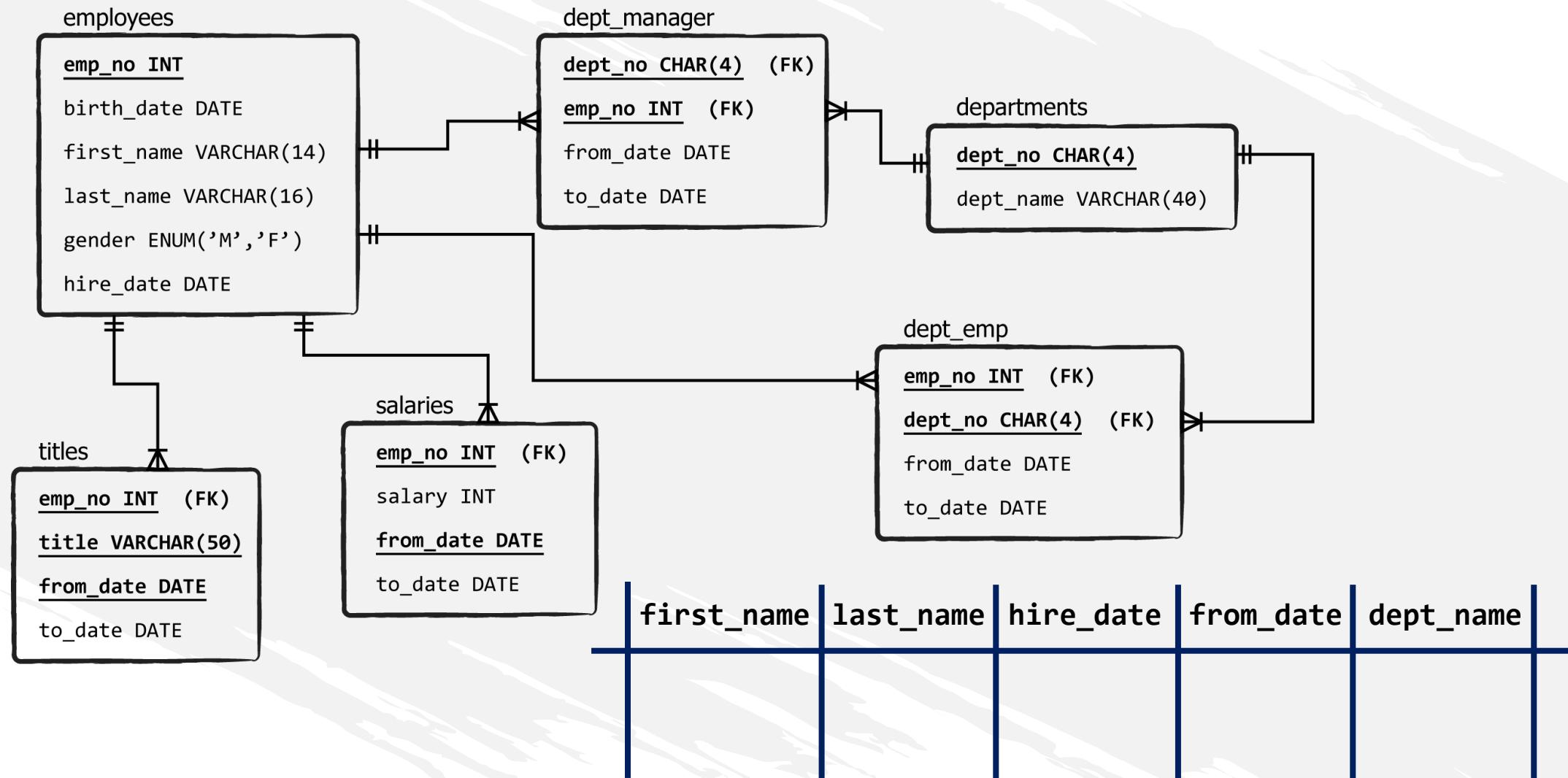
# JOIN More Than Two Tables in SQL

first_name	last_name	hire_date	from_date	dept_name

# JOIN More Than Two Tables in SQL

first_name	last_name	hire_date	from_date	dept_name

# JOIN More Than Two Tables in SQL



A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

# Tips and Tricks for Joins

# Tips and Tricks for Joins

dept_name

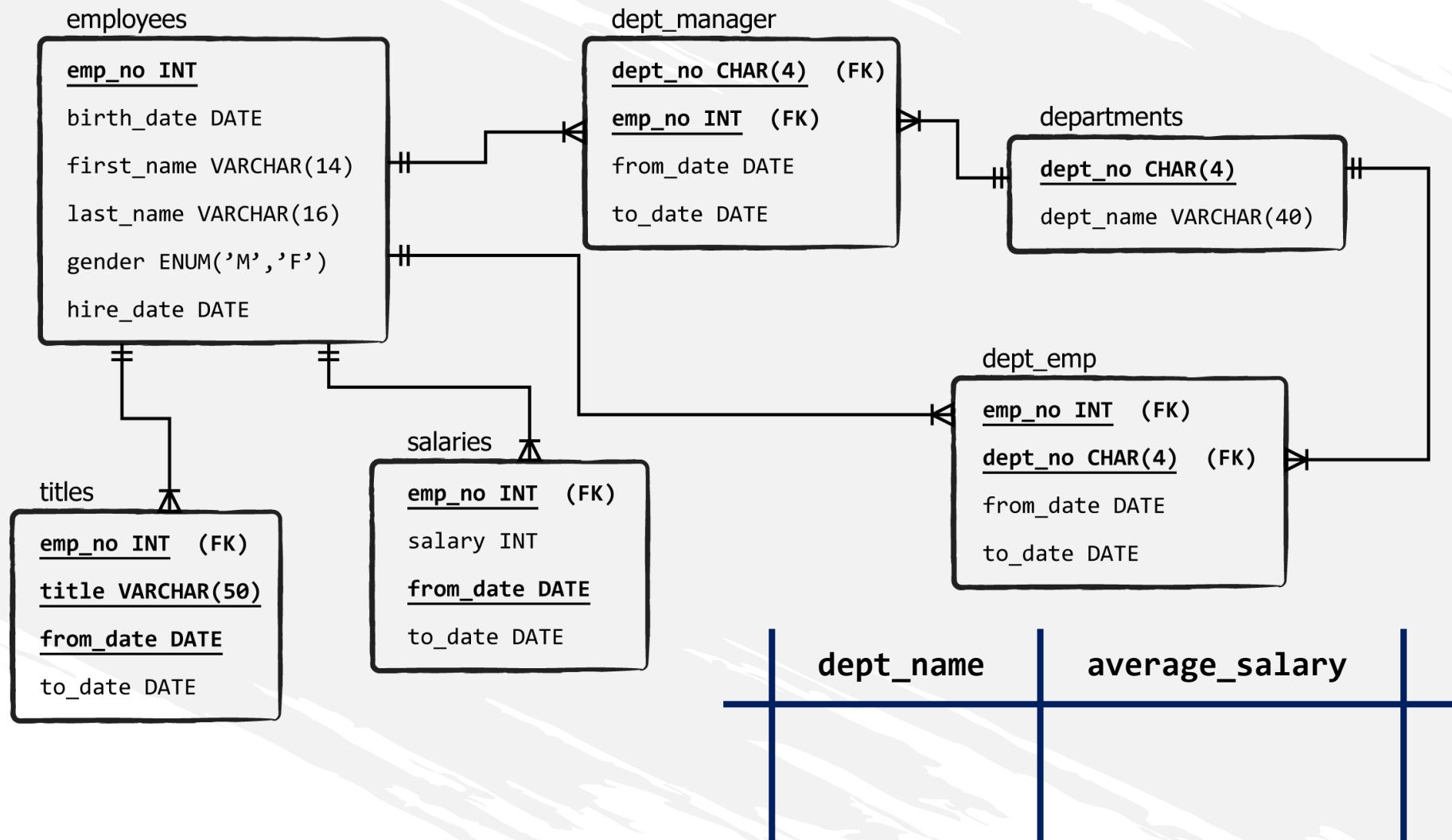
# Tips and Tricks for Joins

dept_name	average_salary

# Tips and Tricks for Joins

dept_name	average_salary

# Tips and Tricks for Joins



# Tips and Tricks for Joins

- JOINS

# Tips and Tricks for Joins

## JOINS

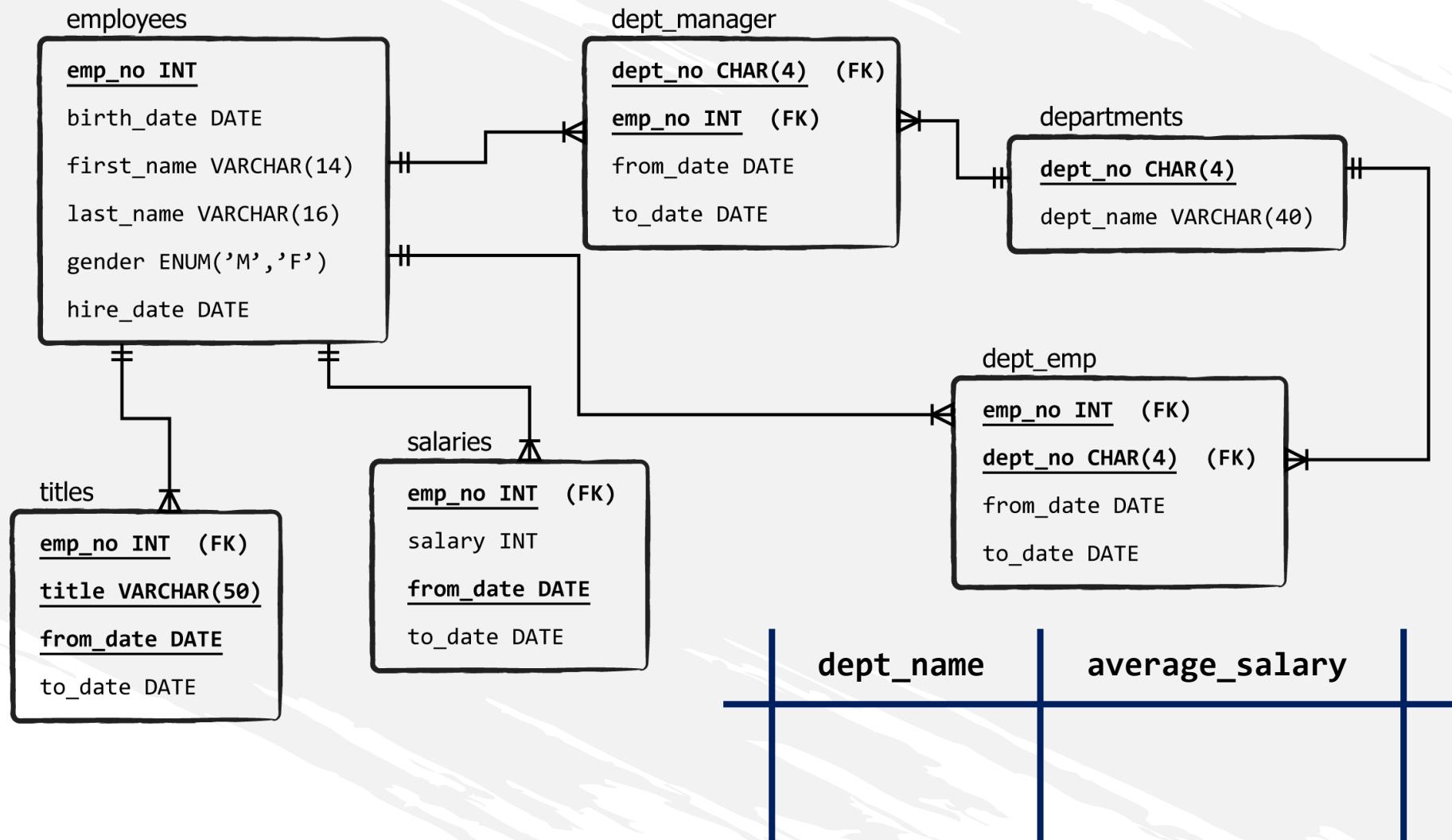
- one should look for key columns, which are common between *the tables involved in the analysis* and are necessary to solve the task at hand

# Tips and Tricks for Joins

## JOINS

- one should look for key columns, which are common between *the tables involved in the analysis* and are necessary to solve the task at hand
- these columns do not need to be foreign or private keys

# Tips and Tricks for Joins



A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

# UNION vs UNION ALL

# UNION vs UNION ALL

- UNION ALL

## UNION vs UNION ALL

- **UNION ALL**

used to combine a few SELECT statements in a single output

# UNION vs UNION ALL

- **UNION ALL**

used to combine a few SELECT statements in a single output

- you can think of it as a tool that allows you to *unify tables*

# UNION vs UNION ALL

- **UNION ALL**

used to combine a few SELECT statements in a single output

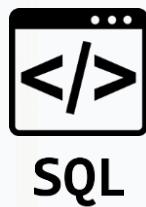


```
SELECT  
      N columns  
FROM  
      table_1  
UNION ALL SELECT  
      N columns  
FROM  
      table_2;
```

# UNION vs UNION ALL

- **UNION ALL**

used to combine a few SELECT statements in a single output



```
SELECT  
      N columns  
FROM  
      table_1  
UNION ALL SELECT  
      N columns  
FROM  
      table_2;
```

*We have to select the same number of columns from each table.*

# UNION vs UNION ALL

- **UNION ALL**

used to combine a few SELECT statements in a single output



```
SELECT  
      N columns  
FROM  
      table_1  
UNION ALL SELECT  
      N columns  
FROM  
      table_2;
```

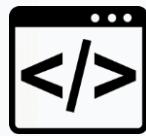
*We have to select the same number of columns from each table.*

*These columns should have the same name,*

# UNION vs UNION ALL

## UNION ALL

used to combine a few SELECT statements in a single output



```
SELECT  
      N columns  
FROM  
      table_1  
UNION ALL SELECT  
      N columns  
FROM  
      table_2;
```

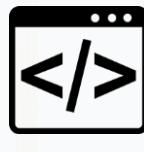
*We have to select the same number of columns from each table.*

*These columns should have the same name,  
should be in the same order,*

# UNION vs UNION ALL

## UNION ALL

used to combine a few SELECT statements in a single output

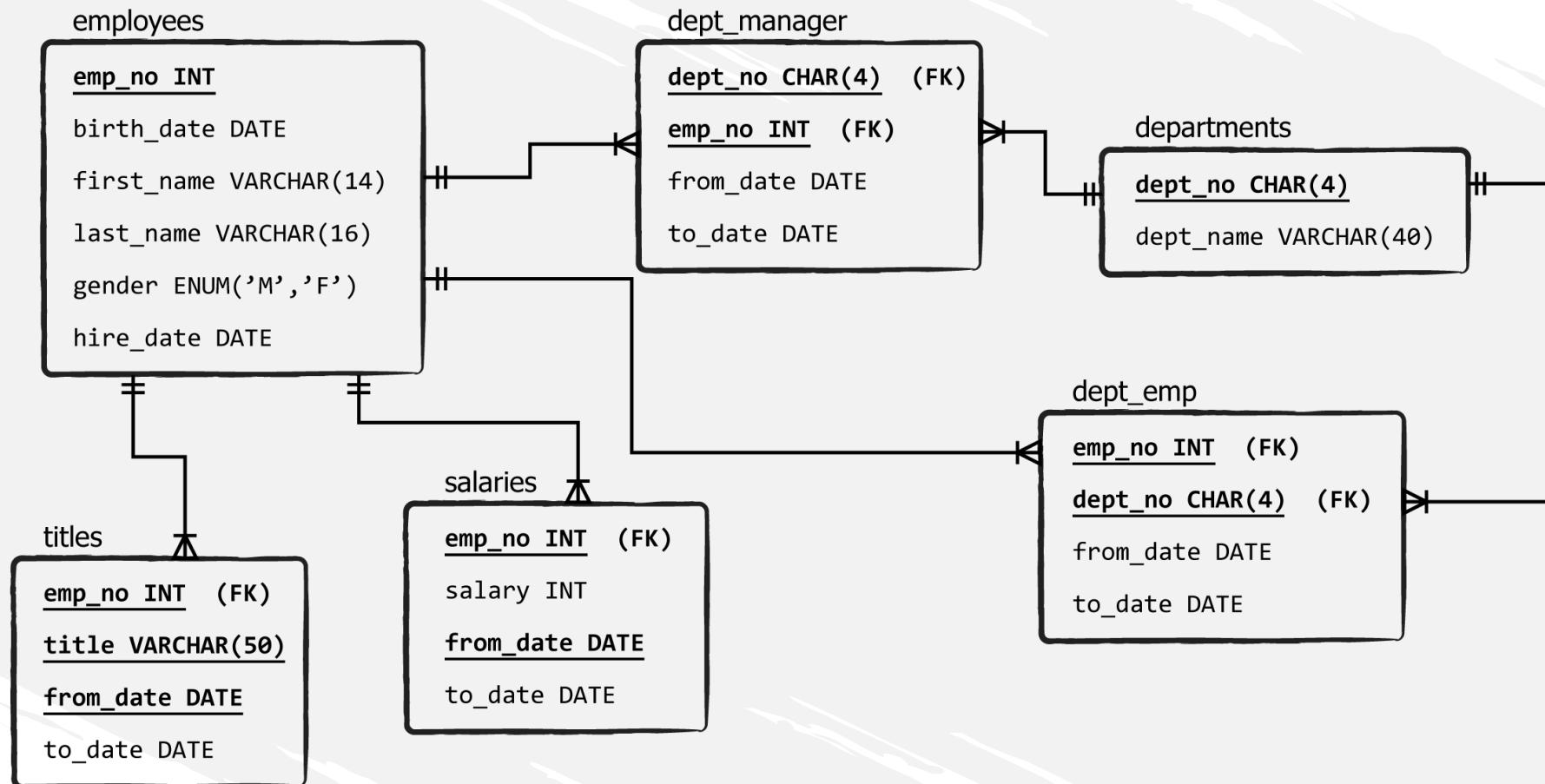


```
SELECT  
      N columns  
FROM  
      table_1  
UNION ALL SELECT  
      N columns  
FROM  
      table_2;
```

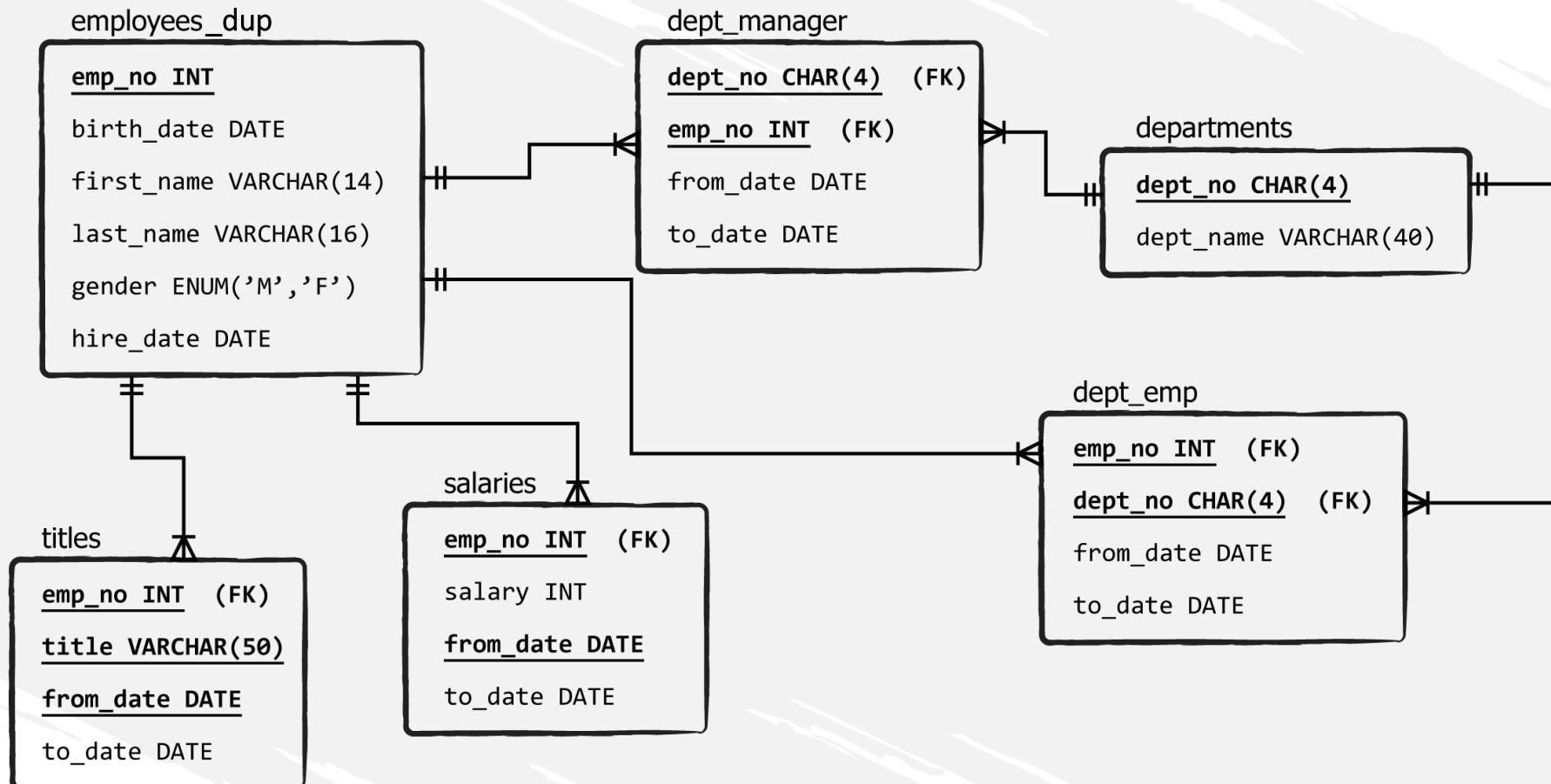
*We have to select the same number of columns from each table.*

*These columns should have the same name, should be in the same order, and should contain related data types.*

# Tips and Tricks for Joins



# Tips and Tricks for Joins



# UNION vs UNION ALL

## UNION



SQL

```
SELECT  
      N columns  
FROM  
      table_1  
UNION SELECT  
      N columns  
FROM  
      table_2;
```

## UNION vs UNION ALL

- when uniting two identically organized tables

## UNION vs UNION ALL

- when uniting two identically organized tables
  - UNION displays only distinct values in the output

## UNION vs UNION ALL

- when uniting two identically organized tables
  - UNION displays only distinct values in the output
  - UNION ALL retrieves the duplicates as well

## UNION vs UNION ALL

- when uniting two identically organized tables
  - UNION displays only distinct values in the output
  - UNION ALL retrieves the duplicates as well

## UNION vs UNION ALL

- when uniting two identically organized tables

- UNION displays only distinct values in the output
  - UNION uses more MySQL resources
- UNION ALL retrieves the duplicates as well

## UNION vs UNION ALL

- when uniting two identically organized tables

- UNION displays only distinct values in the output
  - UNION uses more MySQL resources (computational power and storage space)
- UNION ALL retrieves the duplicates as well

## UNION vs UNION ALL

- Looking for better results?

# UNION vs UNION ALL

- Looking for better results?

- use UNION

# UNION vs UNION ALL

- Looking for better results?
  - use UNION
- Seeking to optimize performance?

# UNION vs UNION ALL

- Looking for better results?
  - use UNION
- Seeking to optimize performance?
  - opt for UNION ALL



# SQL Subqueries



# SQL Subqueries with IN Nested Inside WHERE

# SQL Subqueries with IN Nested Inside WHERE

**subqueries**

queries embedded in a query

# SQL Subqueries with IN Nested Inside WHERE

- subqueries

# SQL Subqueries with IN Nested Inside WHERE

- **subqueries**

queries embedded in a query

# SQL Subqueries with IN Nested Inside WHERE

- subqueries = inner queries

queries embedded in a query

# SQL Subqueries with IN Nested Inside WHERE

- subqueries = inner queries = nested queries

queries embedded in a query

# SQL Subqueries with IN Nested Inside WHERE

- subqueries = inner queries = nested queries

queries embedded in a query

- they are part of *another* query, called an outer query

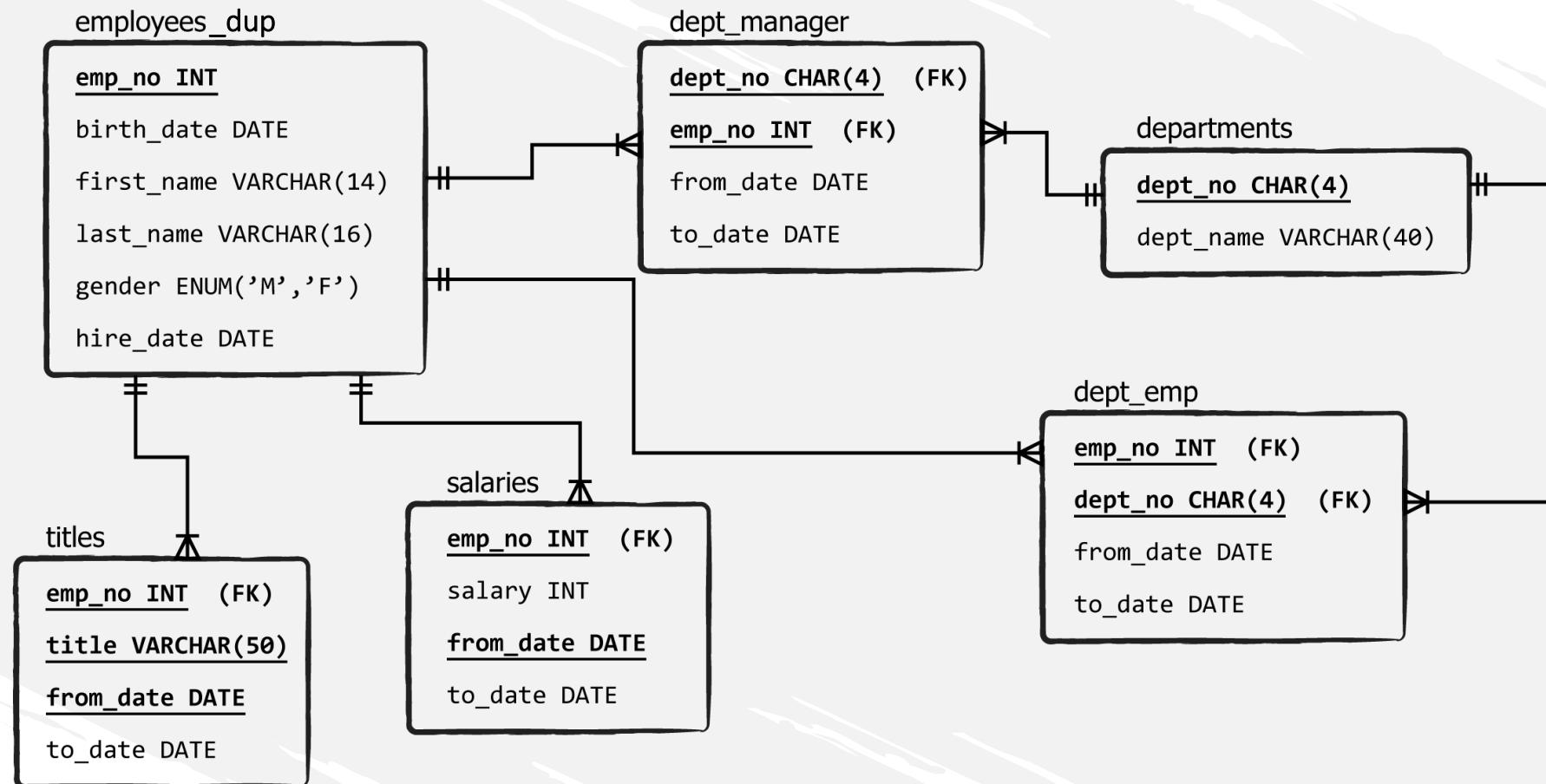
# SQL Subqueries with IN Nested Inside WHERE

- subqueries = inner queries = nested queries = inner select  
queries embedded in a query
- they are part of *another* query, called an outer query

# SQL Subqueries with IN Nested Inside WHERE

- subqueries = inner queries = nested queries = inner select  
queries embedded in a query
  - they are part of *another* query, called an outer query  
= outer select

# SQL Subqueries with IN Nested Inside WHERE



# SQL Subqueries with IN Nested Inside WHERE

- subqueries

# SQL Subqueries with IN Nested Inside WHERE

- **subqueries**

- a subquery should *always* be placed within parentheses

# SQL Subqueries with IN Nested Inside WHERE

- 1) the SQL engine starts by running the *inner query*

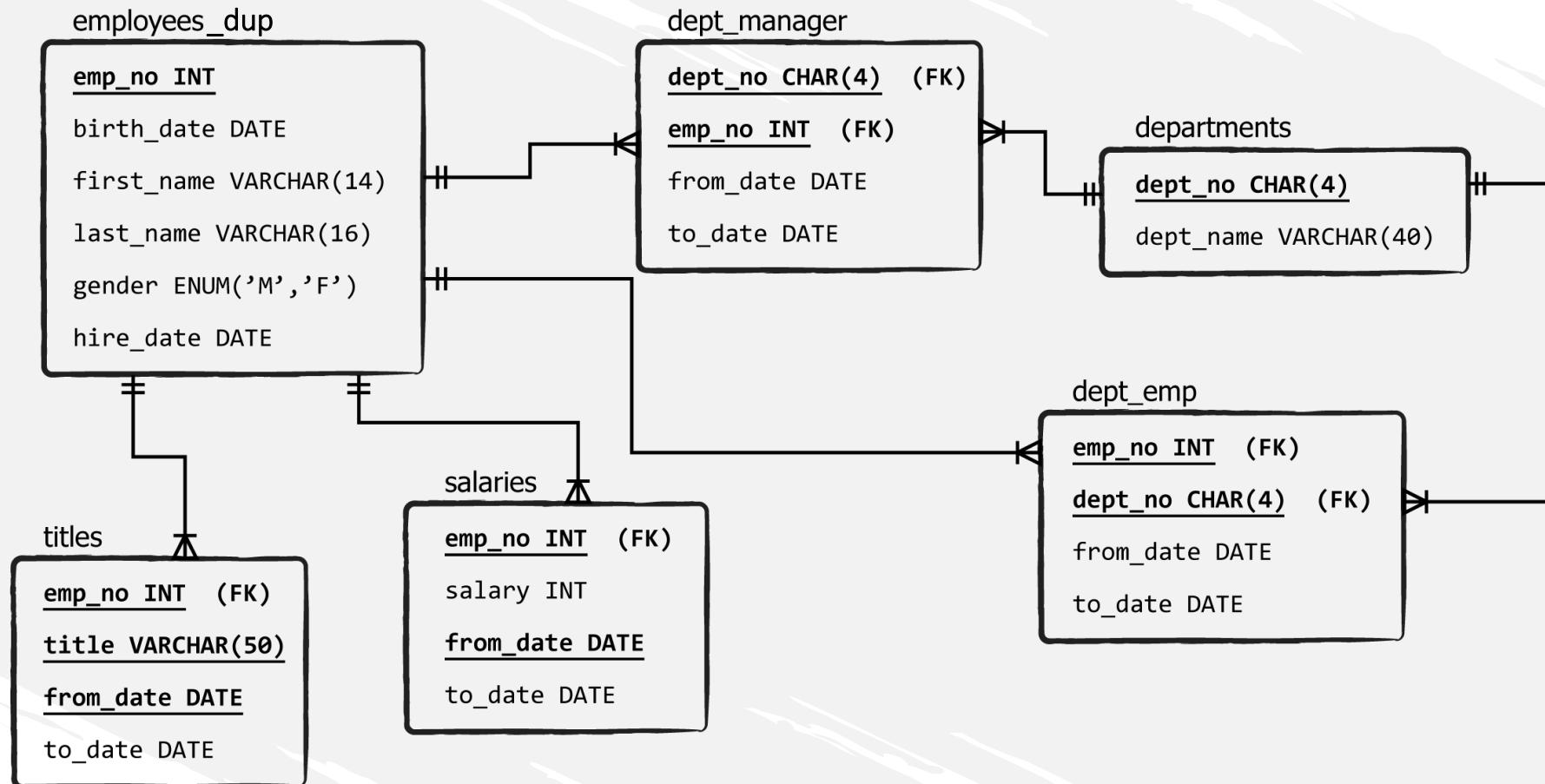
# SQL Subqueries with IN Nested Inside WHERE

- 1) the SQL engine starts by running the *inner query*
- 2) then it uses its returned output, which is intermediate, to execute the *outer query*

# SQL Subqueries with IN Nested Inside WHERE

- a subquery may return *a single value (a scalar), a single row, a single column, or an entire table*

# SQL Subqueries with IN Nested Inside WHERE



# SQL Subqueries with IN Nested Inside WHERE

- a subquery may return *a single value (a scalar), a single row, a single column, or an entire table*

# SQL Subqueries with IN Nested Inside WHERE

- a subquery may return *a single value (a scalar), a single row, a single column, or an entire table*
  - you can have a lot more than one subquery in your outer query

# SQL Subqueries with IN Nested Inside WHERE

- a subquery may return *a single value (a scalar), a single row, a single column, or an entire table*
  - you can have a lot more than one subquery in your outer query
  - it is possible to nest *inner queries within other inner queries*

# SQL Subqueries with IN Nested Inside WHERE

- a subquery may return *a single value (a scalar), a single row, a single column, or an entire table*
  - you can have a lot more than one subquery in your outer query
  - it is possible to nest *inner queries within other inner queries*  
*in that case, the SQL engine would execute the innermost query first*

# SQL Subqueries with IN Nested Inside WHERE

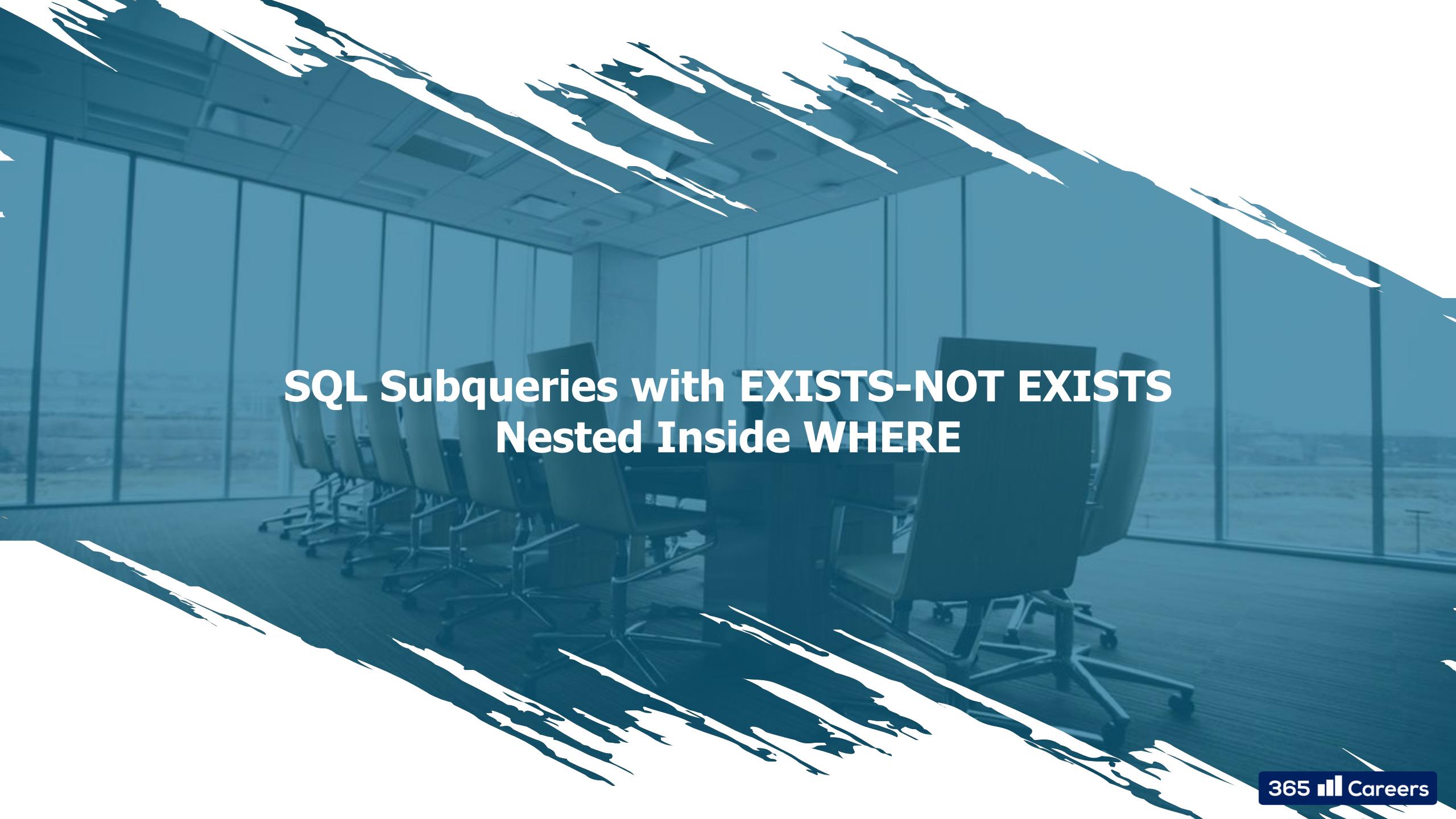
- a subquery may return *a single value (a scalar), a single row, a single column, or an entire table*
  - you can have a lot more than one subquery in your outer query
  - it is possible to nest *inner queries within other inner queries*  
*in that case, the SQL engine would execute the innermost query first, and then each subsequent query*

# SQL Subqueries with IN Nested Inside WHERE

- a subquery may return *a single value (a scalar), a single row, a single column, or an entire table*

- you can have a lot more than one subquery in your outer query
- it is possible to nest *inner queries within other inner queries*

in that case, the SQL engine would execute the *innermost query first*, and then *each subsequent query*, until it runs the *outermost query last*



# **SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE**

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- EXISTS

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
- it returns a Boolean value

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
  - it returns a Boolean value
-

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
- it returns a Boolean value

---

if a row value of a subquery **exists**

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
- it returns a Boolean value

---

if a row value of a subquery **exists** → **TRUE**

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
- it returns a Boolean value

---

if a row value of a subquery **exists** → **TRUE** → *the corresponding record of the outer query is extracted*

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
- it returns a Boolean value

---

if a row value of a subquery **exists** → **TRUE** → *the corresponding record of the outer query is extracted*

if a row value of a subquery  
**doesn't exist**

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
- it returns a Boolean value

---

if a row value of a subquery **exists** → **TRUE** → *the corresponding record of the outer query is extracted*

if a row value of a subquery  
**doesn't exist** → **FALSE**

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **EXISTS**

checks whether certain row values are found within a subquery

- this check is conducted *row by row*
- it returns a Boolean value

---

if a row value of a subquery **exists** → **TRUE** → *the corresponding record of the outer query is extracted*

if a row value of a subquery **doesn't exist** → **FALSE** → *no row value from the outer query is extracted*

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

EXISTS

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

EXISTS

IN

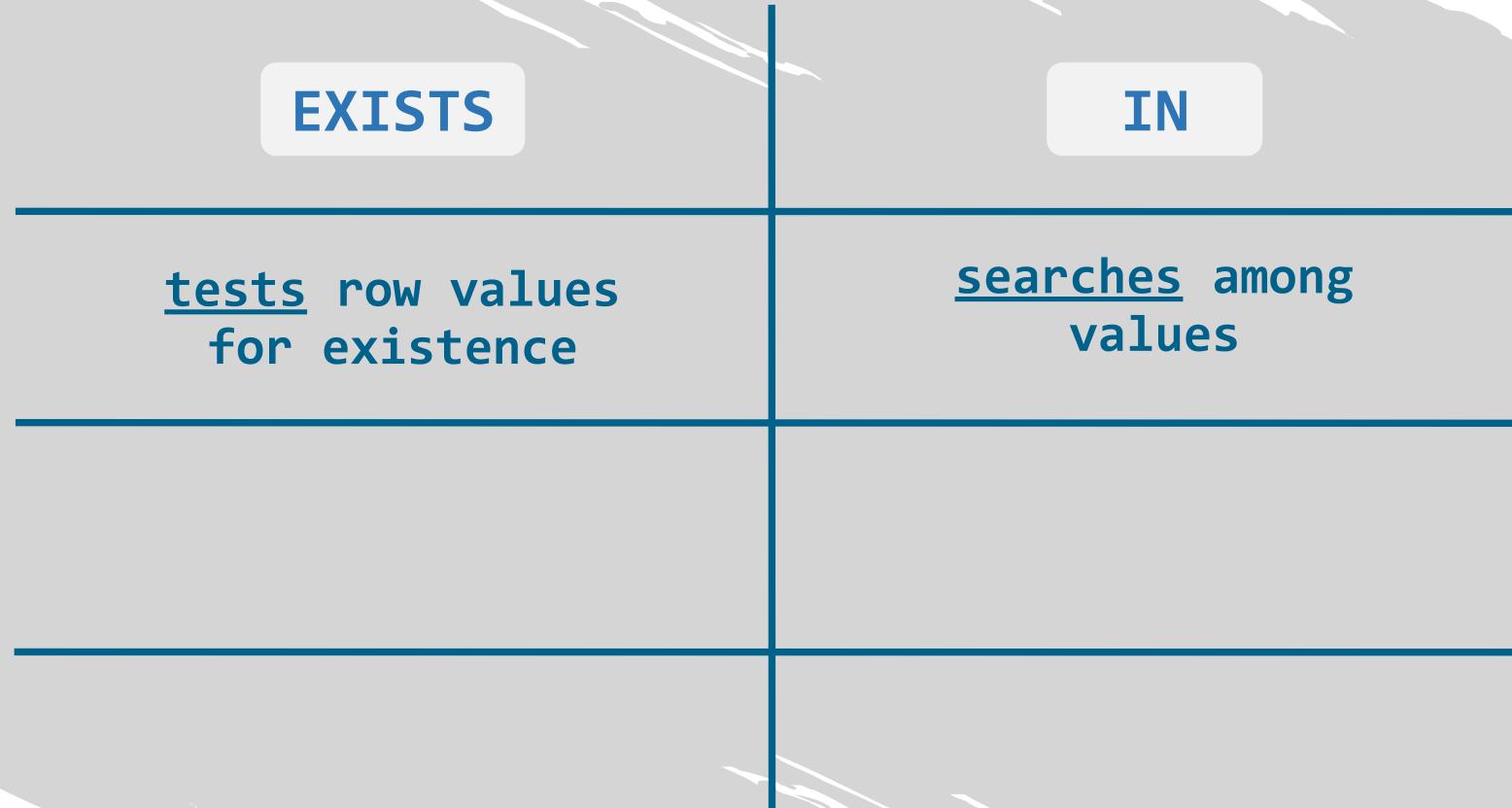
# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

EXISTS

IN

tests row values  
for existence

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE



# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

EXISTS

IN

tests row values  
for existence

searches among  
values

quicker in retrieving  
large amounts of data

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

EXISTS

IN

tests row values  
for existence

searches among  
values

quicker in retrieving  
large amounts of data

faster with  
smaller datasets

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- ORDER BY (nested queries)

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **ORDER BY** (nested queries)

it is more professional to apply **ORDER BY** in the outer query

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- ORDER BY (nested queries)

it is more professional to apply ORDER BY in the outer query

- it is more acceptable logically to sort the *final* version of your dataset

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- some, though not all, nested queries can be rewritten using joins, which are more efficient in general

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- some, though not all, nested queries can be rewritten using joins, which are more efficient in general
- this is true particularly for inner queries using the WHERE clause

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- subqueries:

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **subqueries:**

- allow for better *structuring* of the outer query

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **subqueries:**

- allow for better *structuring* of the outer query
  - thus, each inner query can be thought of in *isolation*

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **subqueries:**

- allow for better structuring of the outer query
  - thus, each inner query can be thought of in isolation
  - *hence the name of SQL - Structured Query Language!*

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **subqueries:**

- allow for better *structuring* of the outer query
  - thus, each inner query can be thought of in *isolation*
  - *hence the name of SQL - Structured Query Language!*
- in some situations, the use of subqueries is much *more intuitive* compared to the use of complex joins and unions

# SQL Subqueries with EXISTS-NOT EXISTS Nested Inside WHERE

- **subqueries:**

- allow for better *structuring* of the outer query
  - thus, each inner query can be thought of in *isolation*
  - *hence the name of SQL - Structured Query Language!*
- in some situations, the use of subqueries is much *more intuitive* compared to the use of complex joins and unions
- many users prefer subqueries simply because they offer *enhanced code readability*

A large, modern conference room is shown from a perspective looking down rows of black office chairs with chrome legs, all facing towards a front wall. The wall features a large, dark rectangular screen or projection area. The room has a polished wooden floor and a ceiling with a grid of recessed lights. The overall atmosphere is professional and spacious.

# SQL Subqueries Nested in SELECT and FROM

# SQL Subqueries Nested in SELECT and FROM

You have advanced with SQL *a lot* at this point!

# SQL Subqueries Nested in SELECT and FROM

You have advanced with SQL *a lot* at this point!



# SQL Subqueries Nested in SELECT and FROM

In this lecture:

# SQL Subqueries Nested in SELECT and FROM

In this lecture:



challenging  
task

# SQL Subqueries Nested in SELECT and FROM

In this lecture:



challenging  
task

# SQL Subqueries Nested in SELECT and FROM

In this lecture:



challenging  
task



exercise

# SQL Self Join

A large conference room with rows of chairs facing a central table. The room has a modern design with a glass partition and large windows. The text "SQL Self Join" is overlaid on the image.

# SQL Self Join

# SQL Self Join

- self join

# SQL Self Join

- **self join**

applied when a table must join itself

# SQL Self Join

- **self join**

applied when a table must join itself

- if you'd like to combine certain rows of a table with other rows of the same table, you need a self-join

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

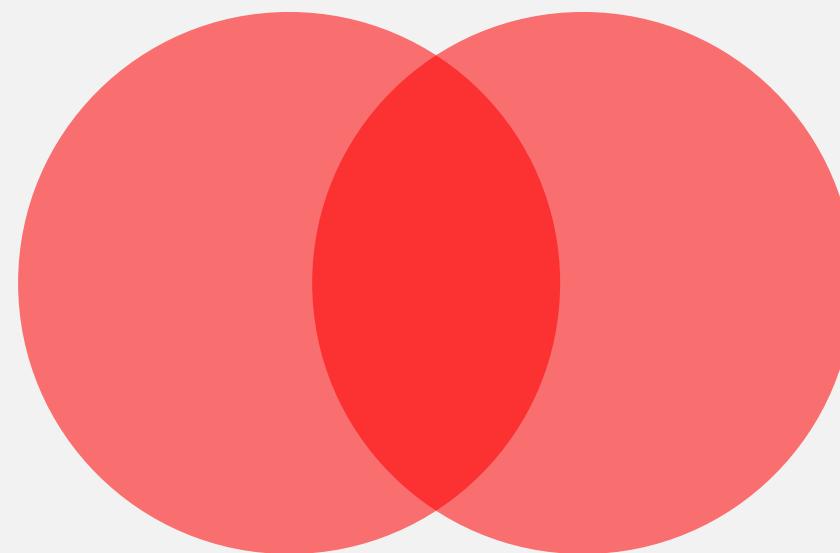
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



Related column: dept\_no

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

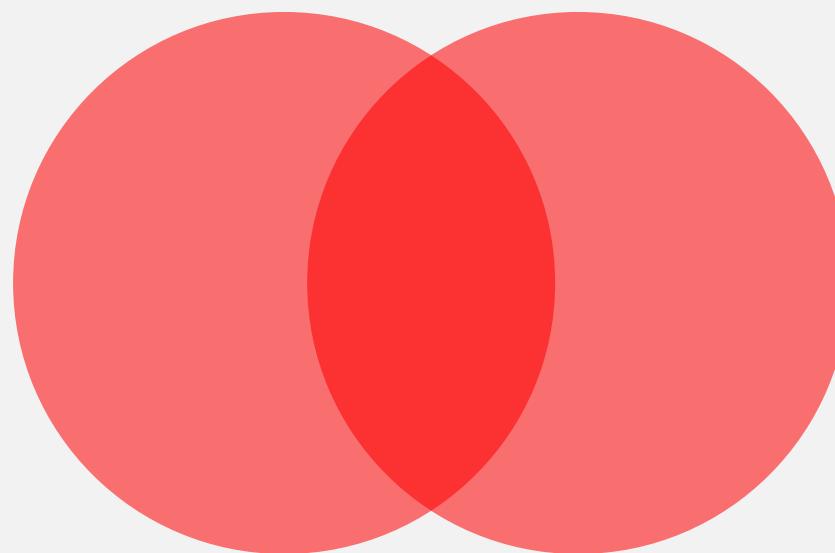
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

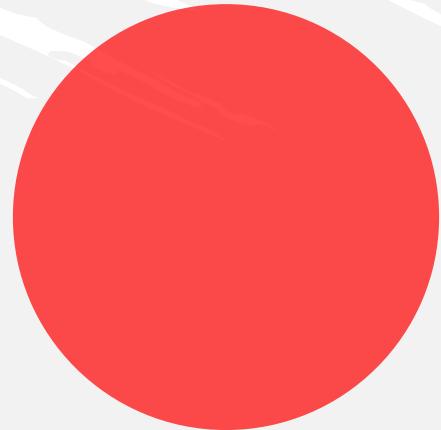


in a self-join statement, you will have to comply with the same logical and syntactic structure

# SQL Self Join

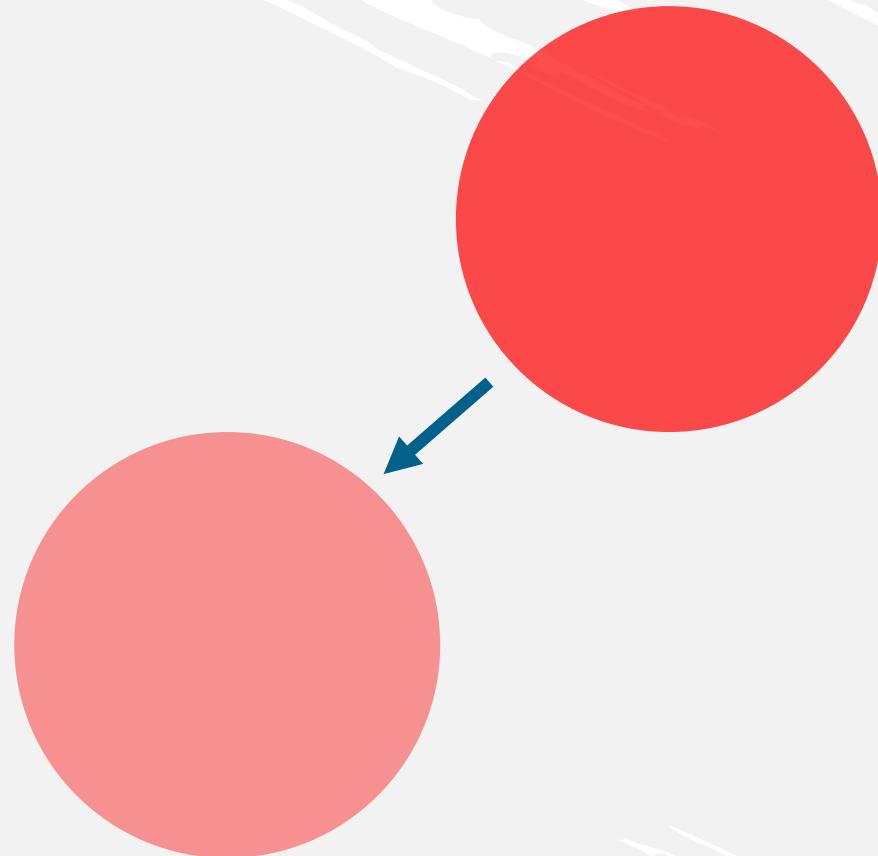
in a self-join statement, you will have to comply with the same logical and syntactic structure

# SQL Self Join



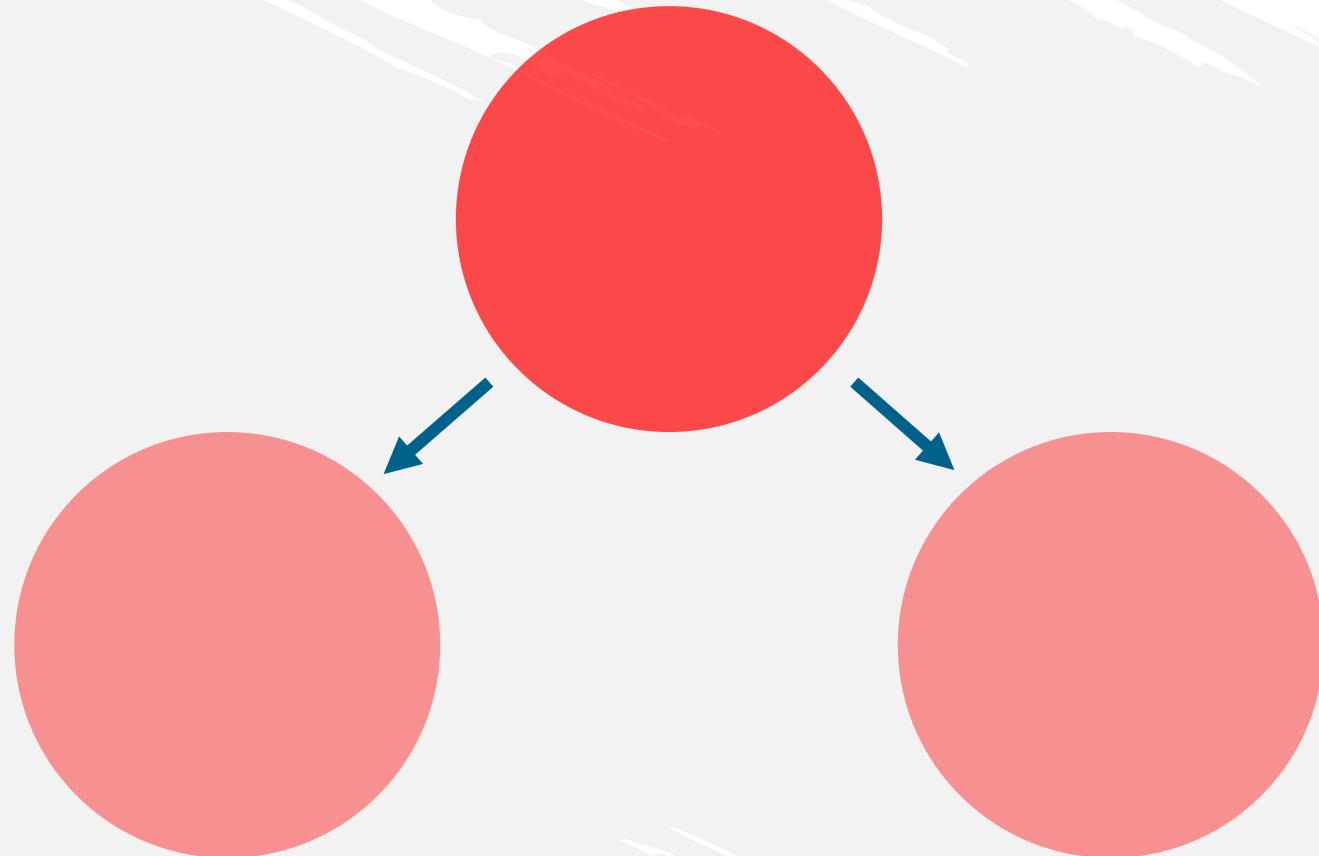
**in a self-join statement, you will have to comply with the same logical and syntactic structure**

# SQL Self Join



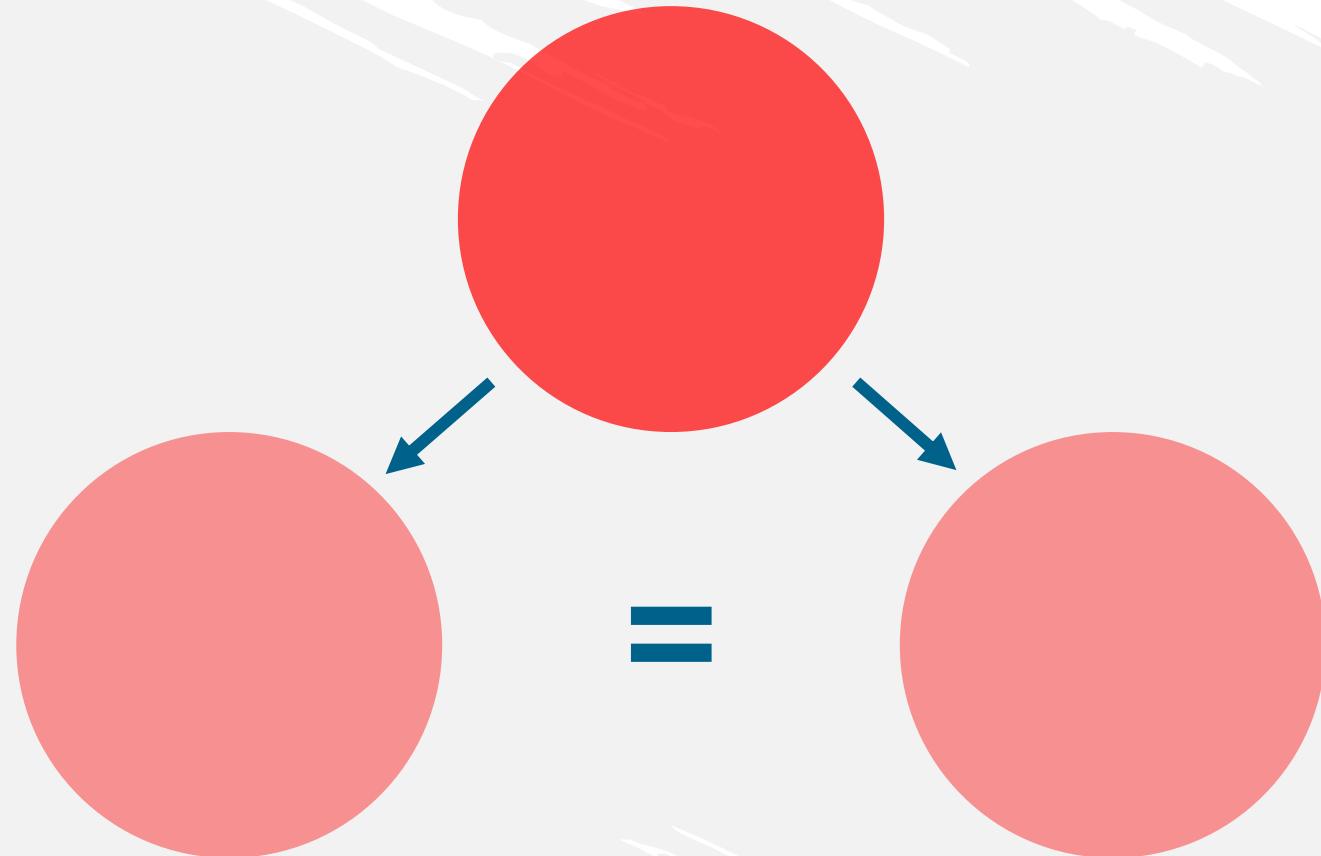
- the 2 tables will be *identical* to the table you'll be using in the self-join

# SQL Self Join



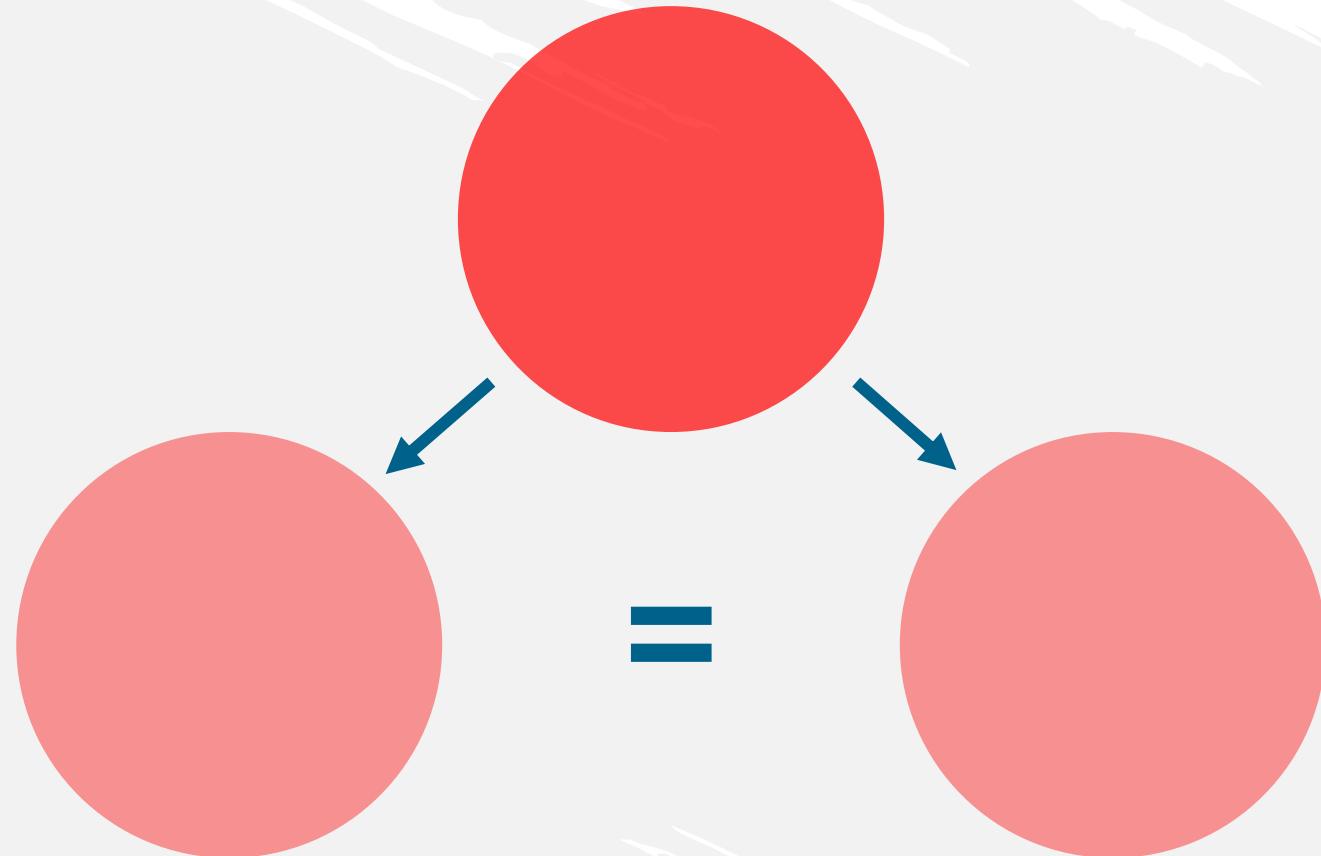
- the 2 tables will be *identical* to the table you'll be using in the self-join

# SQL Self Join



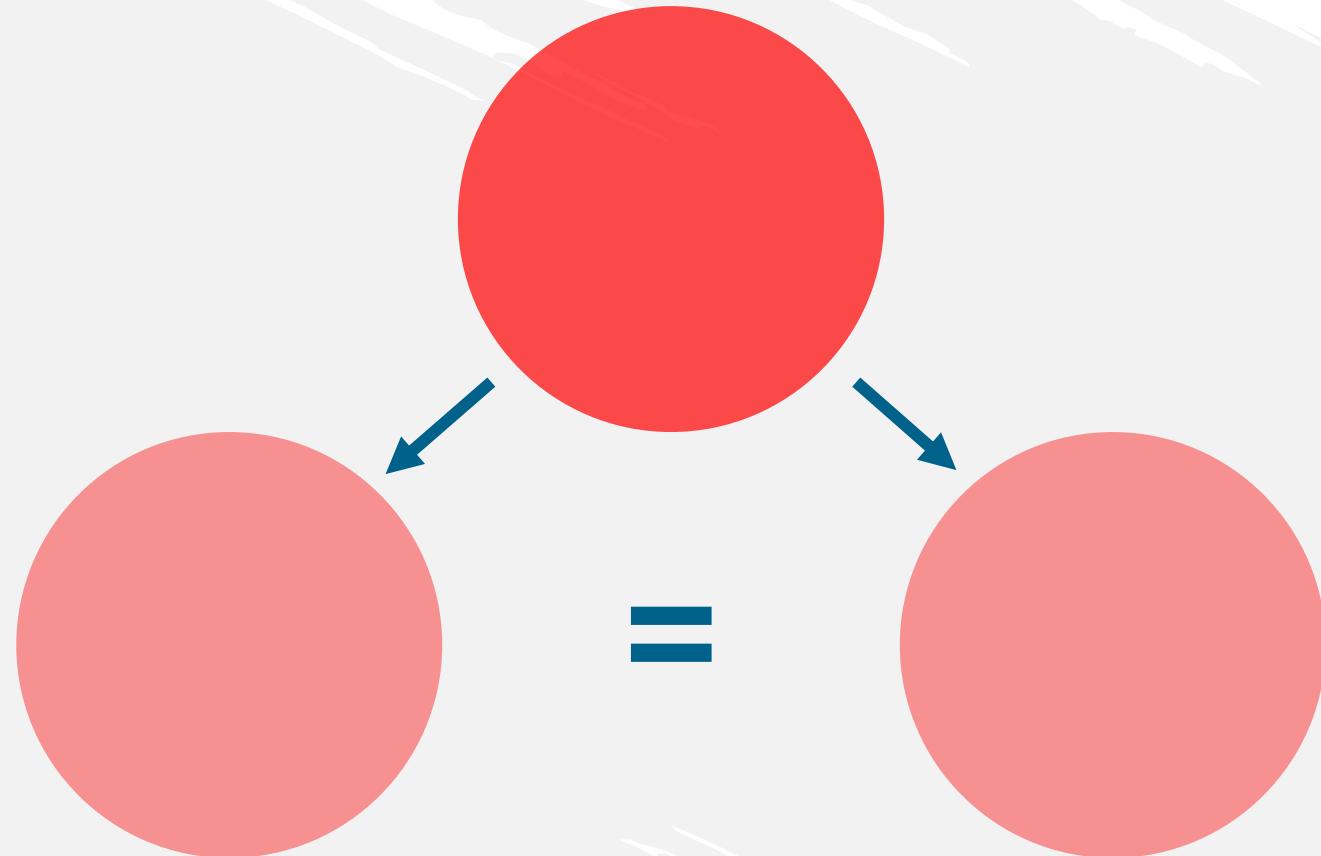
- the 2 tables will be *identical* to the table you'll be using in the self-join

# SQL Self Join



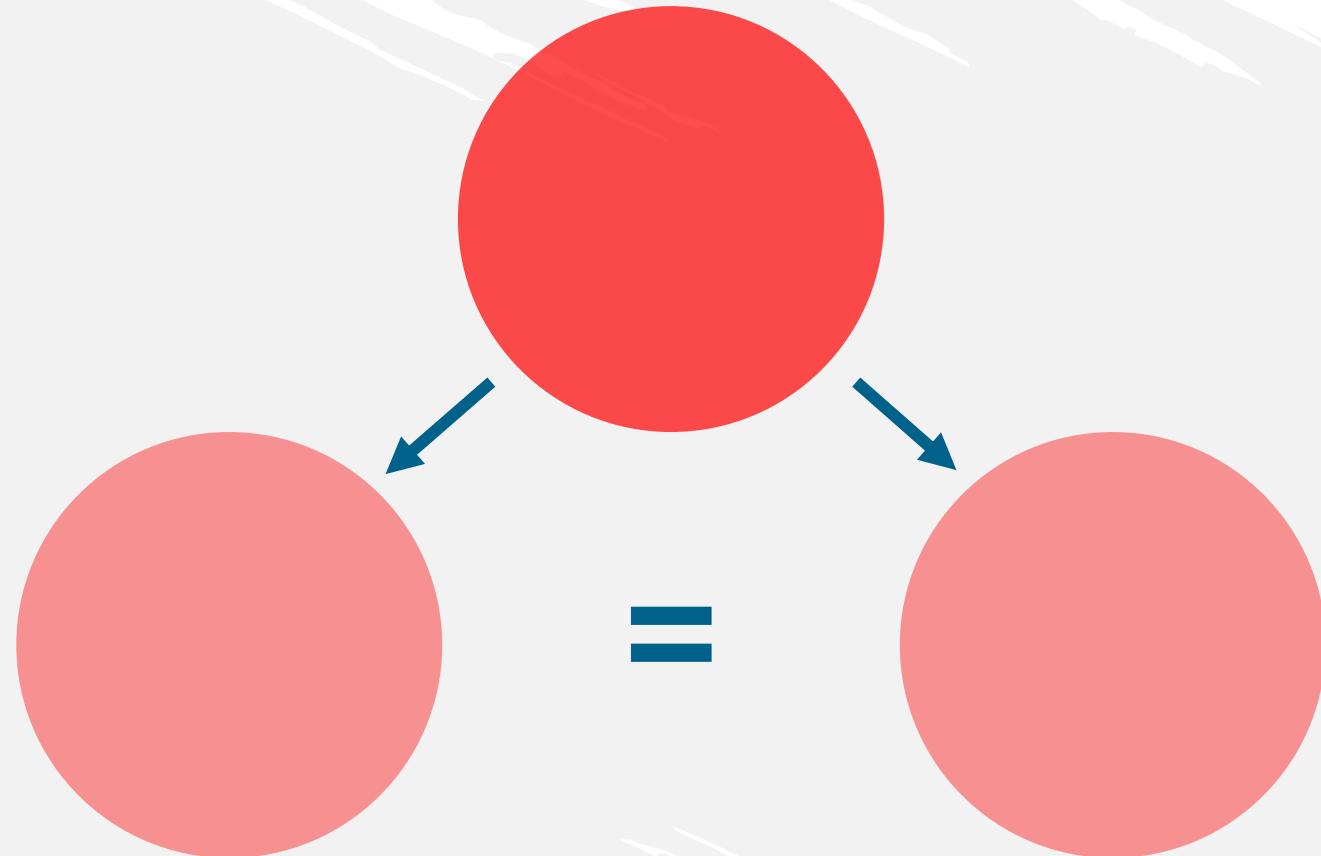
- the 2 tables will be *identical* to the table you'll be using in the self-join
- you can think of them as virtual projections of the underlying, base table

# SQL Self Join



- the self-join will reference both implied tables and will treat them as two separate tables in its operations

# SQL Self Join



- the data used will come from a single source, which is the underlying table that stores data *physically*

# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

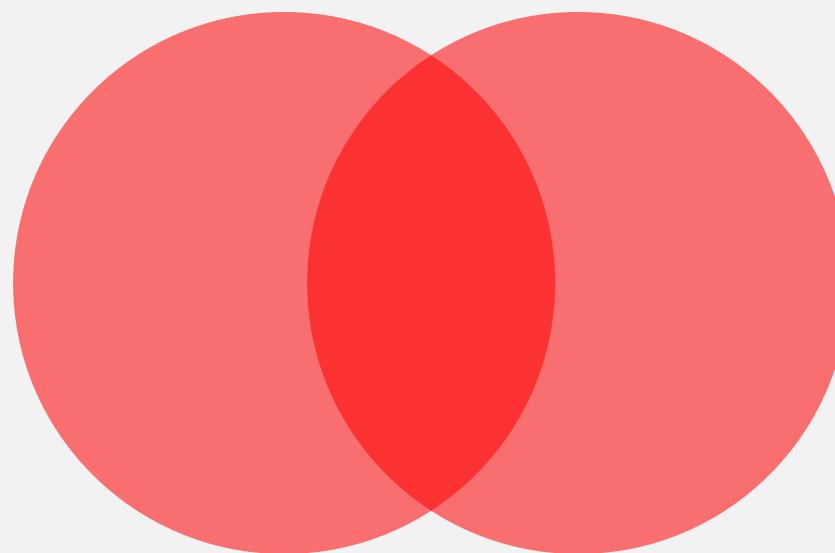
from\_date DATE

to\_date DATE

departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)



# INNER JOIN

dept\_manager\_dup

dept\_no CHAR(4)

emp\_no INT

from\_date DATE

to\_date DATE

M

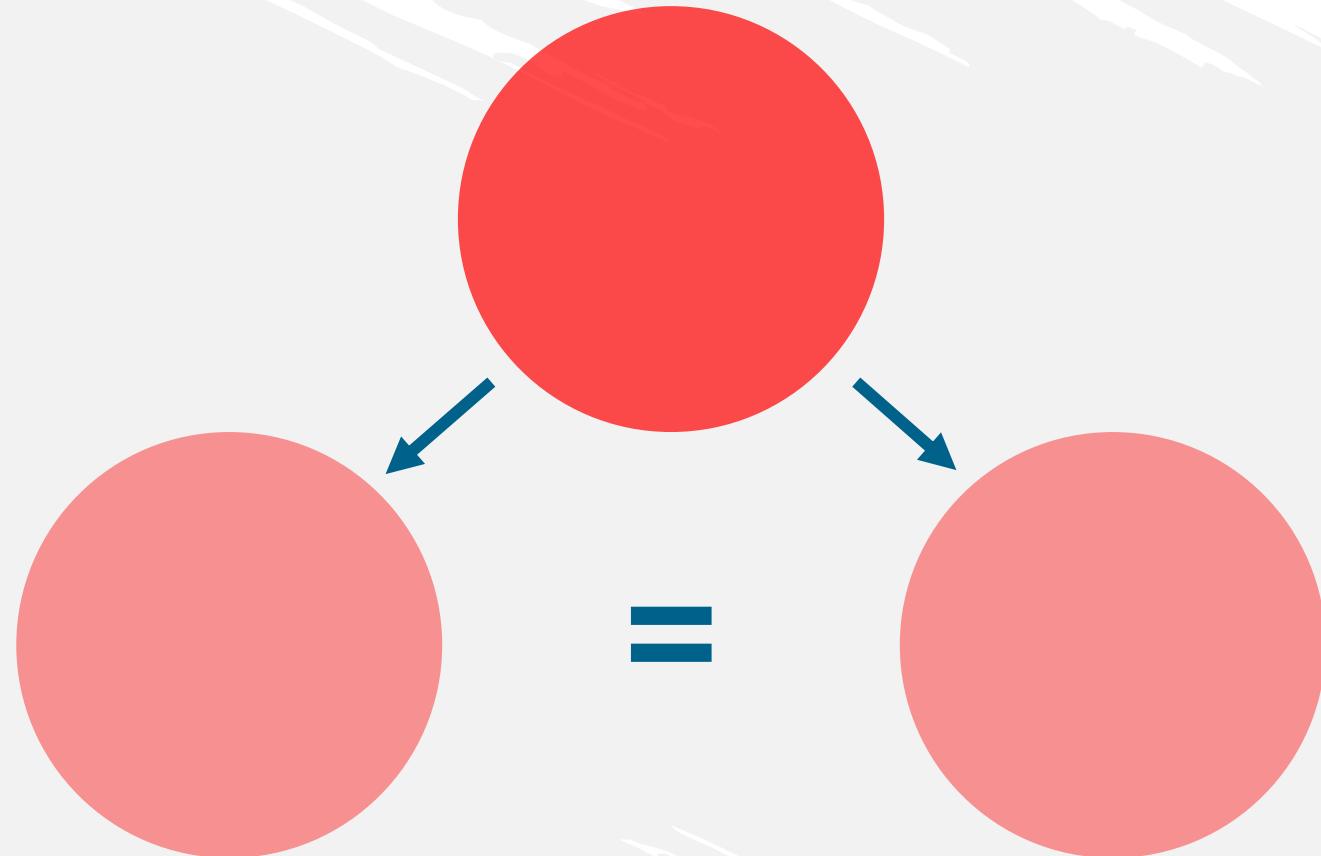
departments\_dup

dept\_no CHAR(4)

dept\_name VARCHAR(40)

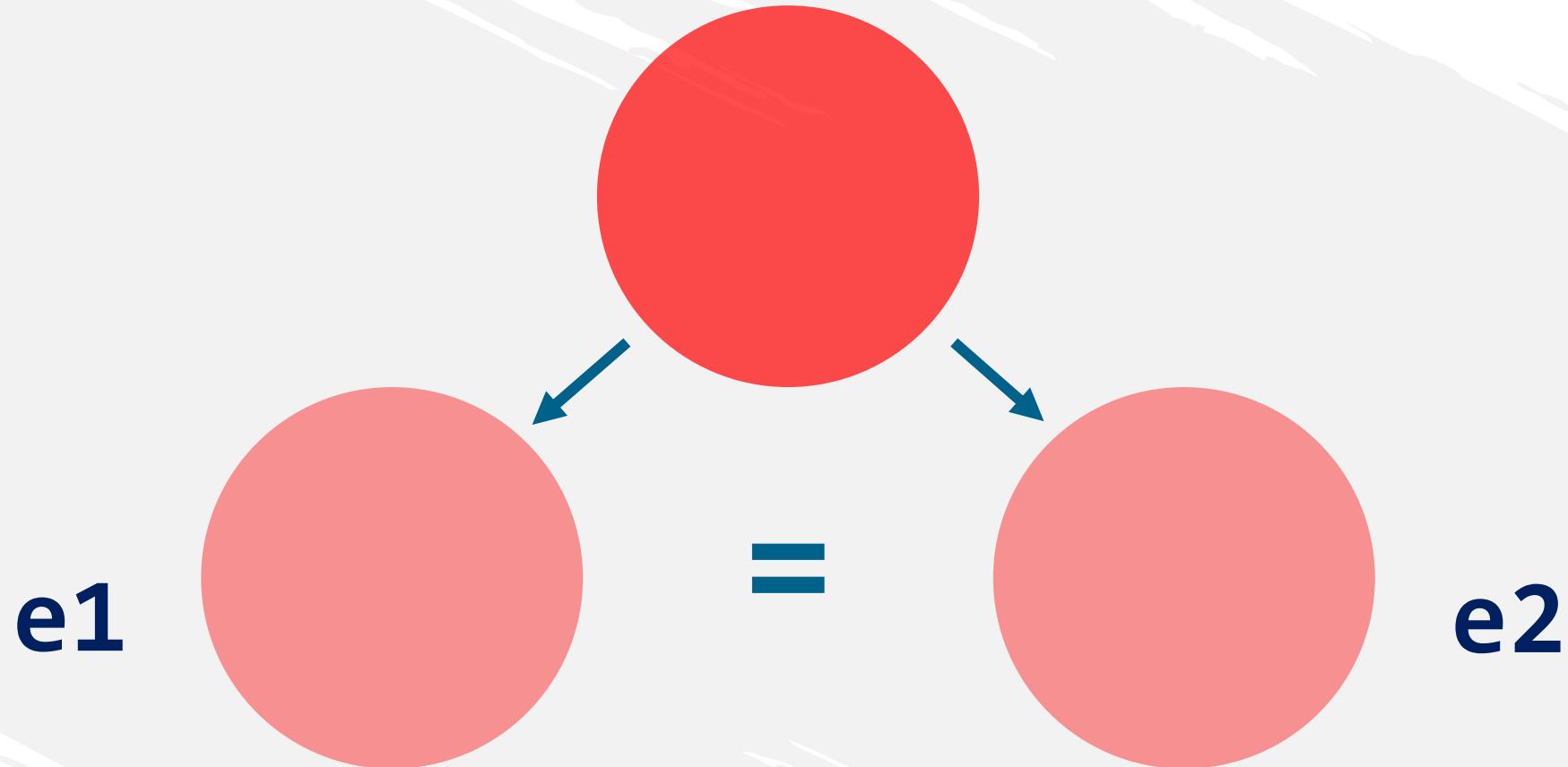
D

# SQL Self Join



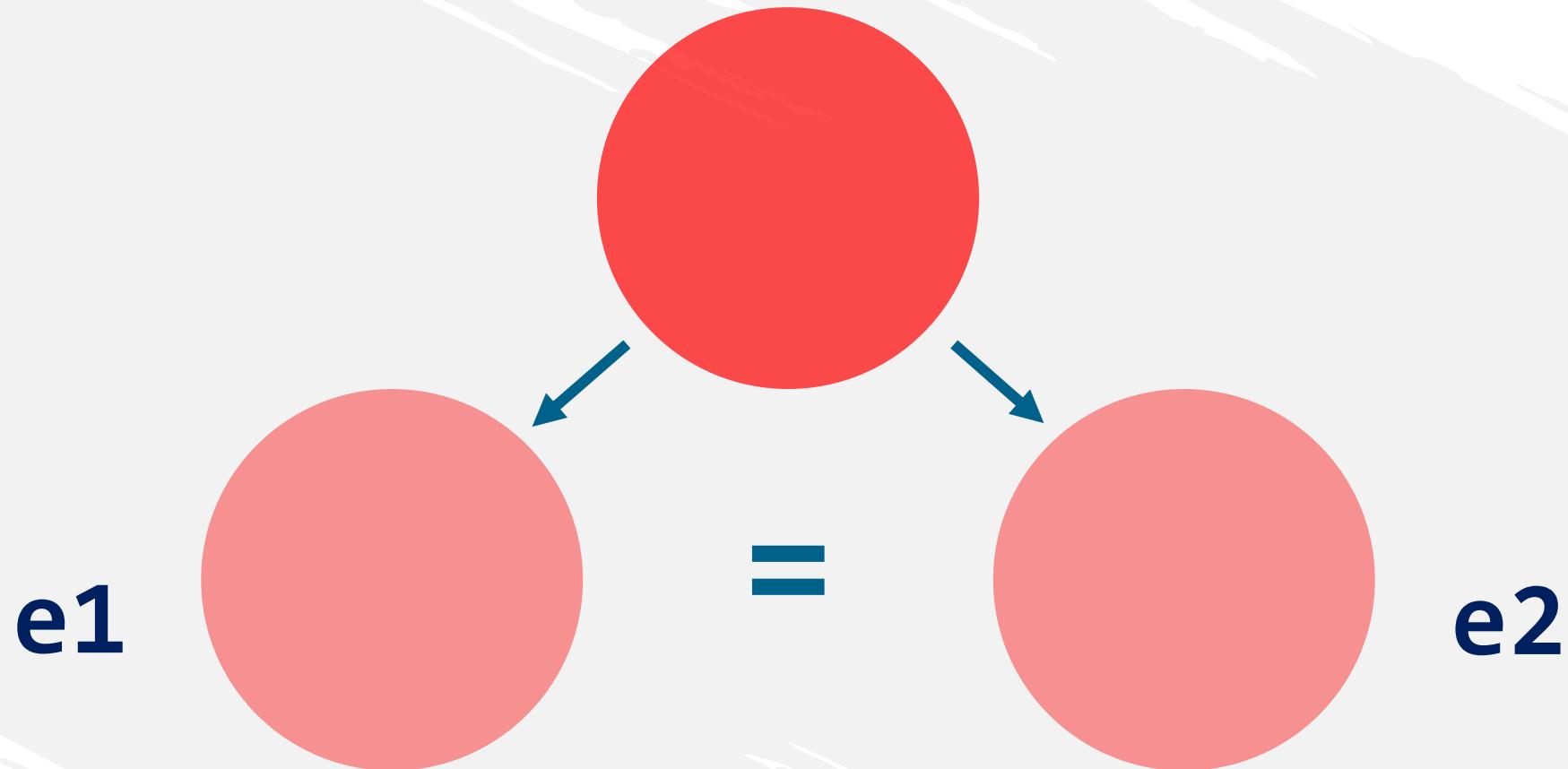
- using aliases is *obligatory*

# SQL Self Join



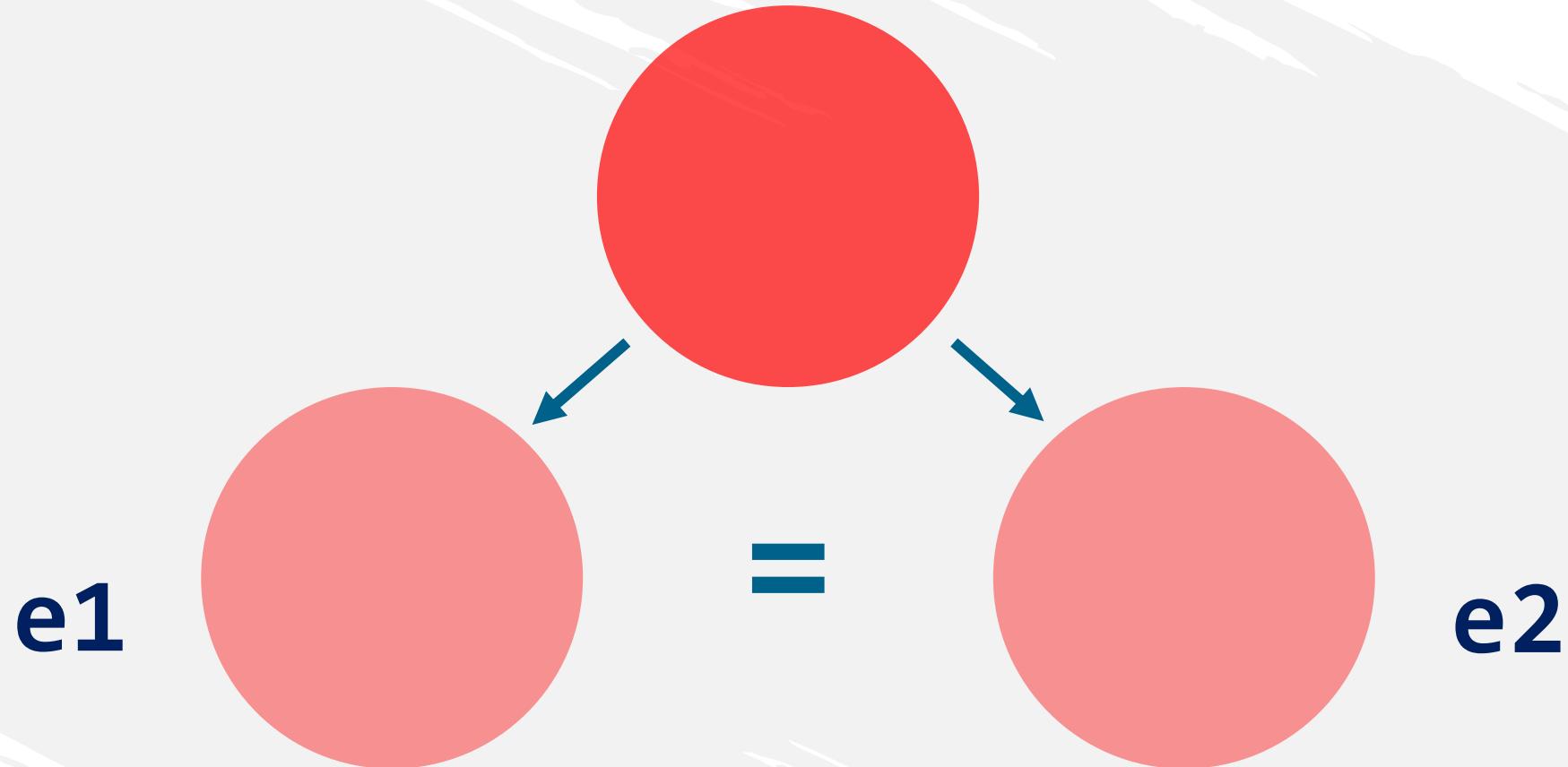
- using aliases is *obligatory*

## SQL Self Join



- these references to the original table let you use different blocks of the available data

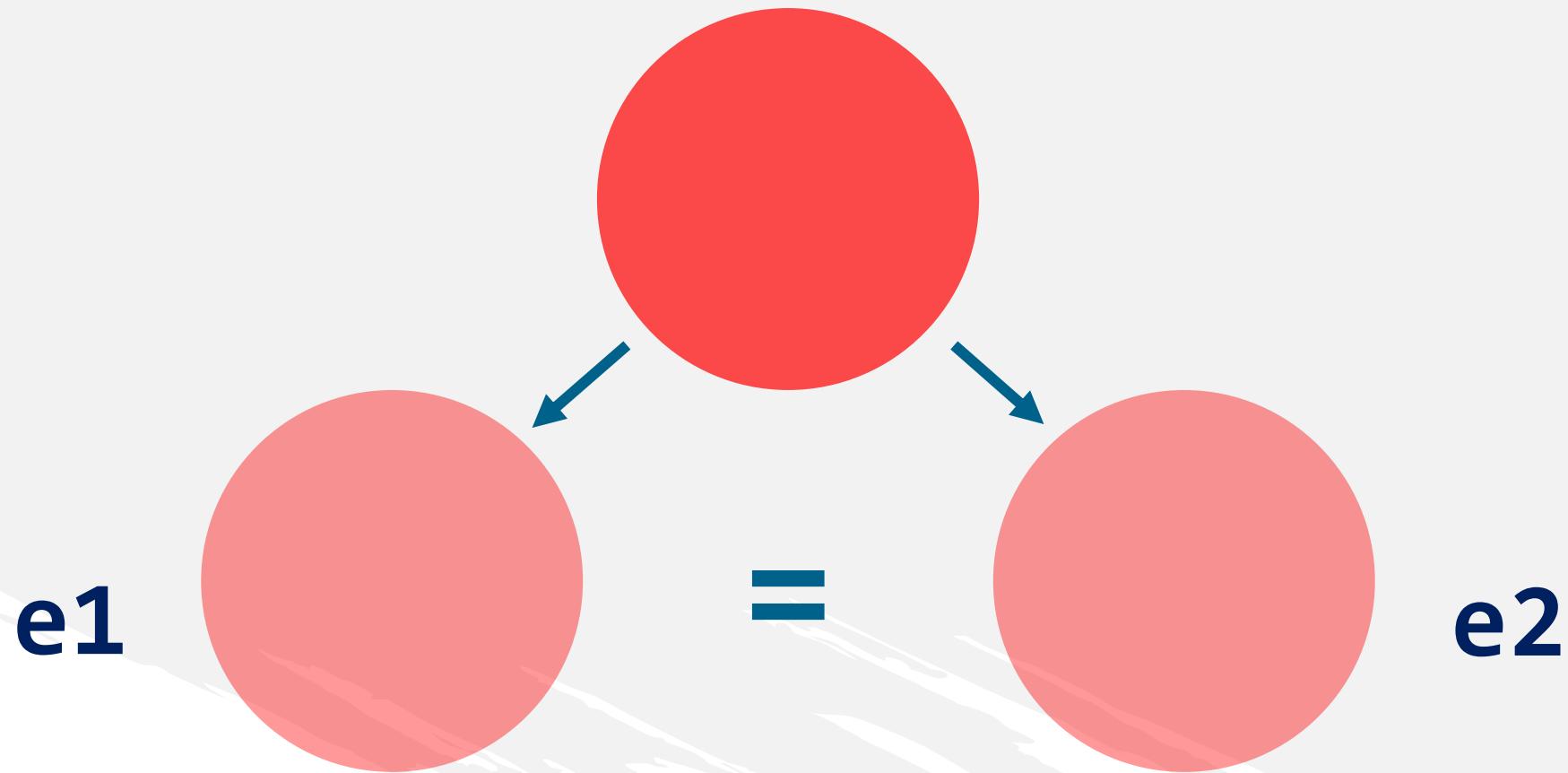
## SQL Self Join



- you can either filter both in the join, or you can filter one of them in the WHERE clause, and the other one - in the join

# SQL Self Join

`emp_manager`





# SQL Views

A large conference room with rows of chairs facing a central table. The room has a modern design with a glass partition and large windows overlooking a landscape. The chairs are arranged in several rows, pointing towards the front of the room where a table is set up.

# Using SQL Views

# Using SQL Views

## view

a virtual table whose contents are obtained from an existing table or tables, called **base tables**

# Using SQL Views

## view

a virtual table whose contents are obtained from an existing table or tables, called ***base tables***

- the retrieval happens through an SQL statement, incorporated into the view

# Using SQL Views

- SQL View

# Using SQL Views

- **SQL View**

- think of a view object as *a view into the base table*

# Using SQL Views

- **SQL View**

- think of a view object as *a view into the base table*
- the view itself does *not* contain any real data; the data is physically stored in the base table

# Using SQL Views

- **SQL View**

- think of a view object as *a view into the base table*
- the view itself does *not* contain any real data; the data is physically stored in the base table
- the view *simply shows the data contained in the base table*

# Using SQL Views

- SQL View



SQL

```
CREATE VIEW view_name AS  
SELECT  
    column_1, column_2,... column_n  
FROM  
    table_name;
```

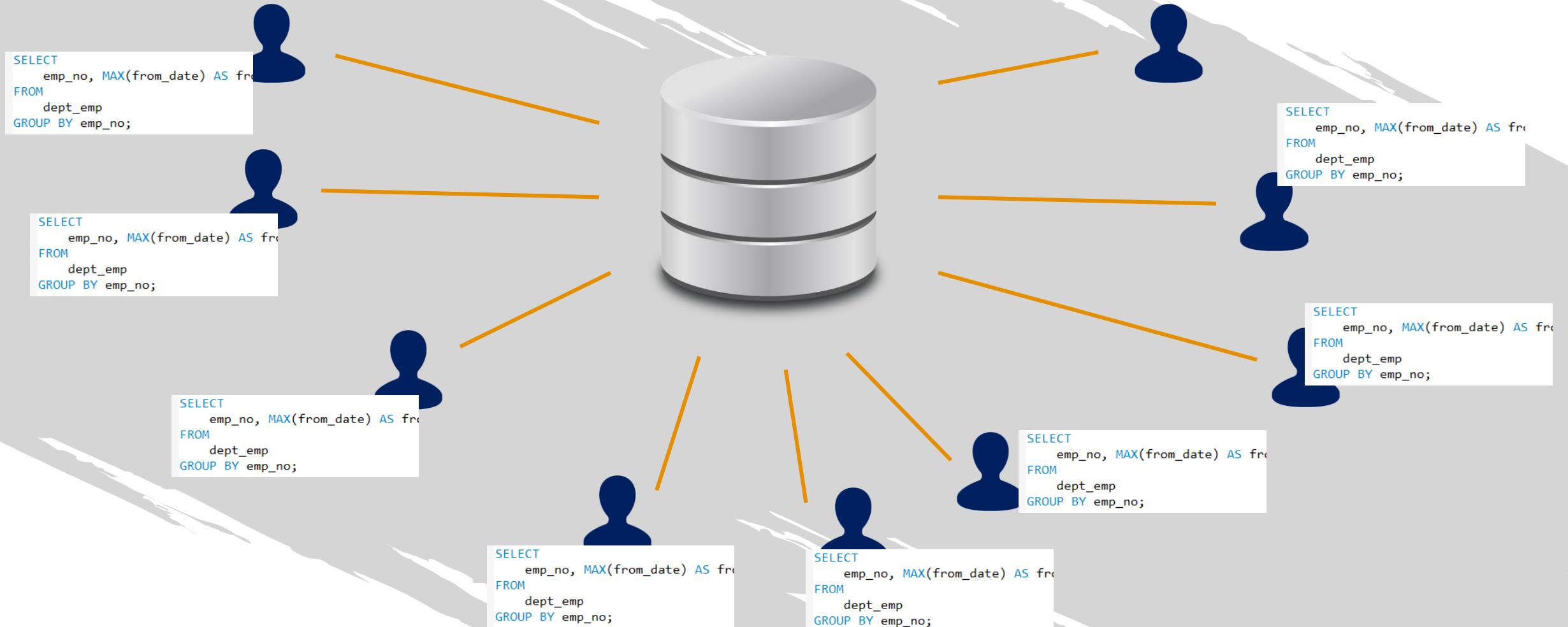
# Using SQL Views



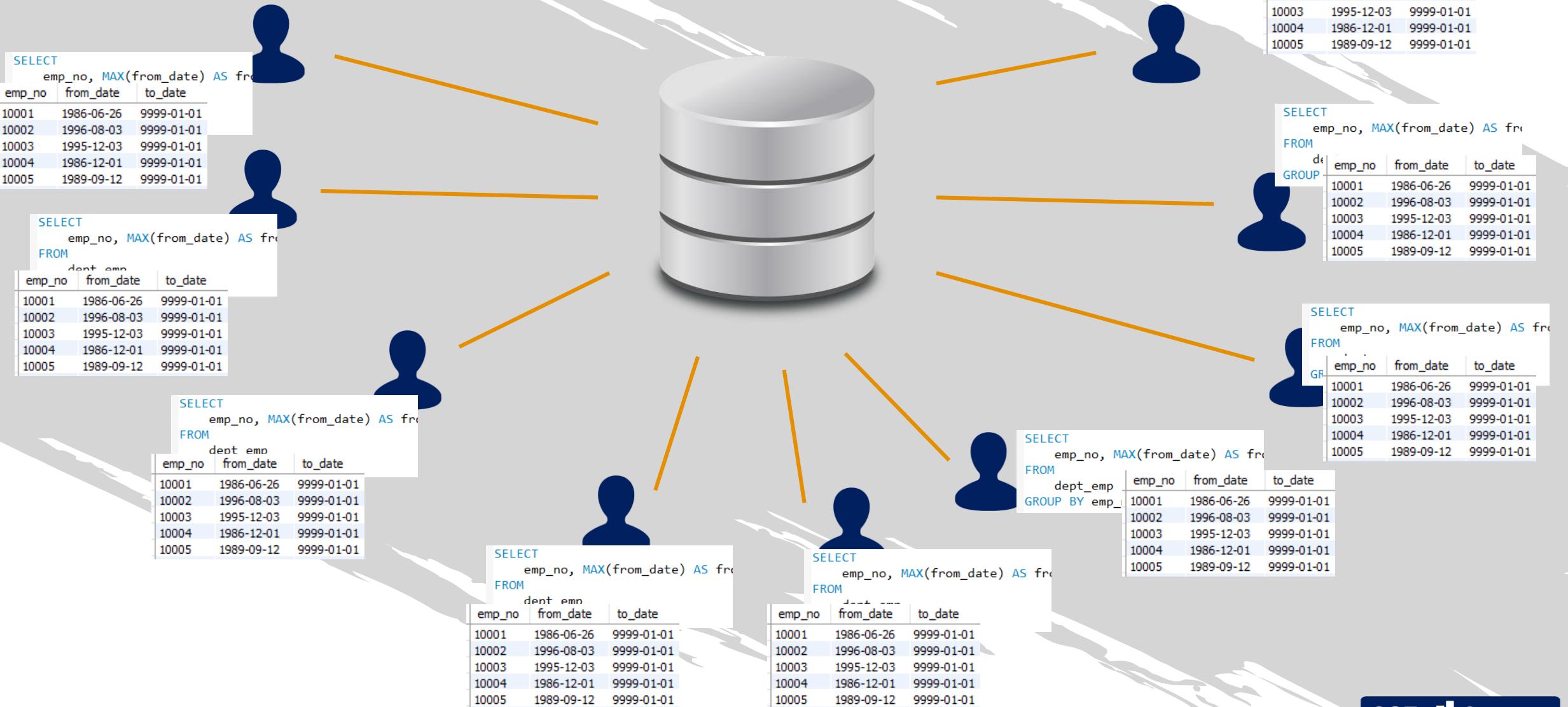
# Using SQL Views



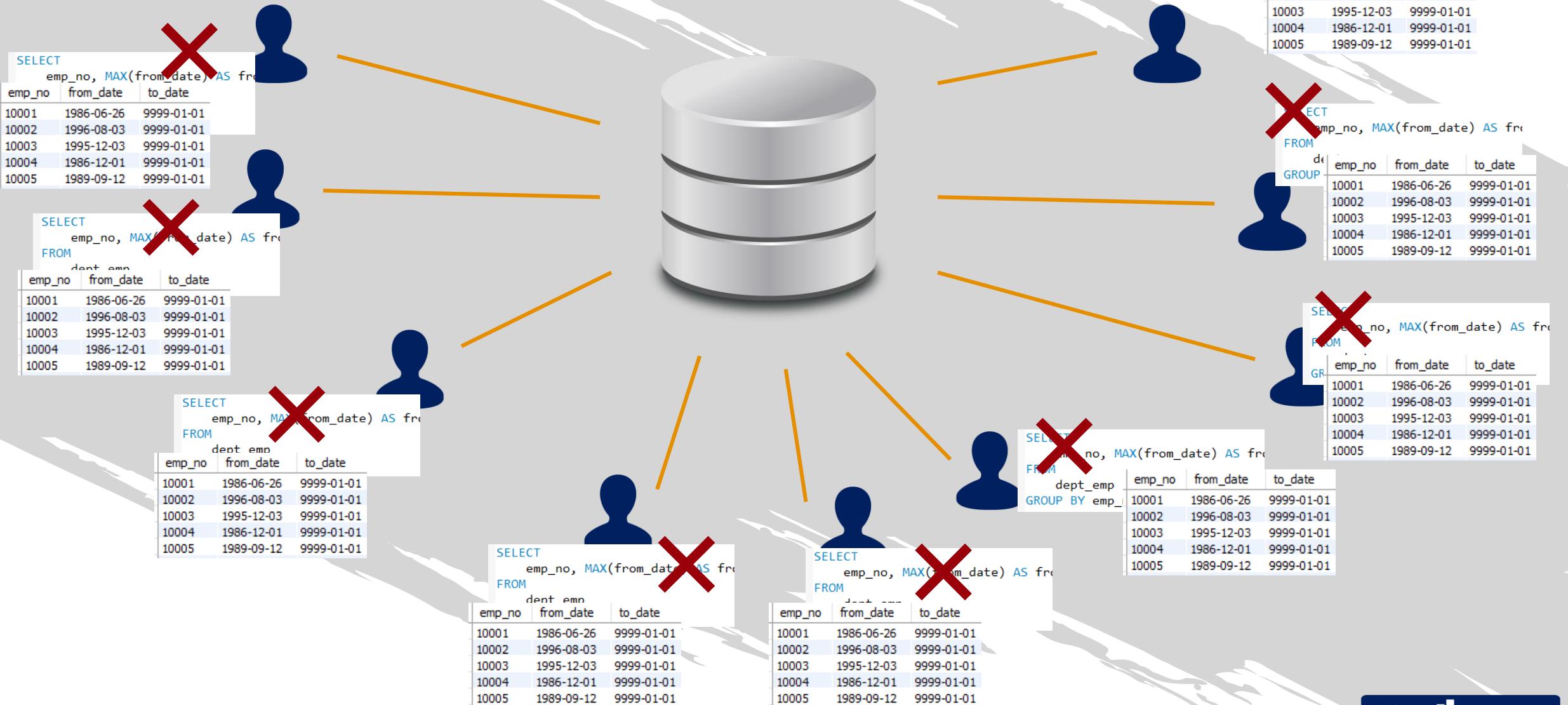
# Using SQL Views



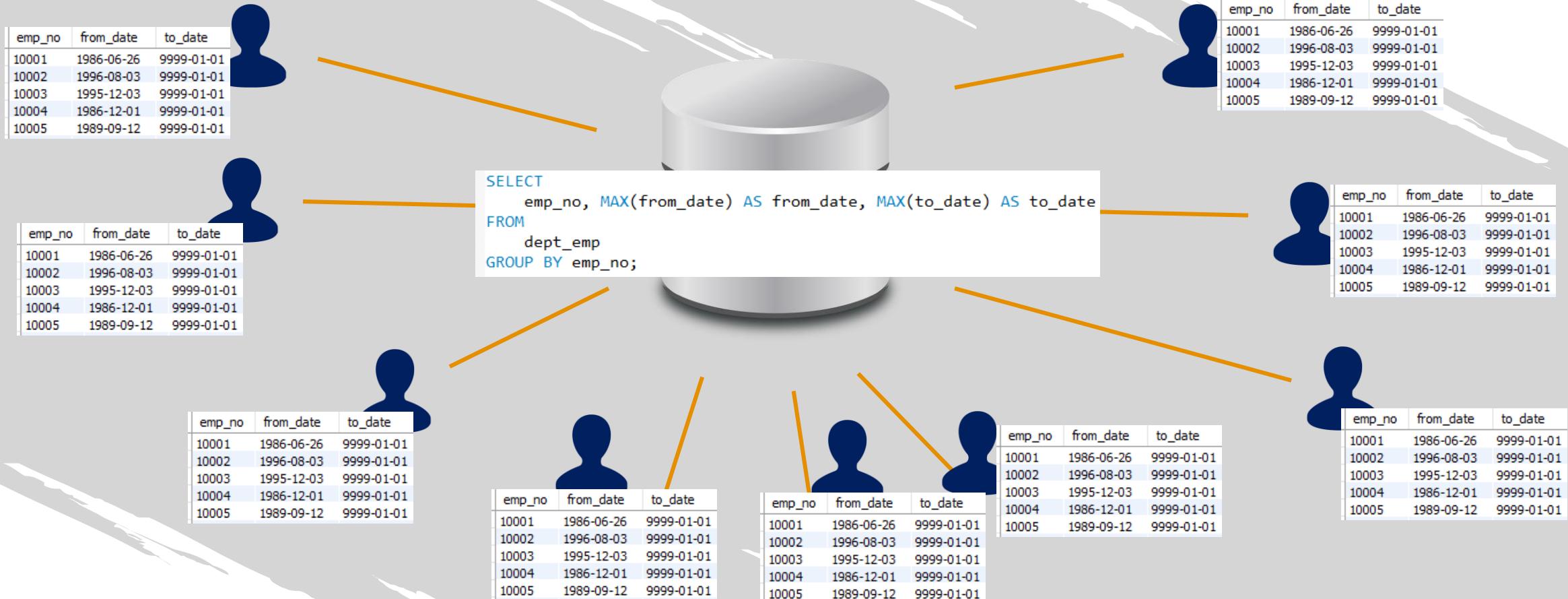
# Using SQL Views



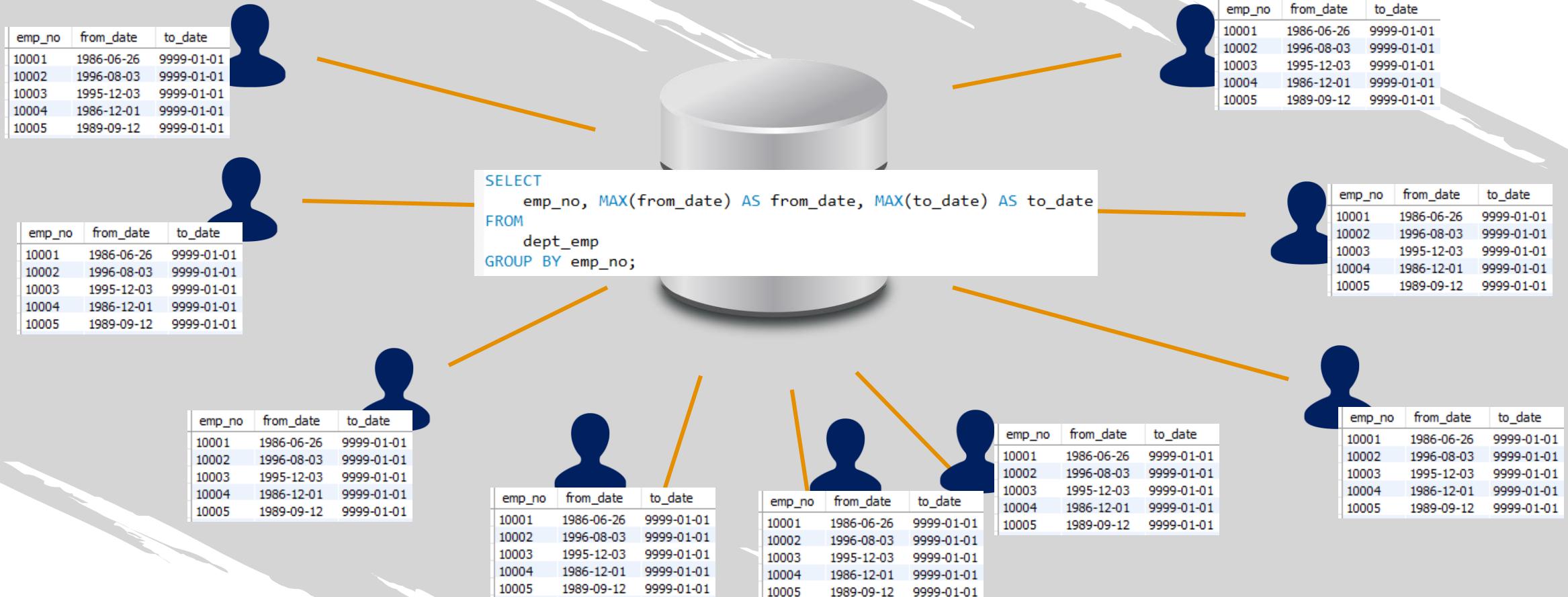
# Using SQL Views



# Using SQL Views



# Using SQL Views



A view acts as a *shortcut* for writing the same `SELECT` statement every time a new request has been made

# Using SQL Views

- **SQL View**

- saves a lot of coding time

# Using SQL Views

- **SQL View**

- saves a lot of coding time
- occupies no extra memory

# Using SQL Views

- **SQL View**

- acts as a *dynamic table* because it instantly reflects data and structural changes in the base table

# Using SQL Views

- **SQL View**

- acts as a *dynamic table* because it instantly reflects data and structural changes in the base table

‘dept\_emp’ (table)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	9999-01-01

# Using SQL Views

- SQL View

- acts as a *dynamic table* because it instantly reflects data and structural changes in the base table

‘dept\_emp’ (table)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	9999-01-01

‘dept\_emp’ (view)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	9999-01-01



# Using SQL Views

- **SQL View**

- acts as a *dynamic table* because it instantly reflects data and structural changes in the base table

‘dept\_emp’ (table)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	9999-01-01

# Using SQL Views

- **SQL View**

- acts as a *dynamic table* because it instantly reflects data and structural changes in the base table

‘dept\_emp’ (table)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	2025-06-05

# Using SQL Views

- SQL View

- acts as a *dynamic table* because it instantly reflects data and structural changes in the base table

‘dept\_emp’ (table)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	2025-06-05

‘dept\_emp’ (view)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	9999-01-01

# Using SQL Views

- SQL View

- acts as a *dynamic table* because it instantly reflects data and structural changes in the base table

‘dept\_emp’ (table)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	2025-06-05

‘dept\_emp’ (view)

emp_no	dept_no	from_date	to_date
10001	d005	1986-06-26	2025-06-05



# Using SQL Views

- SQL Views

# Using SQL Views

- **SQL Views**

Don't forget they are not real, physical data sets, meaning we cannot insert or update the information that has already been extracted.

# Using SQL Views

- **SQL Views**

Don't forget they are not real, physical data sets, meaning we cannot insert or update the information that has already been extracted.

- they should be seen as *temporary virtual data tables* retrieving information from base tables

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

# Stored Routines

A large, modern conference room is shown from a perspective looking down the length of the room. On the left, there are several rows of light-colored office chairs arranged facing a front wall. The wall features a large, dark rectangular screen or projection area. Above the screen, there are two smaller, mounted displays. The room has a polished wooden floor and a ceiling with a grid of recessed lighting fixtures. The overall atmosphere is professional and spacious.

# Introduction to Stored Routines

# Introduction to Stored Routines

**routine** (*in a context other than computer science*)

a usual, fixed action, or series of actions, repeated periodically

# Introduction to Stored Routines

```
SELECT  
    emp_no, MAX(from_date) AS from_date, MAX(to_date)  
FROM  
    dept_emp  
GROUP BY emp_no;
```

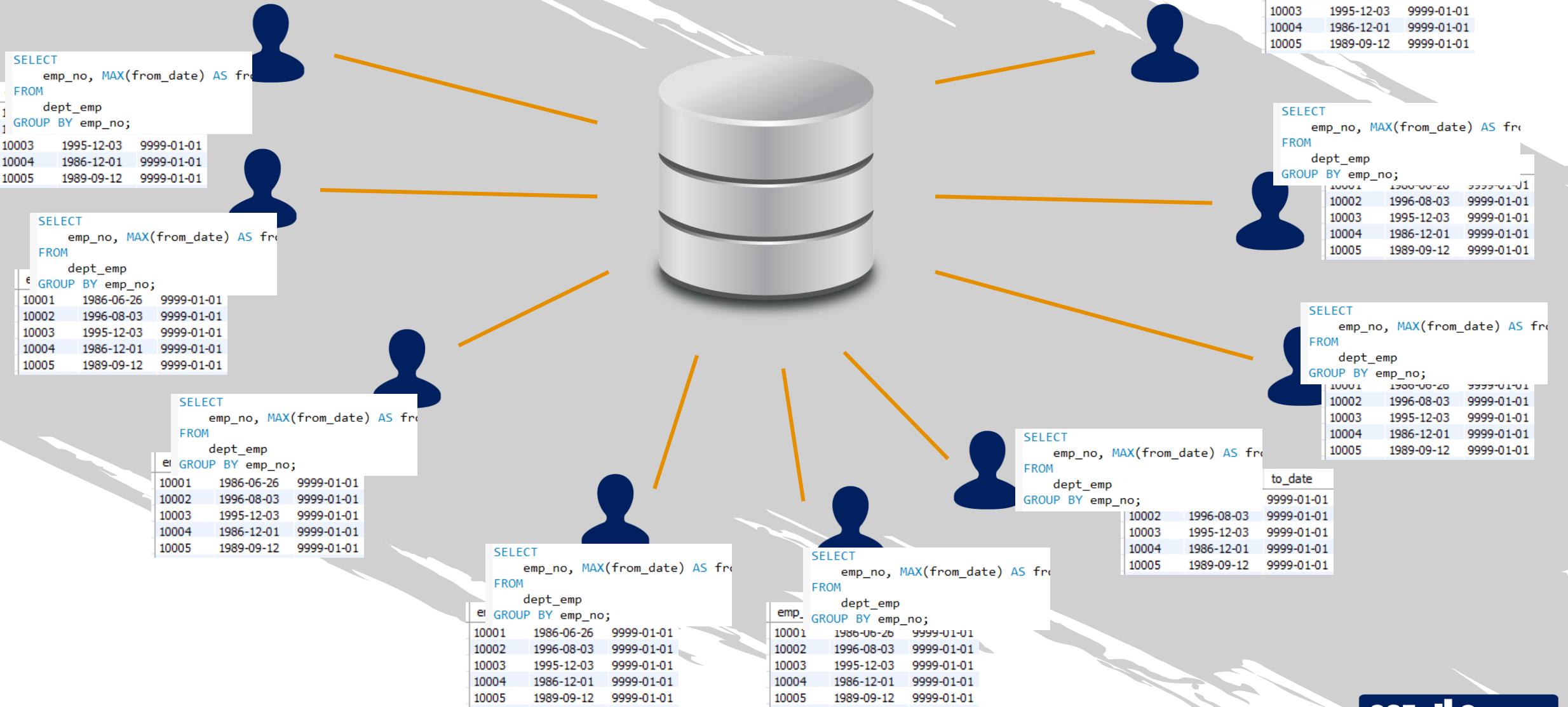
query

emp_no	from_date	to_date
10001	1986-06-26	9999-01-01
10002	1996-08-03	9999-01-01
10003	1995-12-03	9999-01-01
10004	1986-12-01	9999-01-01
10005	1989-09-12	9999-01-01

output



# Introduction to Stored Routines



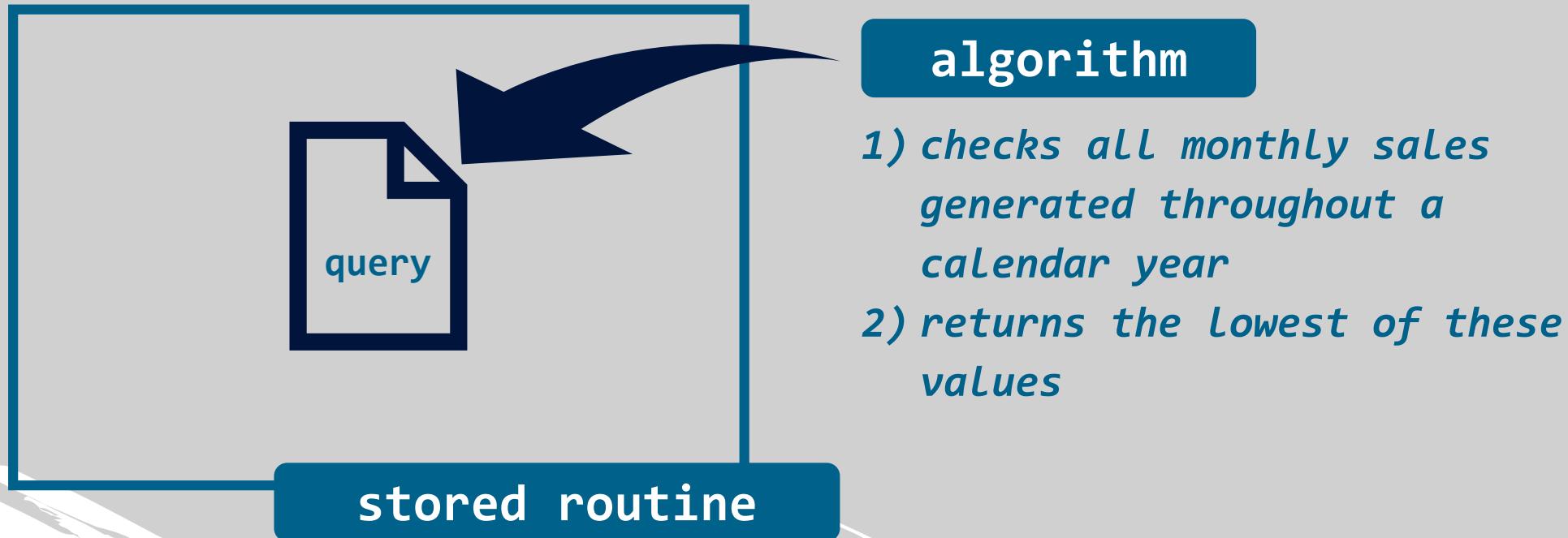
# Introduction to Stored Routines

## stored routine

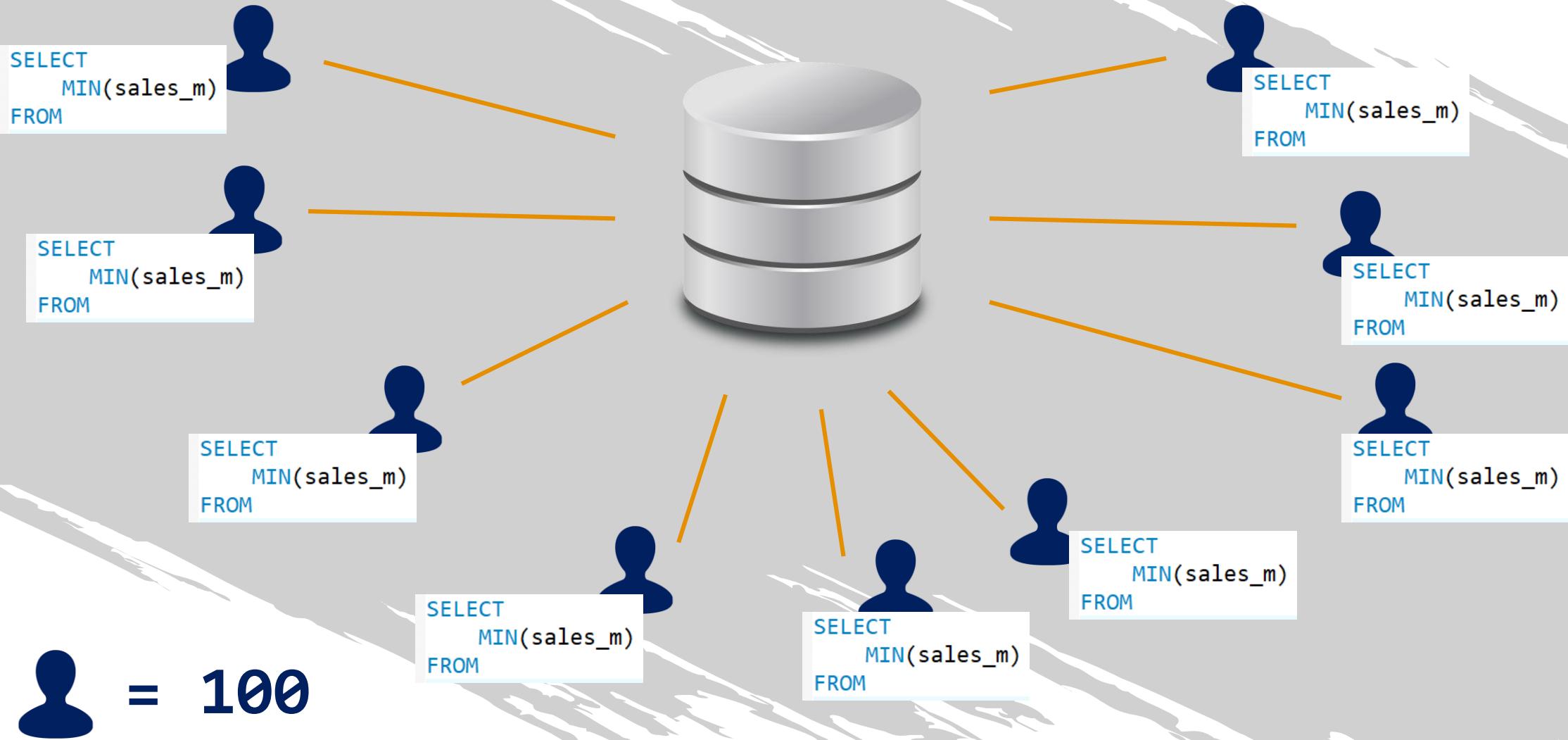
an SQL statement, or a set of SQL statements, that can be stored on the database server

- whenever a user needs to run the query in question, they can call, reference, or invoke the routine

# Introduction to Stored Routines



# Introduction to Stored Routines



# Introduction to Stored Routines

```
SELECT  
    MIN(sales_m)  
FROM
```

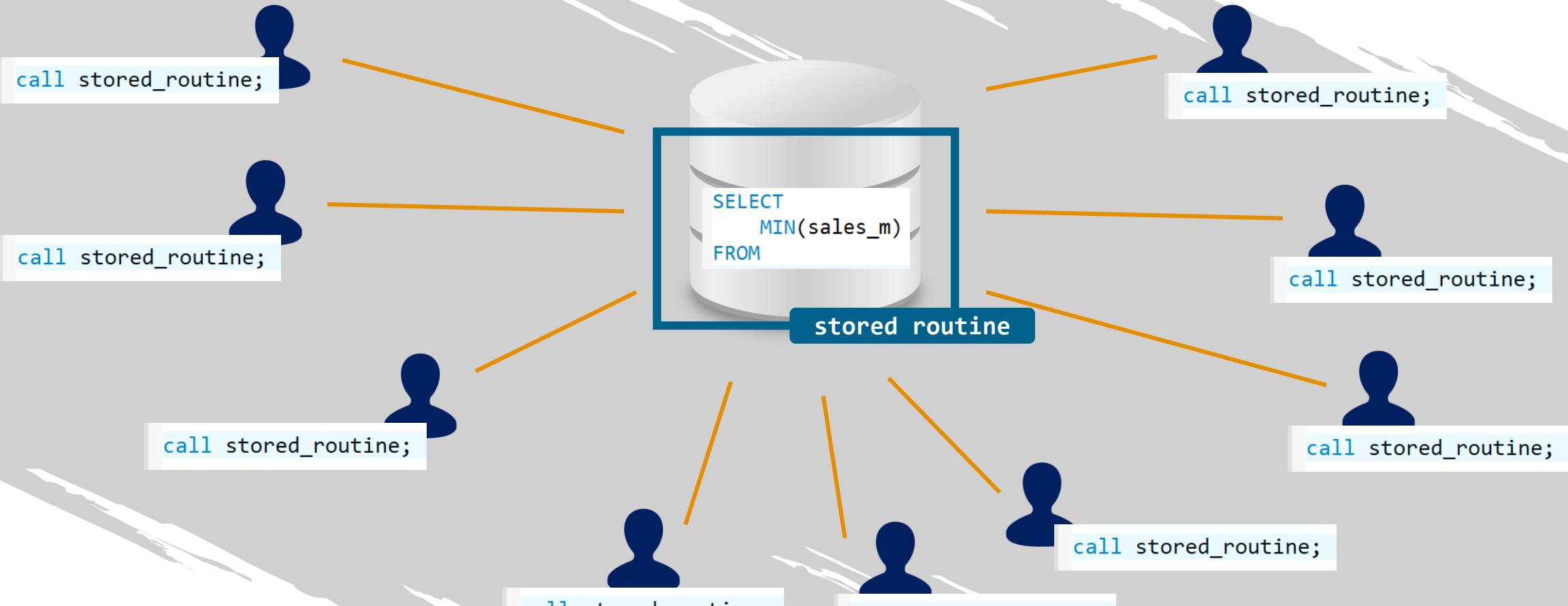
stored routine

# Introduction to Stored Routines

```
SELECT  
    MIN(sales_m)  
FROM
```

stored routine

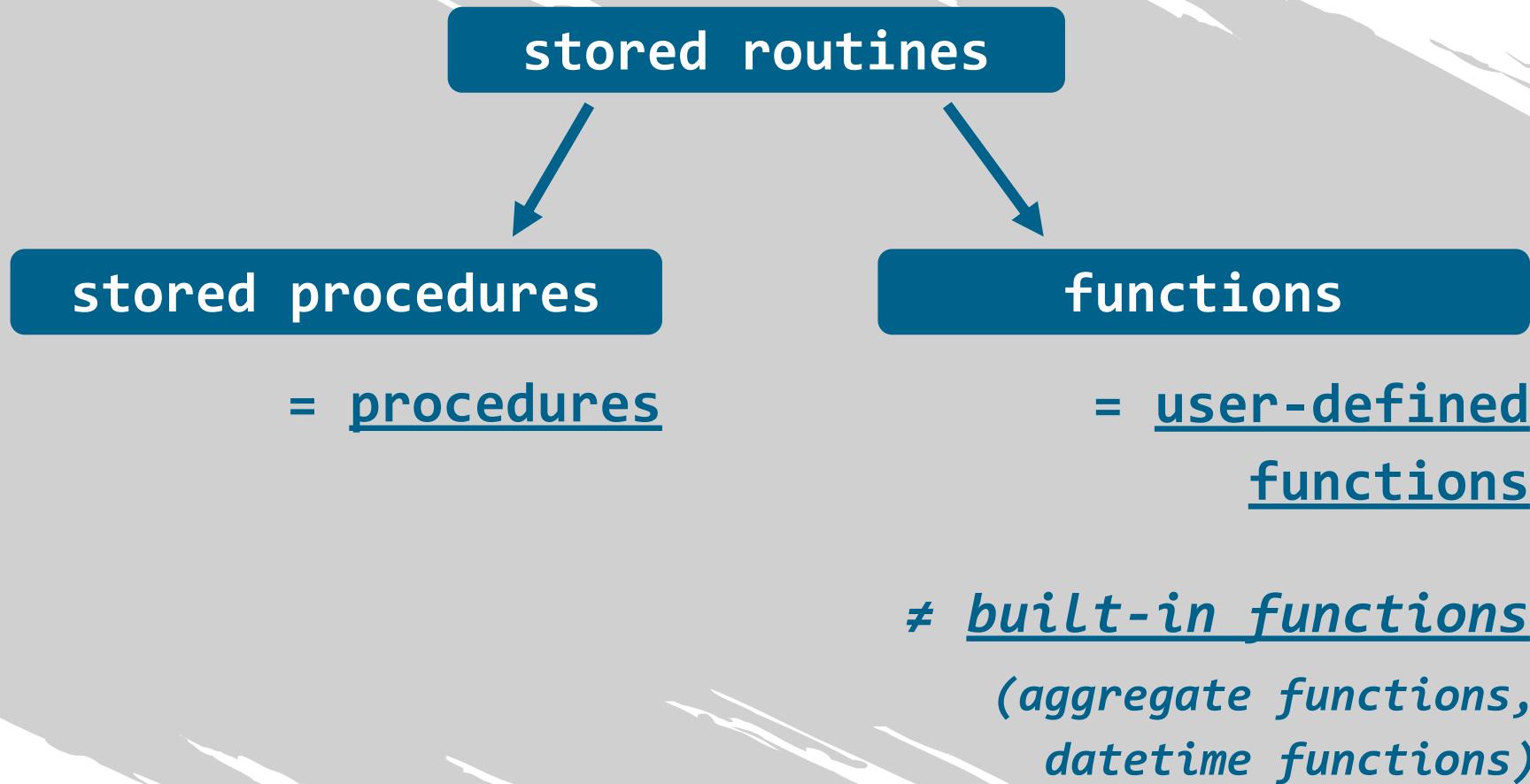
# Introduction to Stored Routines



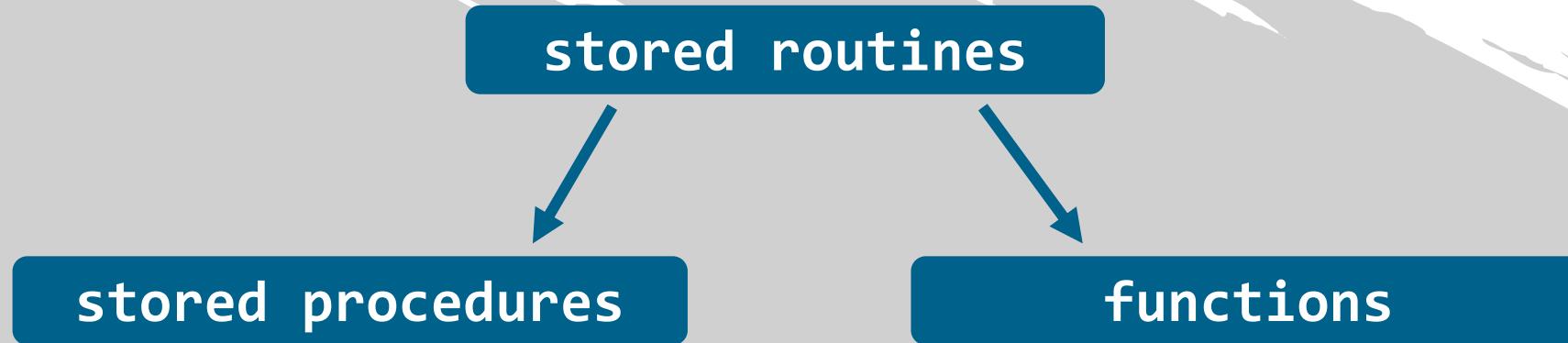
= 100

- this routine can bring the desired result multiple times

# Introduction to Stored Routines



# Introduction to Stored Routines



A large, modern conference room is shown from a perspective looking down rows of black office chairs with wheels. The room has a high ceiling with a grid of recessed lights. On the far wall, there are four large, dark rectangular panels, possibly projection screens or windows. The overall atmosphere is professional and spacious.

# The MySQL Syntax for Stored Procedures

# The MySQL Syntax for Stored Procedures

- semi-colons

;

- they function as a statement terminator
- technically, they can also be called delimiters
- by typing *DELIMITER \$\$*, you'll be able to use the dollar symbols as your delimiter



DELIMITER \$\$

SQL

# The MySQL Syntax for Stored Procedures

Query #1 ;

Query #2 ;

call stored\_procedure\_1 ;

p\_Query #1 ;

p\_Query #2 ;

stored procedure

# The MySQL Syntax for Stored Procedures

Query #1 ;

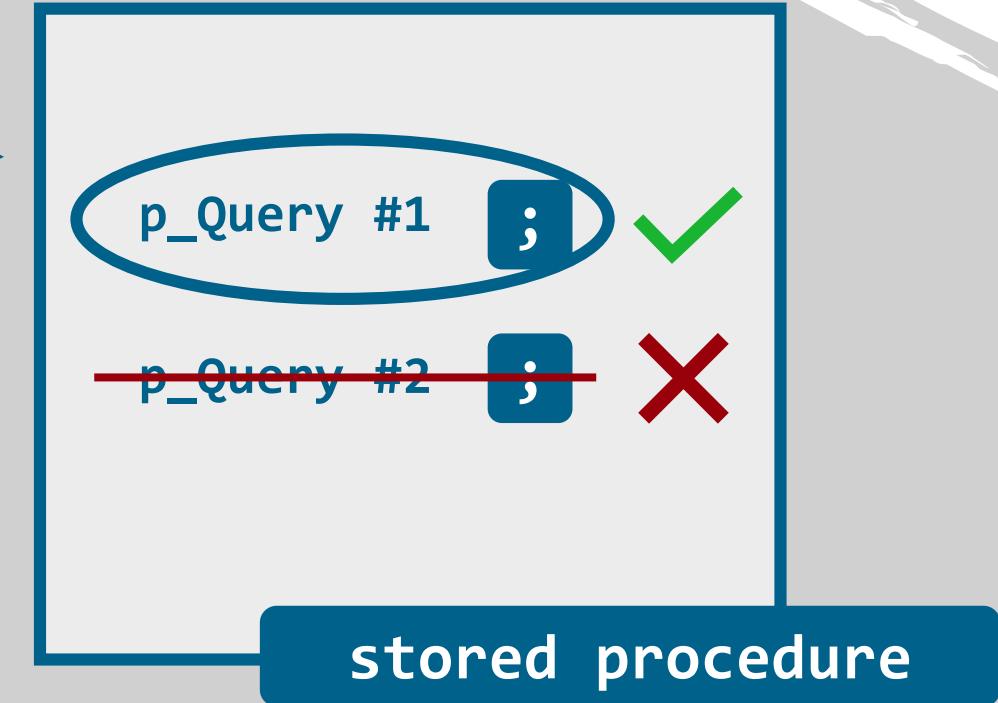
Query #2 ;

call stored\_procedure\_1 ;

Query #4 ;

Query #5 ;

... ;



# The MySQL Syntax for Stored Procedures

Query #1 ;

Query #2 ;

call stored\_procedure\_1 ;

\$\$ or //

DELIMITER

\$\$

p\_Query #1 ;

p\_Query #2 ;

stored procedure

# The MySQL Syntax for Stored Procedures

Query #1 ;

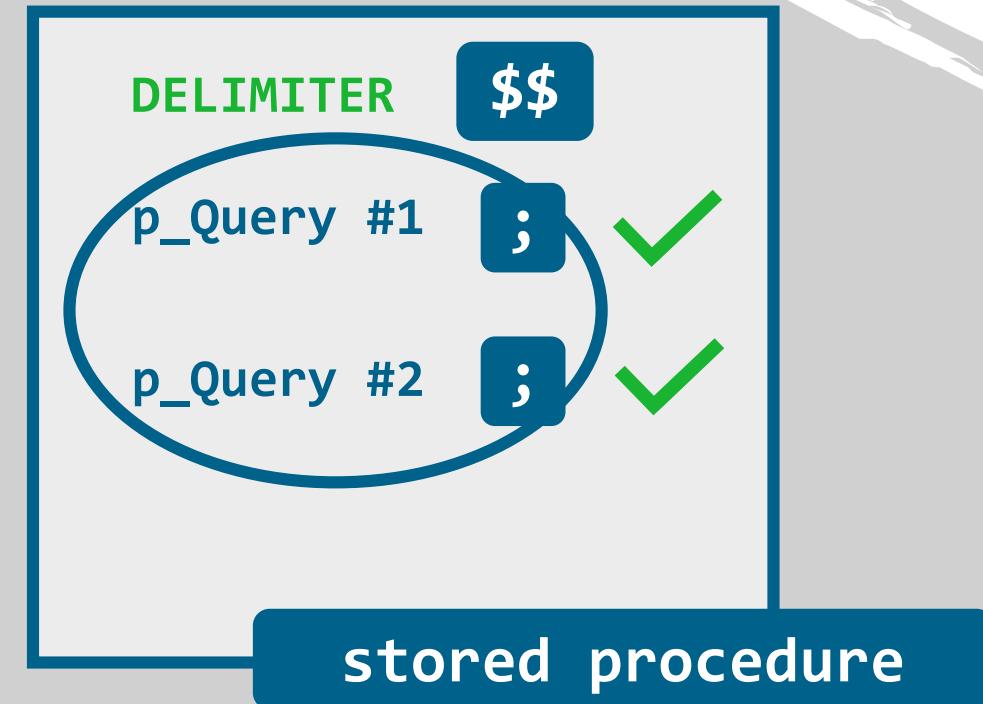
Query #2 ;

call stored\_procedure\_1 ;

Query #4 ;

Query #5 ;

... ;



# The MySQL Syntax for Stored Procedures



```
DELIMITER $$  
CREATE PROCEDURE procedure_name()
```

# The MySQL Syntax for Stored Procedures



DELIMITER \$\$

CREATE PROCEDURE procedure\_name(param\_1, param\_2)

*Parameters represent certain values that the procedure will use to complete the calculation it is supposed to execute*

# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()
```

*A procedure can be created without parameters too!*

*Nevertheless, the parentheses must always be attached to its name*

# The MySQL Syntax for Stored Procedures



```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```

# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```

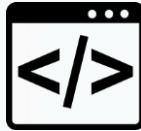
body of the procedure

# The MySQL Syntax for Stored Procedures



```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```

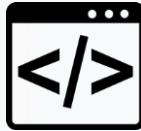
# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```

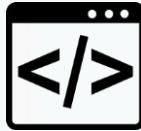
# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN
```

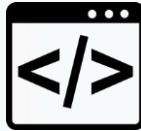
# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```

# The MySQL Syntax for Stored Procedures

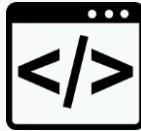


SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```

query

# The MySQL Syntax for Stored Procedures

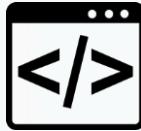


SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```



# The MySQL Syntax for Stored Procedures

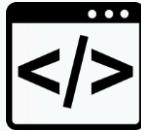


SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```



# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000$$  
END$$
```

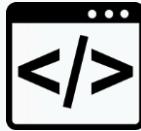
# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000$$ X  
END$$ ✓
```

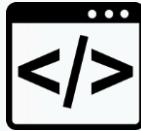
# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$
```

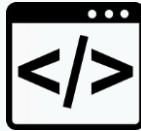
# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$  
DELIMITER ;
```

# The MySQL Syntax for Stored Procedures



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name()  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$  
DELIMITER ;
```

*From this moment on, \$\$ will not act as a delimiter*

A blurred background image of a modern conference room. The room features a long table with several black office chairs arranged around it. In the center of the room is a large, dark rectangular screen or presentation board. The room has a high ceiling with a grid of recessed lighting fixtures. Large windows on one side provide a view of an outdoor area with some greenery and a building across the street.

# Stored Procedures with an Input Parameter

# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value

# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an *input* value and then use it in the query, or queries, written in the body of the procedure

# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an *input* value and then use it in the query, or queries, written in the body of the procedure
  - this value is represented by the **IN** parameter

# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an *input* value and then use it in the query, or queries, written in the body of the procedure
  - this value is represented by the IN parameter

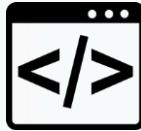
# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an *input* value and then use it in the query, or queries, written in the body of the procedure
  - this value is represented by the IN parameter

# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an input value and then use it in the query, or queries, written in the body of the procedure
  - this value is represented by the IN parameter

# Stored Procedures with an Input Parameter



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name(in parameter)  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$  
DELIMITER ;
```

# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an input value and then use it in the query, or queries, written in the body of the procedure
  - this value is represented by the IN parameter

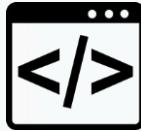
# Stored Procedures with an Input Parameter

- a stored routine can perform a calculation that transforms an *input* value in an *output* value
- stored procedures can take an input value and then use it in the query, or queries, written in the body of the procedure
  - this value is represented by the IN parameter
  - after that calculation is ready, a result will be returned

A blurred background image of a modern conference room. The room features a long table with several black office chairs arranged around it. In the center of the room is a large, dark rectangular screen or presentation board. The room has a high ceiling with a grid of recessed lighting fixtures. Large windows on one side provide a view of an outdoor area with some greenery and a building across the street.

# Stored Procedures with an Output Parameter

# Stored Procedures with an Output Parameter



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name(in parameter, out parameter)  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$  
DELIMITER ;
```

# Stored Procedures with an Output Parameter



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name(in parameter, out parameter)  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$  
DELIMITER ;
```

# Stored Procedures with an Output Parameter



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name(in parameter, out parameter)  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$  
DELIMITER ;
```



# Stored Procedures with an Output Parameter



SQL

```
DELIMITER $$  
CREATE PROCEDURE procedure_name(in parameter, out parameter)  
BEGIN  
    SELECT * FROM employees  
    LIMIT 1000;  
END$$  
DELIMITER ;
```

*it will represent the variable containing the output value of the operation executed by the query of the stored procedure*



# Stored Procedures with an Output Parameter

- every time you create a procedure containing both an IN and an OUT parameter, remember that you must use the SELECT-INTO structure in the query of this object's body!

A large conference room with rows of chairs facing a front wall with a large screen. The room has large windows overlooking a landscape.

# Variables

# Variables

- when you are *defining* a program, such as a stored procedure for instance, you can say you are using 'parameters'

# Variables

- when you are *defining* a program, such as a stored procedure for instance, you can say you are using 'parameters'
  - '*parameters*' are a more abstract term

# Variables

- when you are *defining* a program, such as a stored procedure for instance, you can say you are using 'parameters'
  - 'parameters' are a more abstract term



**DELIMITER \$\$**

```
CREATE PROCEDURE procedure_name (in , out )
```

SQL

# Variables

- when you are *defining* a program, such as a stored procedure for instance, you can say you are using 'parameters'
  - 'parameters' are a more abstract term



SQL

**DELIMITER \$\$**

```
CREATE PROCEDURE procedure_name (in parameter , out )
```

# Variables

- when you are *defining* a program, such as a stored procedure for instance, you can say you are using 'parameters'
  - '*parameters*' are a more abstract term



SQL

**DELIMITER \$\$**

**CREATE PROCEDURE** procedure\_name (**in** parameter, **out** parameter)

# Variables

- once the structure has been solidified, then it will be applied to the database. The input value you insert is typically referred to as the 'argument', while the obtained output value is stored in a 'variable'

# Variables

- once the structure has been solidified, then it will be applied to the database. The input value you insert is typically referred to as the 'argument', while the obtained output value is stored in a 'variable'

```
CREATE PROCEDURE ...
```

# Variables

- once the structure has been solidified, then it will be applied to the database. The input value you insert is typically referred to as the 'argument', while the obtained output value is stored in a 'variable'

input:

```
CREATE PROCEDURE ... argument
```

# Variables

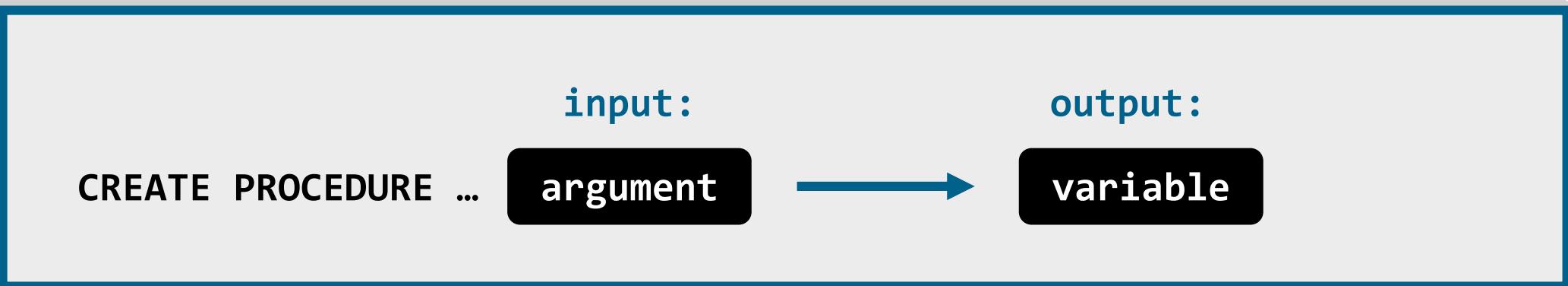
- once the structure has been solidified, then it will be applied to the database. The input value you insert is typically referred to as the 'argument', while the obtained output value is stored in a 'variable'

input:

CREATE PROCEDURE ... argument →

# Variables

- once the structure has been solidified, then it will be applied to the database. The input value you insert is typically referred to as the 'argument', while the obtained output value is stored in a 'variable'



# Variables



SQL

**DELIMITER \$\$**

CREATE PROCEDURE procedure\_name (in parameter , out parameter )

≠

CREATE PROCEDURE ...

input:

argument

output:

variable

# Variables

- IN-OUT parameters

```
CREATE PROCEDURE ...
```

# Variables

- IN-OUT parameters

input:

CREATE PROCEDURE ...    IN parameter

# Variables

- IN-OUT parameters

input:

CREATE PROCEDURE ...

IN parameter



# Variables

- IN-OUT parameters

```
CREATE PROCEDURE ...
```

input:

OUT parameter



# Variables

- IN-OUT parameters

```
CREATE PROCEDURE ...
```

`input = output`

`OUT parameter`





# User-Defined Functions in MySQL

# User-Defined Functions in MySQL



SQL

```
DELIMITER $$  
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type  
DECLARE variable_name data_type  
- BEGIN  
    SELECT ...  
    RETURN variable_name  
END$$  
DELIMITER ;
```

# User-Defined Functions in MySQL



SQL

```
DELIMITER $$  
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type  
DECLARE variable_name data_type  
- BEGIN  
    SELECT ...  
    RETURN variable_name  
END$$  
DELIMITER ;
```

↑  
*here you have no OUT parameters to define between the parentheses after the object's name*

# User-Defined Functions in MySQL



SQL

```
DELIMITER $$  
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type  
DECLARE variable_name data_type  
- BEGIN  
    SELECT ...  
    RETURN variable_name  
END$$  
DELIMITER ;
```

↑

*here you have no OUT parameters to define between the parentheses after the object's name*

*all parameters are IN, and since this is well known, you need not explicitly indicate it with the word, 'IN'*

# User-Defined Functions in MySQL



SQL

```
DELIMITER $$  
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type  
DECLARE variable_name data_type  
- BEGIN  
    SELECT ...  
    RETURN variable_name ← although there are no OUT parameters, there is a 'return value'  
END$$  
DELIMITER ;
```

# User-Defined Functions in MySQL



```
DELIMITER $$
```

```
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type
```

```
DECLARE variable_name data_type
```

```
- BEGIN
```

```
    SELECT ...
```

```
    RETURN variable_name
```

```
END$$
```

```
DELIMITER ;
```

*although there are no OUT parameters, there is a 'return value'*

*it is obtained after running the query contained in the body of the function*

# User-Defined Functions in MySQL



```
DELIMITER $$  
CREATE FUNCTION function_name(parameter data_type) RETURNS data_type  
DECLARE variable_name data_type  
- BEGIN  
    SELECT ...  
    RETURN variable_name  
END$$  
DELIMITER ;
```

*it can be of any data type*

# User-Defined Functions in MySQL

- we cannot *call* a function!

# User-Defined Functions in MySQL

- we cannot *call* a function!
- we can *select* it, indicating an input value within parentheses

# User-Defined Functions in MySQL

- we cannot *call* a function!
- we can *select* it, indicating an input value within parentheses



SQL

```
SELECT function_name(input_value);
```

A large conference room with rows of chairs facing a front wall with presentation screens. The room has a modern design with a grid ceiling and large windows overlooking a landscape.

# Stored Routines - Conclusion

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

stored procedure

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

stored procedure

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

stored procedure

user-defined  
function

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

stored procedure

user-defined  
function

returns a value

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

stored procedure

does not return a  
value

user-defined  
function

returns a value

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

stored procedure	user-defined function
does not return a value	returns a value
<code>CALL procedure;</code>	

# Stored Routines - Conclusion

## TECHNICAL DIFFERENCES

stored procedure	user-defined function
does not return a value	returns a value
<code>CALL procedure;</code>	<code>SELECT function;</code>

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure

user-defined  
function

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure

user-defined  
function

can have *multiple OUT*  
parameters

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure

user-defined  
function

can have *multiple* OUT  
parameters

can return a *single*  
value only

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure	user-defined function
can have <i>multiple</i> OUT parameters	can return a <i>single</i> value only

- if you need to obtain more than one value as a result of a calculation, you are better off *using a procedure*

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure	user-defined function
can have <i>multiple OUT parameters</i>	can return a <i>single value only</i>

- if you need to obtain more than one value as a result of a calculation, you are better off *using a procedure*
- if you need to just one value to be returned, then you can *use a function*

# Stored Routines - Conclusion

- how about involving an INSERT, an UPDATE, or a DELETE statement?

# Stored Routines - Conclusion

- how about involving an INSERT, an UPDATE, or a DELETE statement?
  - in those cases, the operation performed will apply changes to the data in your database

# Stored Routines - Conclusion

- how about involving an INSERT, an UPDATE, or a DELETE statement?
  - in those cases, the operation performed will apply changes to the data in your database
  - there will be no value, or values, to be returned and displayed to the user

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure

can have *multiple OUT parameters*

user-defined function

can return a *single value only*

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure

can have *multiple* OUT  
parameters

user-defined  
function

can return a *single*  
value only

INSERT

UPDATE

DELETE

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure

can have *multiple* OUT  
parameters

user-defined  
function

can return a *single*  
value only

INSERT  UPDATE  
DELETE

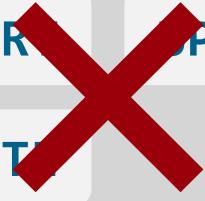
# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure	user-defined function
can have <i>multiple</i> OUT parameters	can return a <i>single</i> value only
INSERT UPDATE DELETE	INSERT <del>UPDATE</del> <del>DELETE</del>

# Stored Routines - Conclusion

## CONCEPTUAL DIFFERENCES

stored procedure	user-defined function
can have <i>multiple</i> OUT parameters	can return a <i>single</i> value only
 <b>INSERT</b> <b>UPDATE</b> <b>DELETE</b>	<b>INSERT</b>  <b>UPDATE</b> <b>DELETE</b>

# Stored Routines - Conclusion

stored procedure

user-defined  
function

# Stored Routines - Conclusion

stored procedure

user-defined  
function

---

TECHNICAL DIFFERENCE

# Stored Routines - Conclusion

stored procedure

user-defined  
function

## TECHNICAL DIFFERENCE

CALL procedure;

SELECT function;

# Stored Routines - Conclusion

stored procedure

user-defined  
function

## TECHNICAL DIFFERENCE

`CALL` procedure;

`SELECT` function;

## CONCEPTUAL DIFFERENCE

# Stored Routines - Conclusion

stored procedure

user-defined  
function

## TECHNICAL DIFFERENCE

`CALL` procedure;

`SELECT` function;

## CONCEPTUAL DIFFERENCE

- you can easily include a function as one of the columns inside a `SELECT` statement

# Stored Routines - Conclusion

stored procedure

user-defined  
function

## TECHNICAL DIFFERENCE

`CALL` procedure;

`SELECT` function;

## CONCEPTUAL DIFFERENCE

- including a procedure in a SELECT statement is impossible

- you can easily include a function as one of the columns inside a SELECT statement

## Stored Routines - Conclusion

- once you become an advanced SQL user, and have gained a lot of practice, you will appreciate the advantages and disadvantages of both types of programs

# Stored Routines - Conclusion

- once you become an advanced SQL user, and have gained a lot of practice, you will appreciate the advantages and disadvantages of both types of programs
  - you will encounter many cases where you should choose between procedures and functions

## Stored Routines - Conclusion

- what we did in this section was to lay the foundation of the relevant syntax, as well as performing exercises on the practical aspects of these tools

A large, modern conference room with rows of chairs facing a central presentation area. The room has a high ceiling with recessed lighting and large windows in the background. The overall atmosphere is professional and spacious.

# Advanced SQL Topics

# Types of MySQL Variables – Local Variables

# Types of MySQL Variables – Local Variables

## scope

the region of a computer program where a phenomenon, such as a variable, is considered valid

# Types of MySQL Variables – Local Variables

## scope

the region of a computer program where a phenomenon, such as a variable, is considered valid

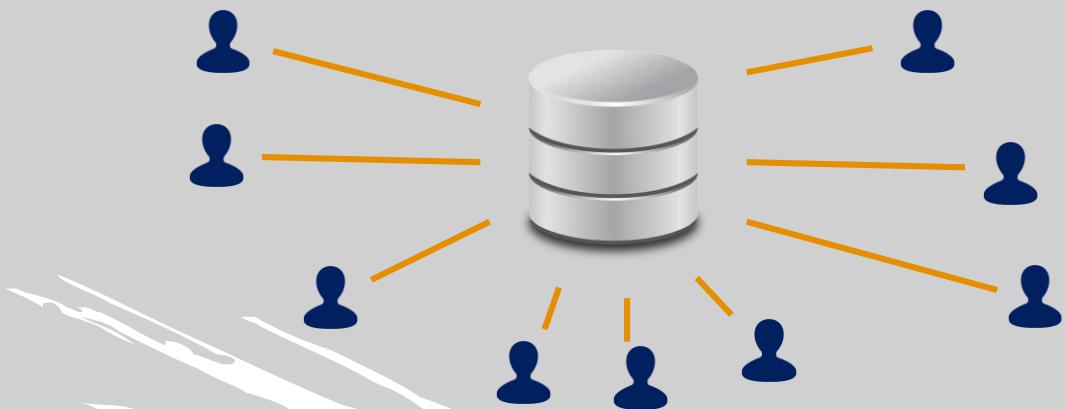
```
BEGIN  
    DECLARE v_avg_salary DECIMAL(10,2);  
    SELECT  
        AVG(s.salary)  
    INTO v_avg_salary FROM  
        employees e
```

# Types of MySQL Variables – Local Variables

## scope

the region of a computer program where a phenomenon, such as a variable, is considered valid

```
BEGIN  
DECLARE v_avg_salary DECIMAL(10,2);  
SELECT  
    AVG(s.salary)  
INTO v_avg_salary FROM  
employees e
```

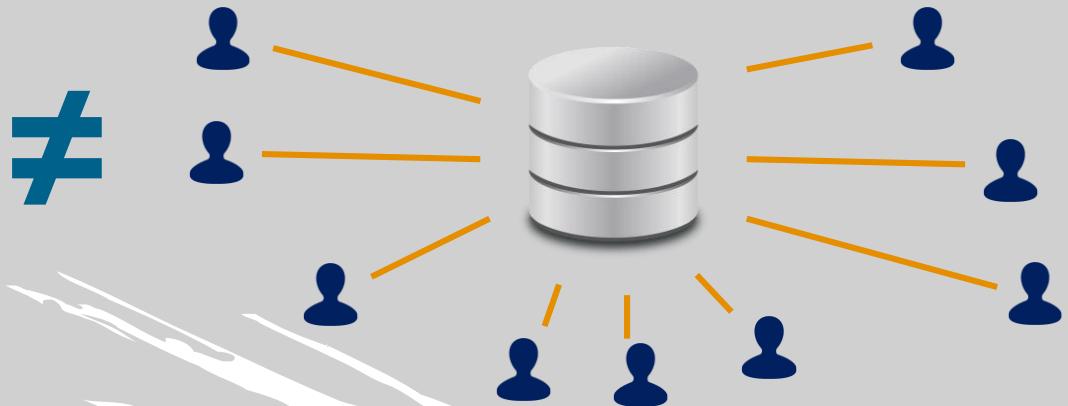


# Types of MySQL Variables – Local Variables

## scope

the region of a computer program where a phenomenon, such as a variable, is considered valid

```
BEGIN  
DECLARE v_avg_salary DECIMAL(10,2);  
SELECT  
    AVG(s.salary)  
INTO v_avg_salary FROM  
    employees e
```

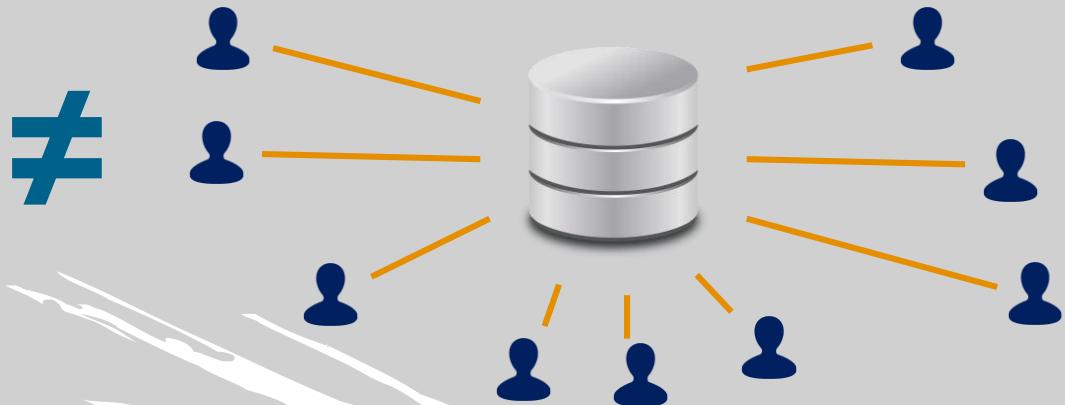


# Types of MySQL Variables – Local Variables

## scope

the region of a computer program where a phenomenon, such as a variable, is considered valid

```
BEGIN  
DECLARE v_avg_salary DECIMAL(10,2);  
SELECT  
    AVG(s.salary)  
INTO v_avg_salary FROM  
    employees e
```

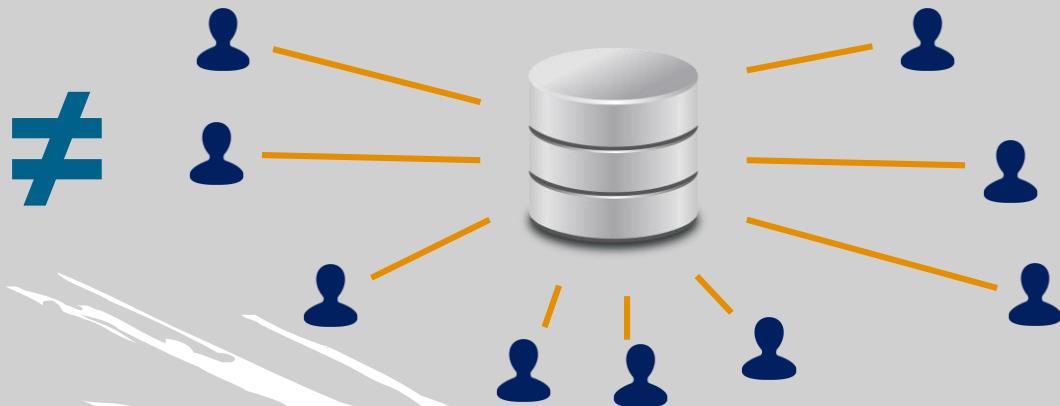


# Types of MySQL Variables – Local Variables

scope =

the region of a computer program where a phenomenon, such as a variable, is considered valid

```
BEGIN  
DECLARE v_avg_salary DECIMAL(10,2);  
SELECT  
    AVG(s.salary)  
INTO v_avg_salary FROM  
    employees e
```

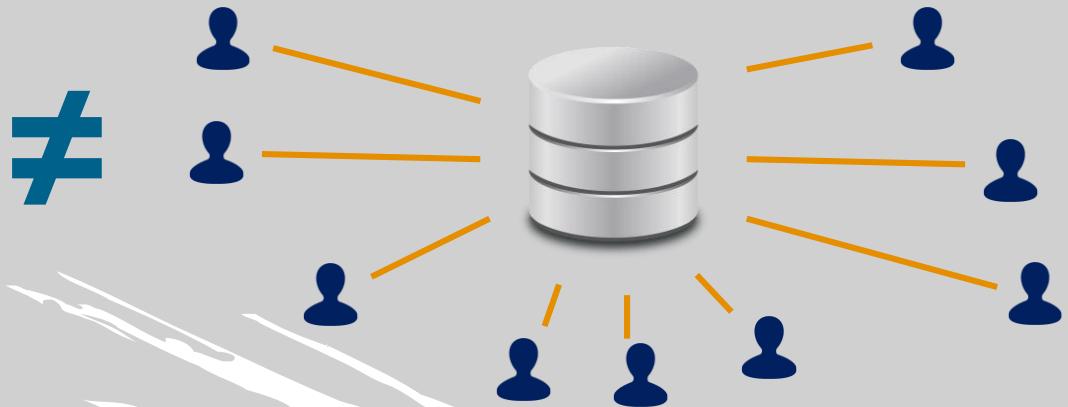


# Types of MySQL Variables – Local Variables

scope = visibility

the region of a computer program where a phenomenon, such as a variable, is considered valid

```
BEGIN  
DECLARE v_avg_salary DECIMAL(10,2);  
SELECT  
    AVG(s.salary)  
INTO v_avg_salary FROM  
    employees e
```

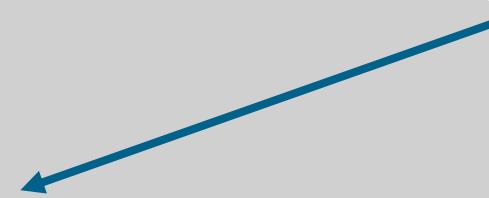


# Types of MySQL Variables – Local Variables

MySQL Variables

# Types of MySQL Variables – Local Variables

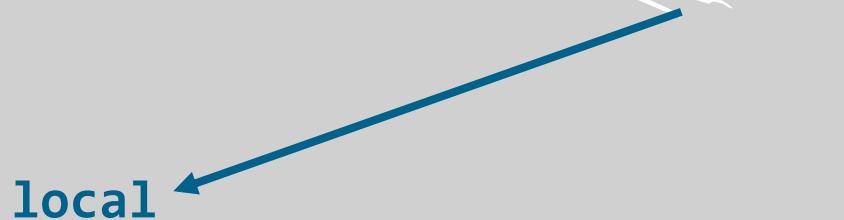
MySQL Variables



# Types of MySQL Variables – Local Variables

MySQL Variables

local

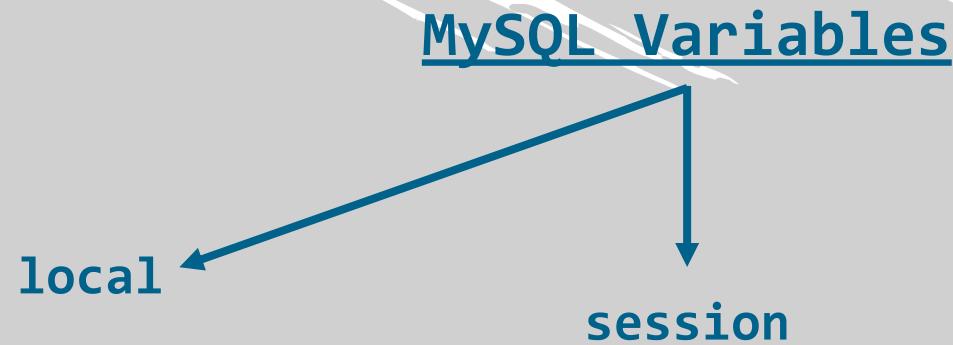


# Types of MySQL Variables – Local Variables

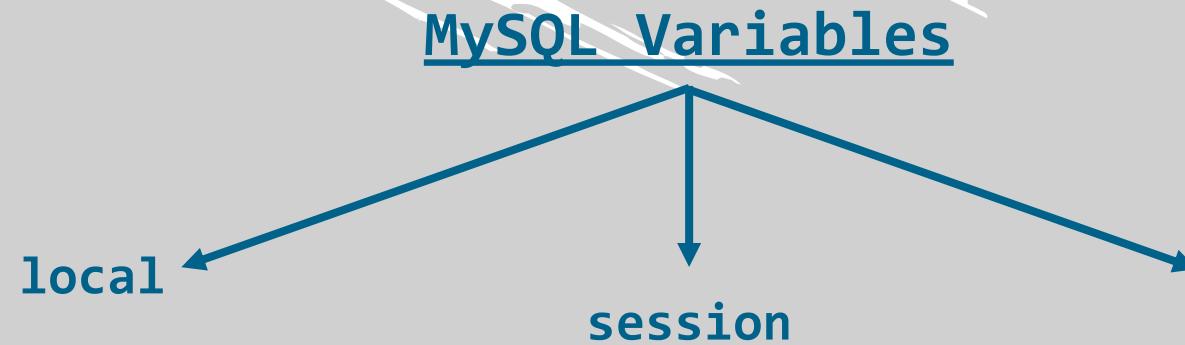
## MySQL Variables

local

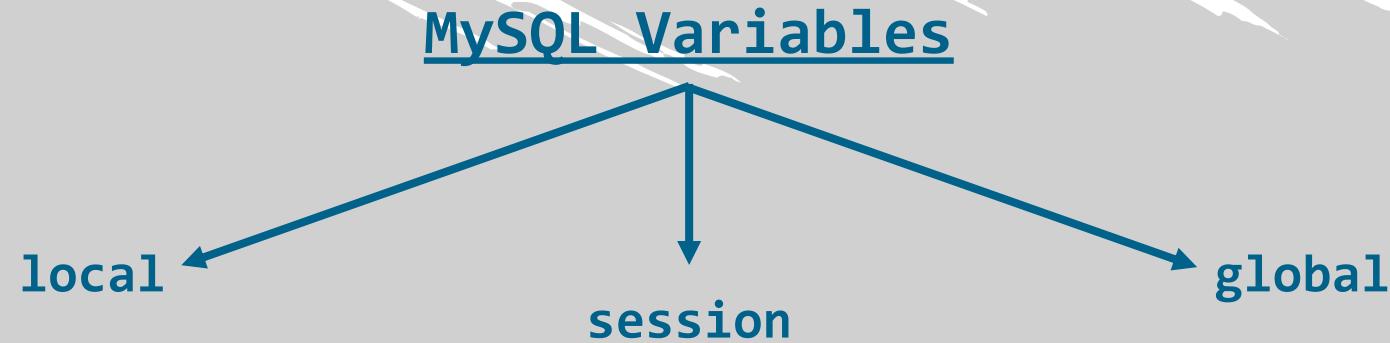
# Types of MySQL Variables – Local Variables



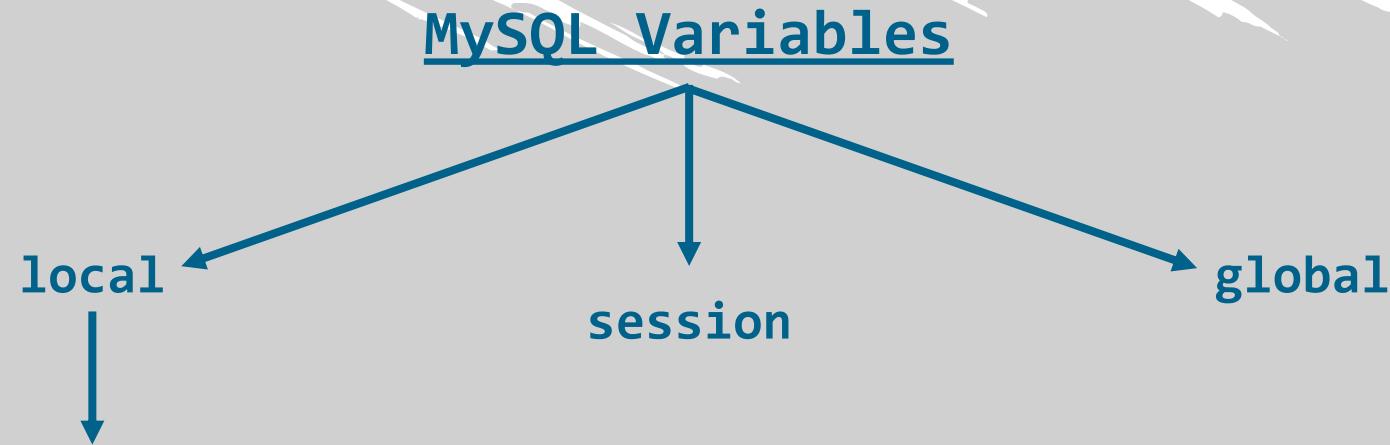
# Types of MySQL Variables – Local Variables



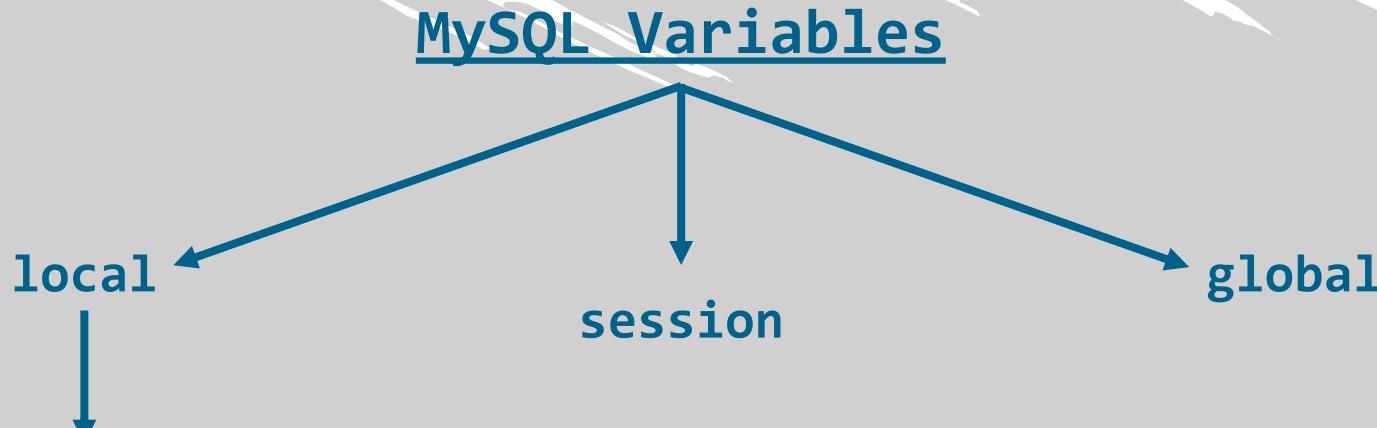
# Types of MySQL Variables – Local Variables



# Types of MySQL Variables – Local Variables



# Types of MySQL Variables – Local Variables



## local variable

a variable that is visible only in the **BEGIN – END** block in which it was created

# Types of MySQL Variables – Local Variables

- DECLARE is a keyword that can be used when creating *Local* variables only
- *v\_avg\_salary* is visible only in the BEGIN - END block

A large conference room with rows of chairs facing a central table. The room has a modern design with a glass partition and a large window overlooking a landscape. The chairs are arranged in several rows, pointing towards the front of the room where a table is set up.

# Session Variables

# Session Variables

## session

a series of information exchange interactions, or a dialogue, between a computer and a user

# Session Variables

## session

a series of information exchange interactions, or a dialogue, between a computer and a user

- e.g. a dialogue between the MySQL server and a client application like MySQL Workbench

# Session Variables

- *a session begins at a certain point in time and terminates at another, later point*

# Session Variables

- *a session begins at a certain point in time and terminates at another, later point*

# Session Variables

- a *session* begins at a certain point in time and terminates at another, later point

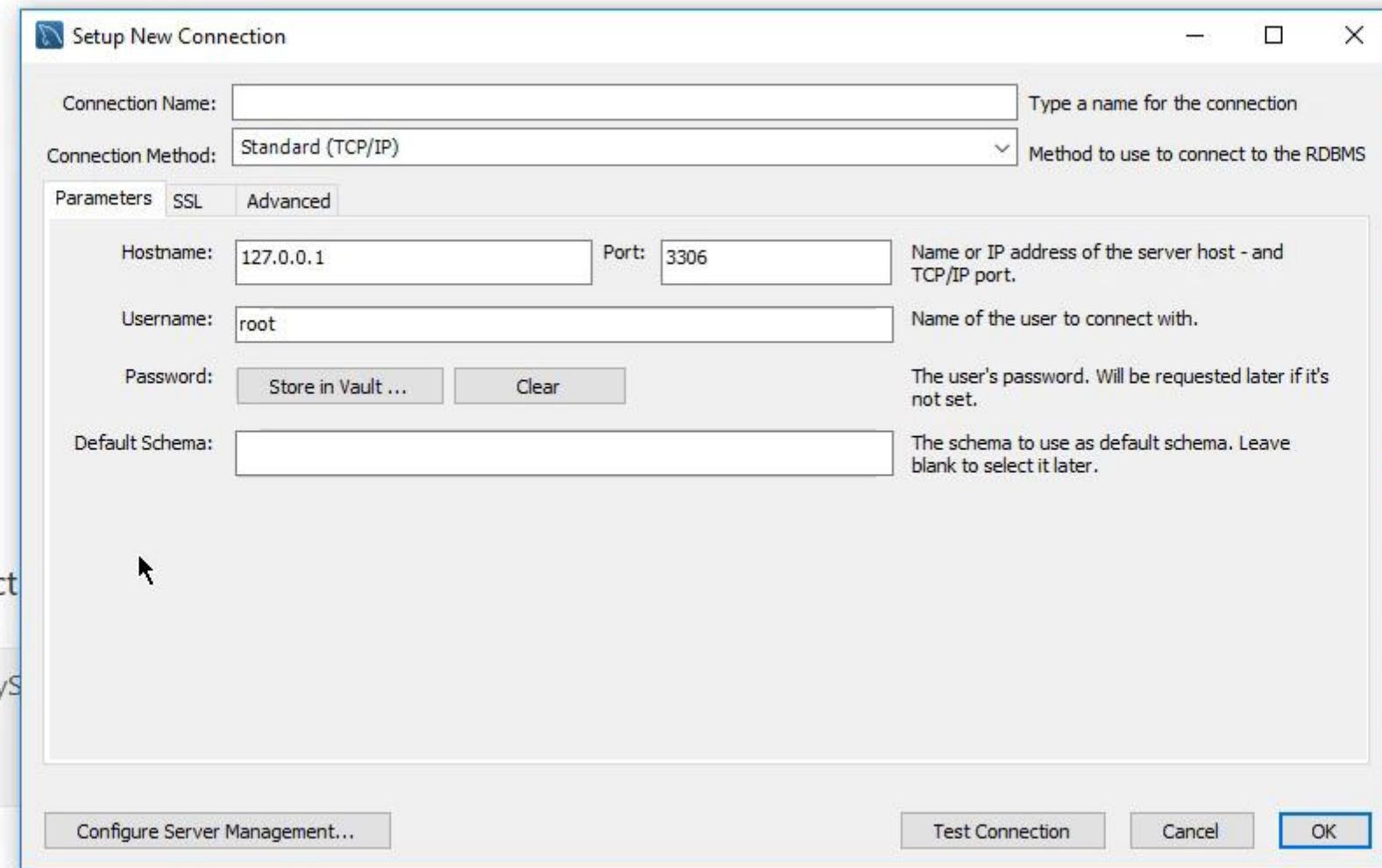
## Step 1



set up a connection



# Step 1



# Session Variables

- a *session* begins at a certain point in time and terminates at another, later point

## Step 1



set up a connection



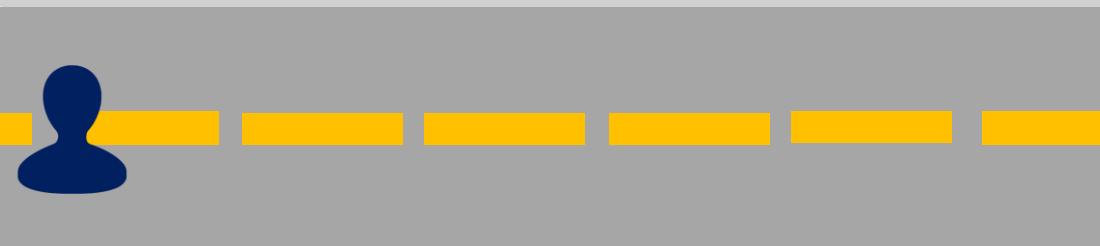
# Session Variables

- a *session* begins at a certain point in time and terminates at another, later point

## Step 1



set up a connection



## Step 2

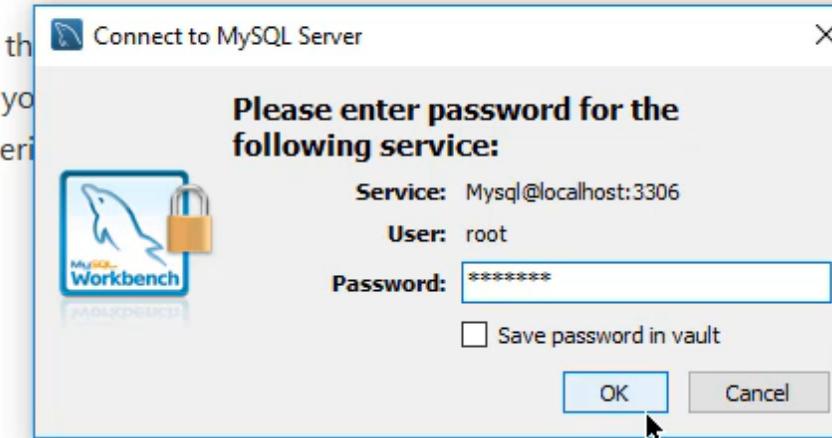
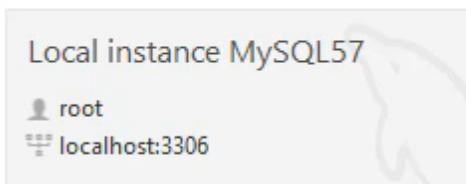
# Welcome to MySQL Workbench

MySQL Workbench is the easiest way to manage MySQL. It allows you to create and browse your databases, design and run SQL queries, and more.

[Browse Documentation >](#)

MySQL Connections  

 [Filter connections](#)



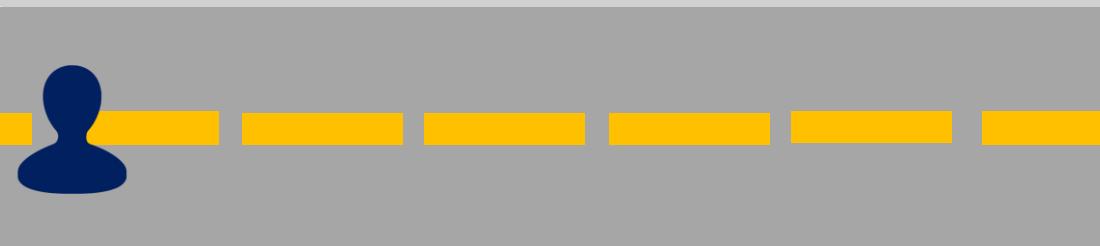
# Session Variables

- a *session* begins at a certain point in time and terminates at another, later point

## Step 1



set up a connection



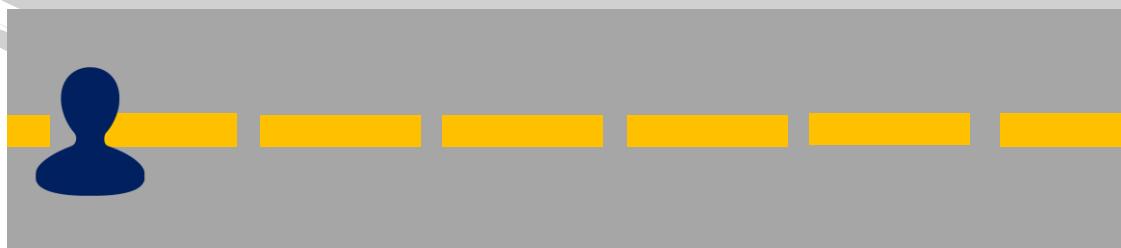
# Session Variables

- a *session* begins at a certain point in time and terminates at another, later point

Step 1



set up a connection



Step 2



establish a connection

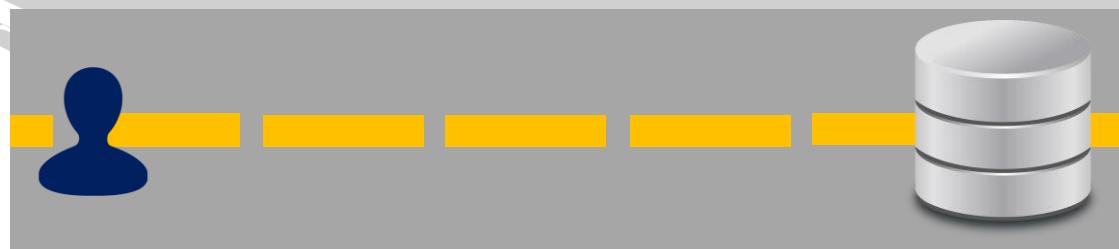
# Session Variables

- a *session* begins at a certain point in time and terminates at another, later point

Step 1



set up a connection



Step 2



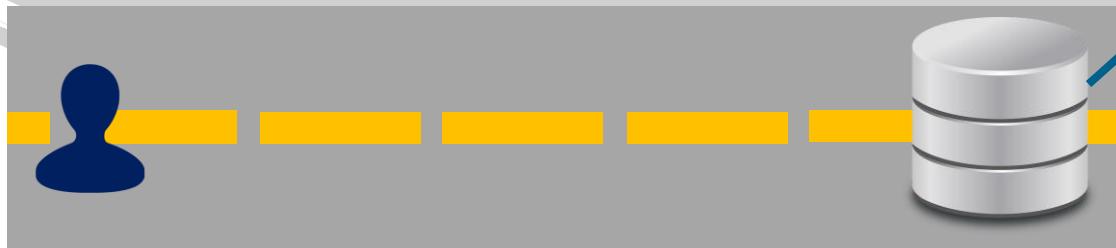
establish a connection

# Session Variables

- a *session* begins at a certain point in time and terminates at another, later point

Step 1

set up a connection



Step 2

establish a connection

Step 3

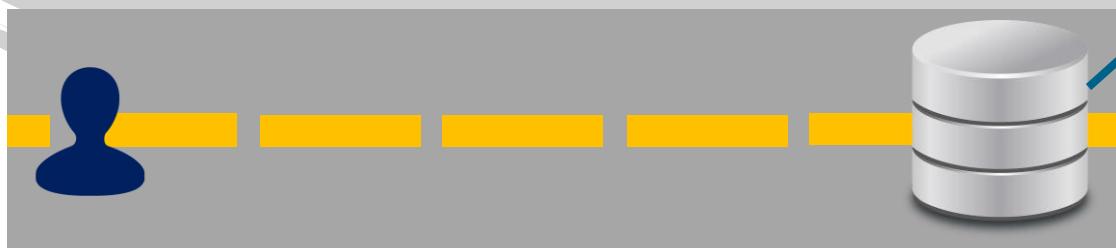
the Workbench interface will open immediately

# Session Variables

- a session begins at a certain point in time and terminates at another, later point

Step 1

set up a connection



Step 2

establish a connection

Step 3

the Workbench interface will open immediately

**MySQL Workbench**

Local instance MySQL57 ×

File Edit View Query Database Server Tools Scripting Help

SQl SQL Databases Tables Views Functions Triggers Events Scripts Reports

Navigator

**MANAGEMENT**

- Server Status
- Client Connections
- Users and Privileges
- Status and System Variables
- Data Export
- Data Import/Restore

**INSTANCE**

- Startup / Shutdown
- Server Logs
- Options File

**PERFORMANCE**

- Dashboard
- Performance Reports
- Performance Schema Setup

**SCHEMAS**

Filter objects

sys

Information

No object selected

Object Info Session

**Step 3**

SQLAdditions

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

Context Help Snippets

Output

Action Output

# Time Action Message Duration / Fetch

365 Careers

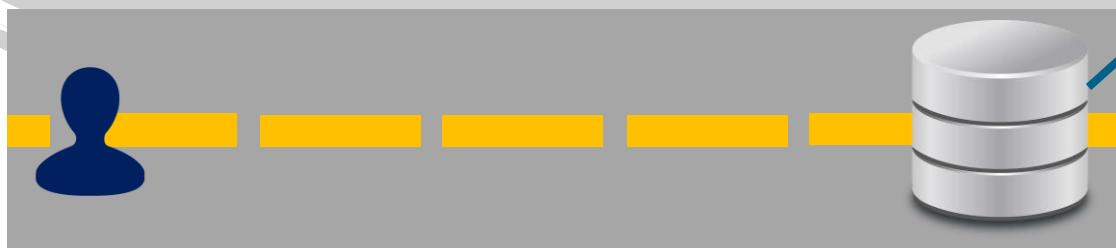
This screenshot shows the MySQL Workbench interface. The main window has several tabs: Navigator, MANAGEMENT, INSTANCE, PERFORMANCE, SCHEMAS, and Information. The SCHEMAS tab is active, showing a list of schemas including 'sys'. The bottom left shows 'Object Info' and 'Session' tabs. A large watermark 'Step 3' is overlaid at the top center. The bottom right corner features the '365 Careers' logo. The SQLAdditions panel displays a message: 'Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.' The bottom output panel shows an 'Action Output' table with columns for #, Time, Action, Message, and Duration / Fetch.

# Session Variables

- a session begins at a certain point in time and terminates at another, later point

Step 1

set up a connection



Step 2

establish a connection

Step 3

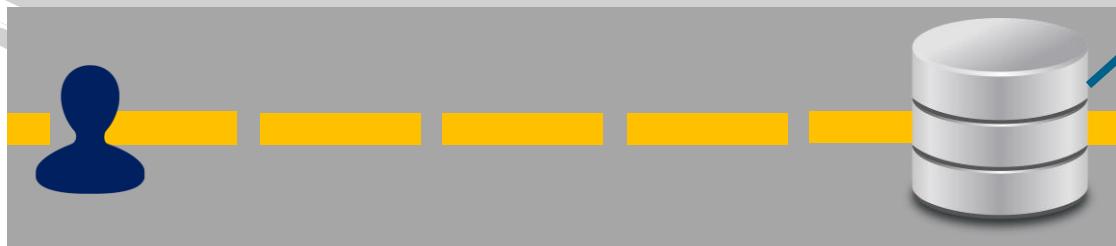
the Workbench interface will open immediately

# Session Variables

- a session begins at a certain point in time and terminates at another, later point

Step 1

set up a connection



Step 2

establish a connection

Step 3

the Workbench interface will open immediately

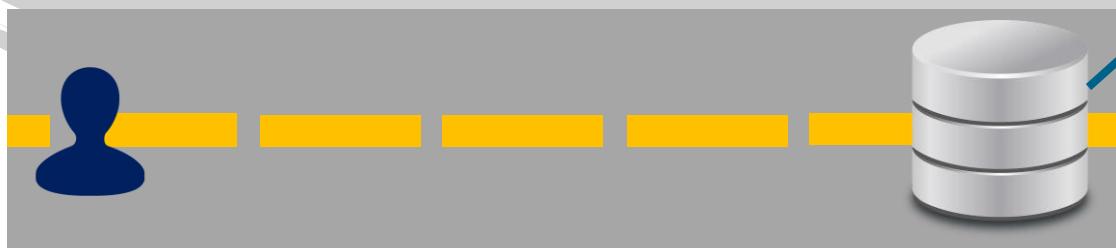


# Session Variables

- a session begins at a certain point in time and terminates at another, later point

Step 1

set up a connection



Step 2

establish a connection

Step 3

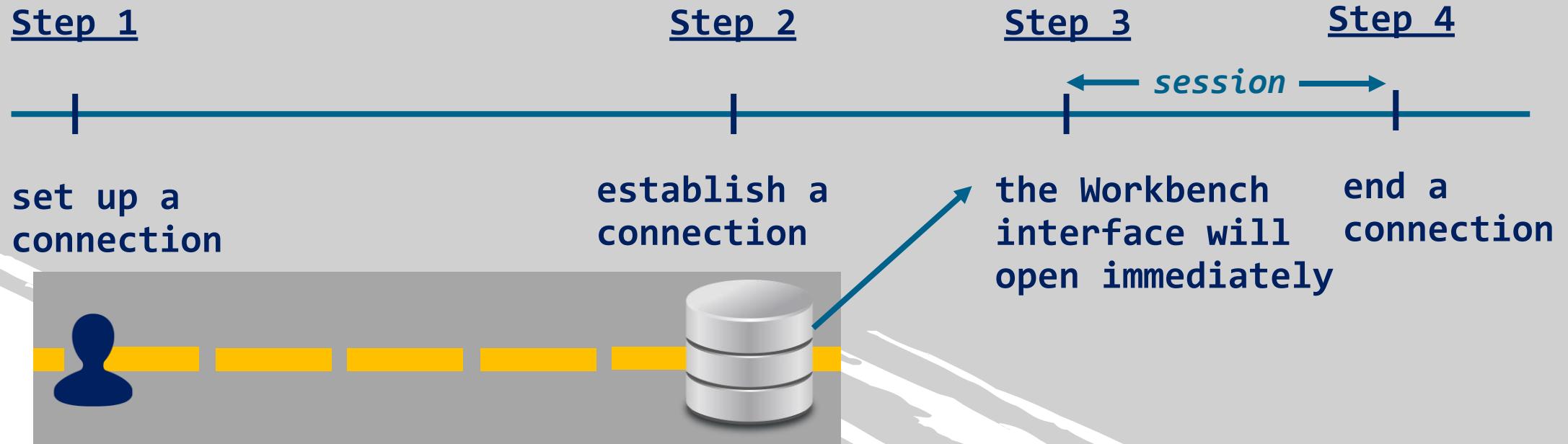
the Workbench interface will open immediately

Step 4



# Session Variables

- a session begins at a certain point in time and terminates at another, later point

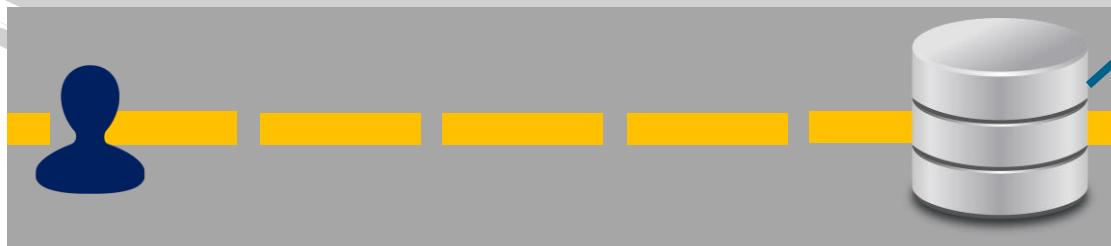


# Session Variables

- a session begins at a certain point in time and terminates at another, later point

Step 1

set up a connection



Step 2

establish a connection

Step 3

the Workbench interface will open immediately

Step 4

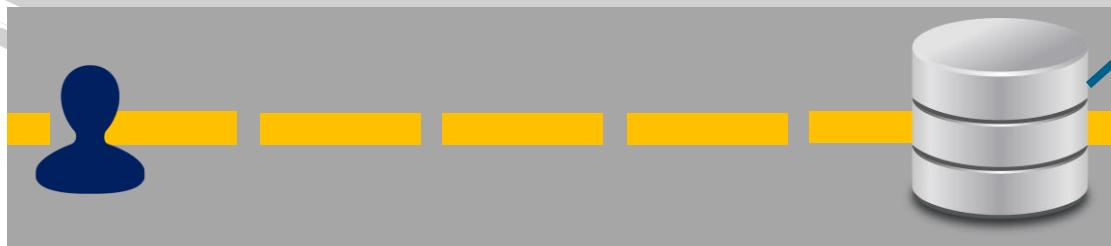
end a connection

# Session Variables

- a session begins at a certain point in time and terminates at another, later point

Step 1

set up a connection



Step 2

establish a connection

Step 3

the Workbench interface will open immediately

Step 4

# Session Variables

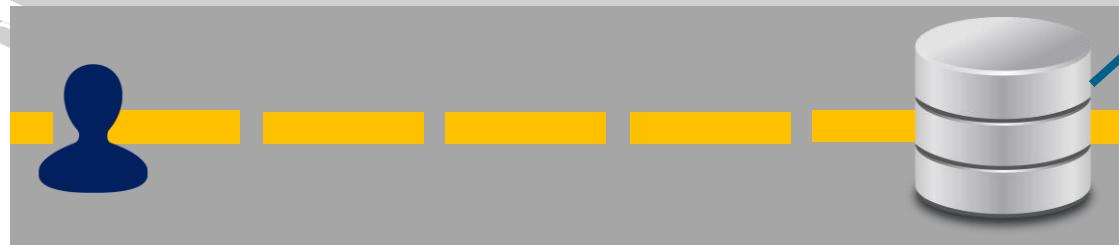
- there are certain SQL objects that are valid for a specific session only

# Session Variables

- there are certain SQL objects that are valid for a specific session only

Step 1

set up a connection



Step 2

establish a connection

Step 3

the Workbench interface will open immediately



# Session Variables

- there are certain SQL objects that are valid for a specific session only

Step 1

set up a connection



Step 2

establish a connection

emp_no	from_date	to_date
10001	1986-06-26	9999-01-01
10002	1996-08-03	9999-01-01
10003	1995-12-03	9999-01-01
10004	1986-12-01	9999-01-01
10005	1989-09-12	9999-01-01



the Workbench interface will open immediately

# Session Variables

- there are certain SQL objects that are valid for a specific session only

Step 1

set up a connection



Step 2

establish a connection

emp_no	from_date	to_date
10001	1986-06-26	9999-01-01
10002	1996-08-03	9999-01-01
10003	1995-12-03	9999-01-01
10004	1986-12-01	9999-01-01
10005	1989-09-12	9999-01-01

← session →

the Workbench interface will open immediately

# Session Variables

- there are certain SQL objects that are valid for a specific session only

Step 1

set up a connection



Step 2

establish a connection

emp_no	from_date	to_date
10001	1986-06-12	9999-01-01
10002	1989-01-12	9999-01-01
10003	1989-01-12	9999-01-01
10004	1989-09-12	9999-01-01
10005	1989-09-12	1999-01-01

the Workbench interface will open immediately

# Session Variables

**session variable**

a variable that exists only for the session in which you are operating

# Session Variables

## session variable

a variable that exists only for the session in which you are operating

- it is *defined* on our server, and it lives there

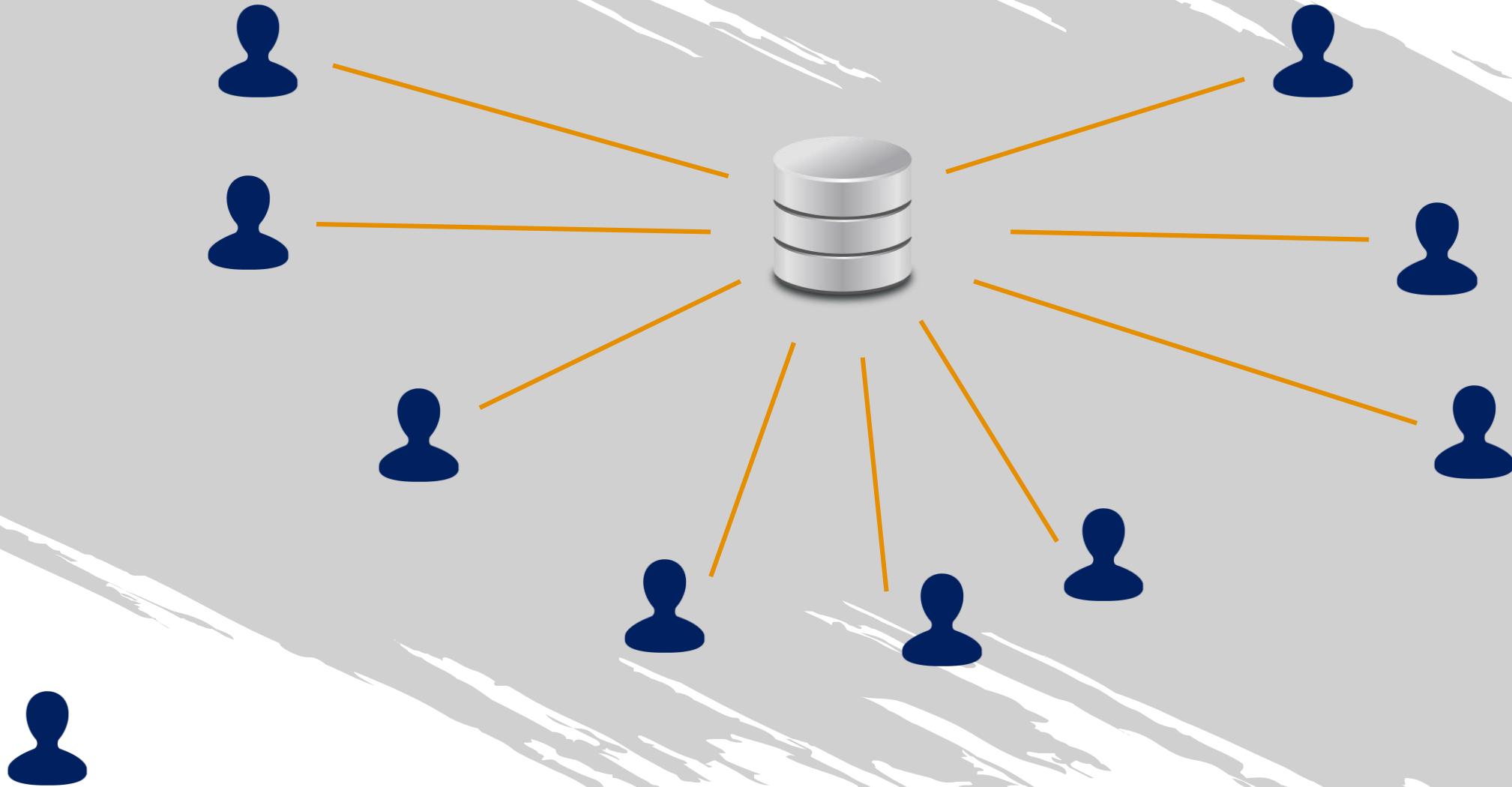
# Session Variables

## session variable

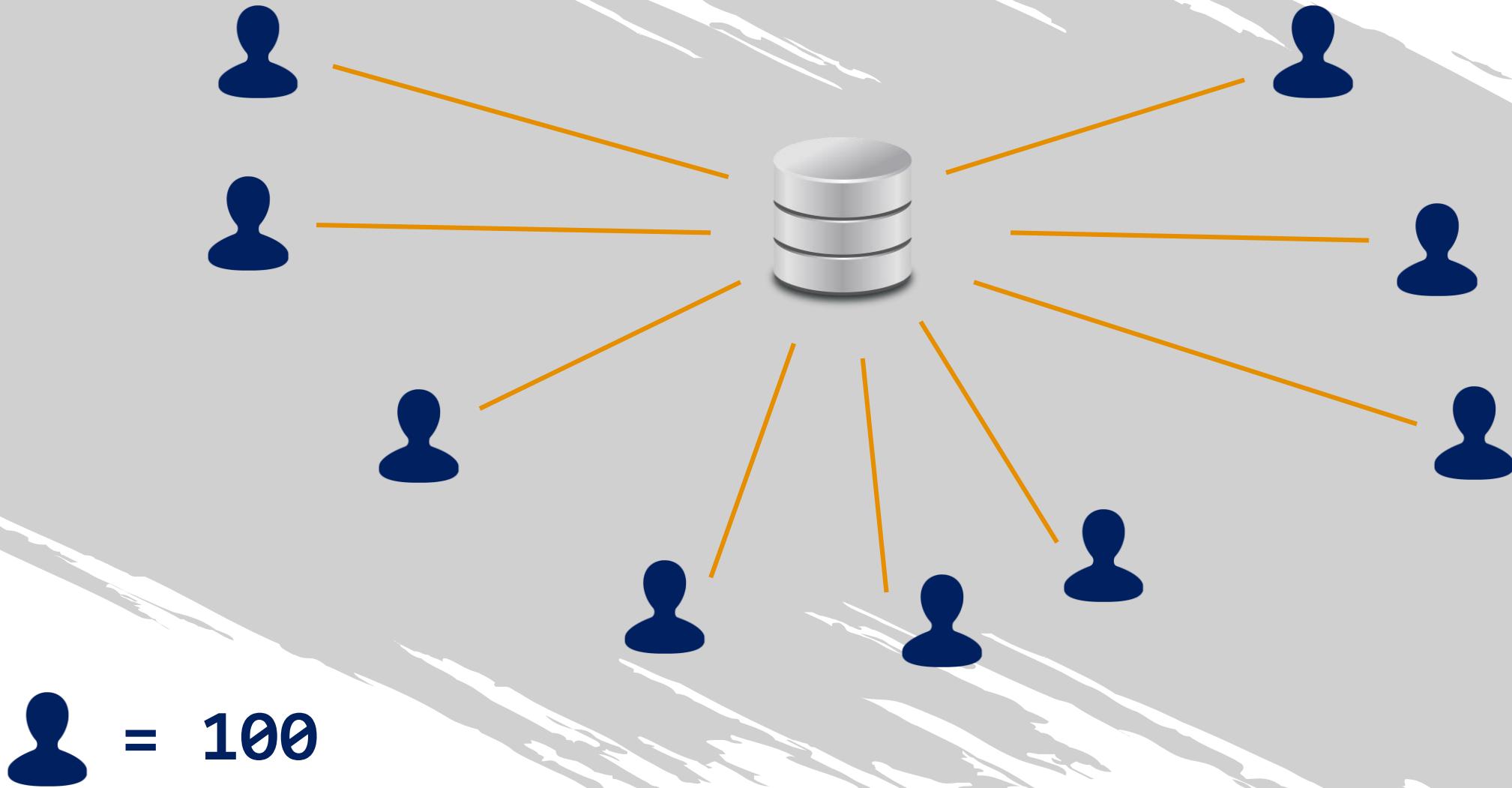
a variable that exists only for the session in which you are operating

- it is *defined* on our server, and it lives there
- it is *visible* to the connection being used only

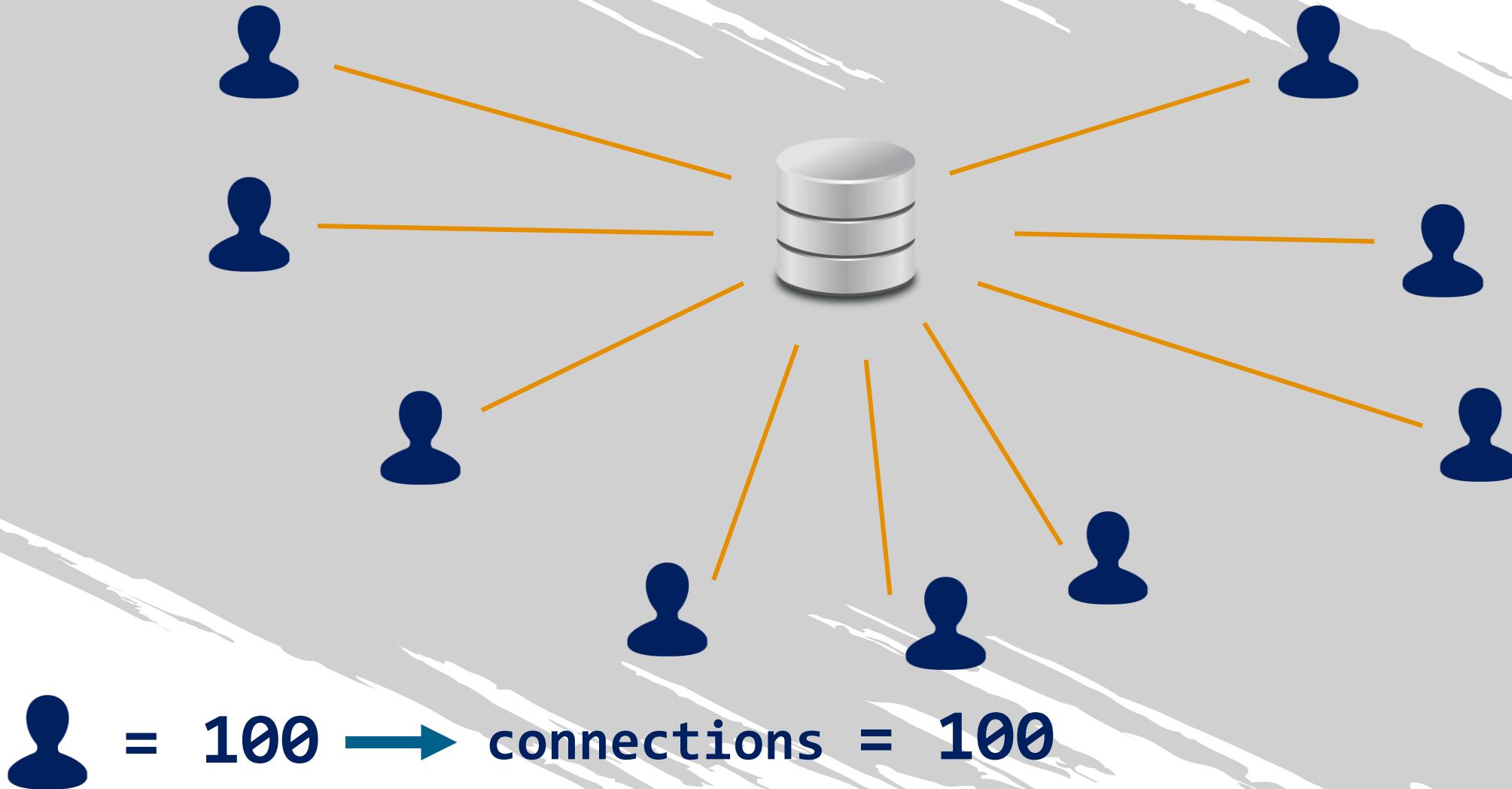
# Session Variables



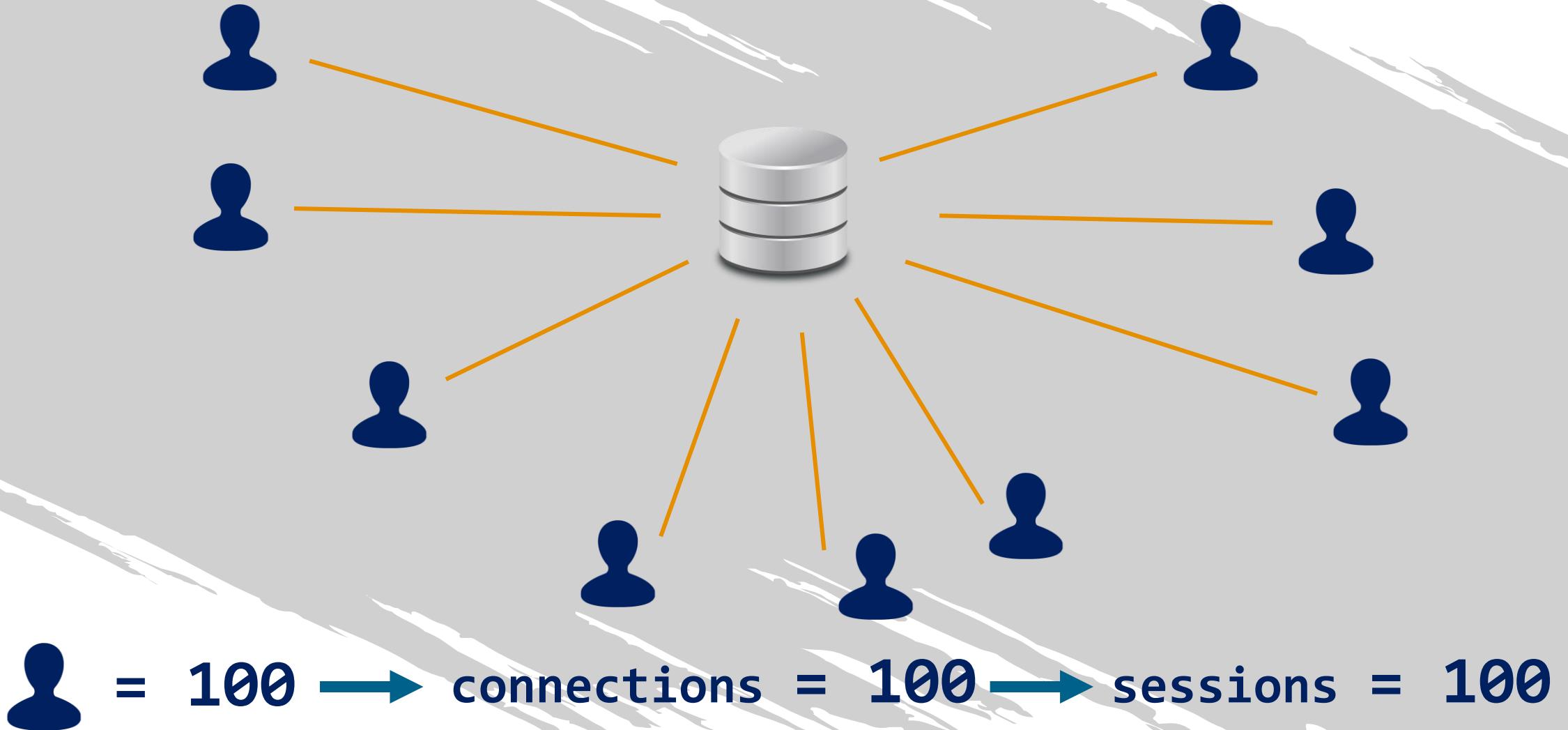
# Session Variables



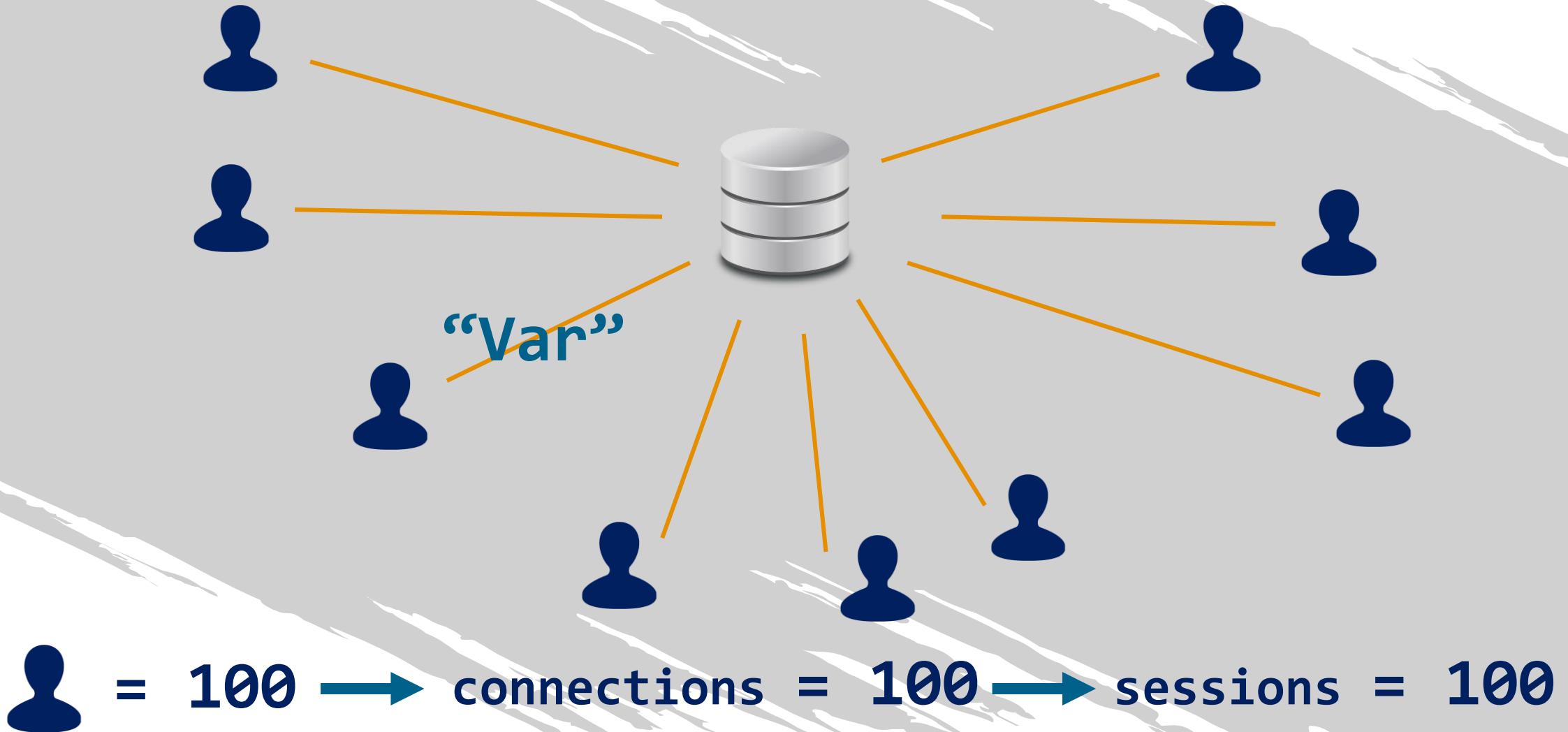
# Session Variables



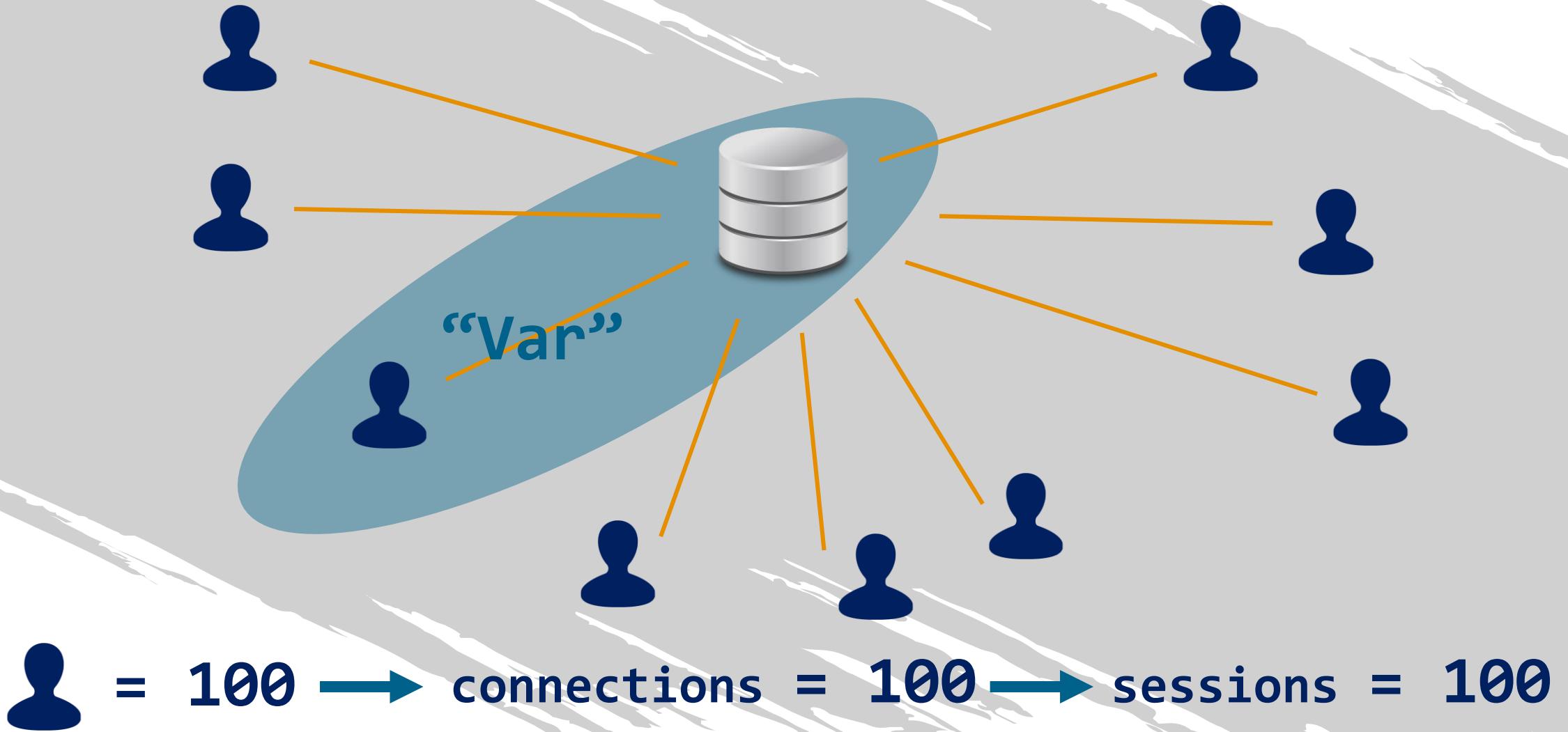
# Session Variables



# Session Variables



# Session Variables



# Session Variables



SQL

```
SET @var_name = value;
```

A large conference room with rows of chairs facing a central table. The room has a modern design with a glass partition and large windows overlooking a landscape. The chairs are arranged in several rows, pointing towards the center of the room.

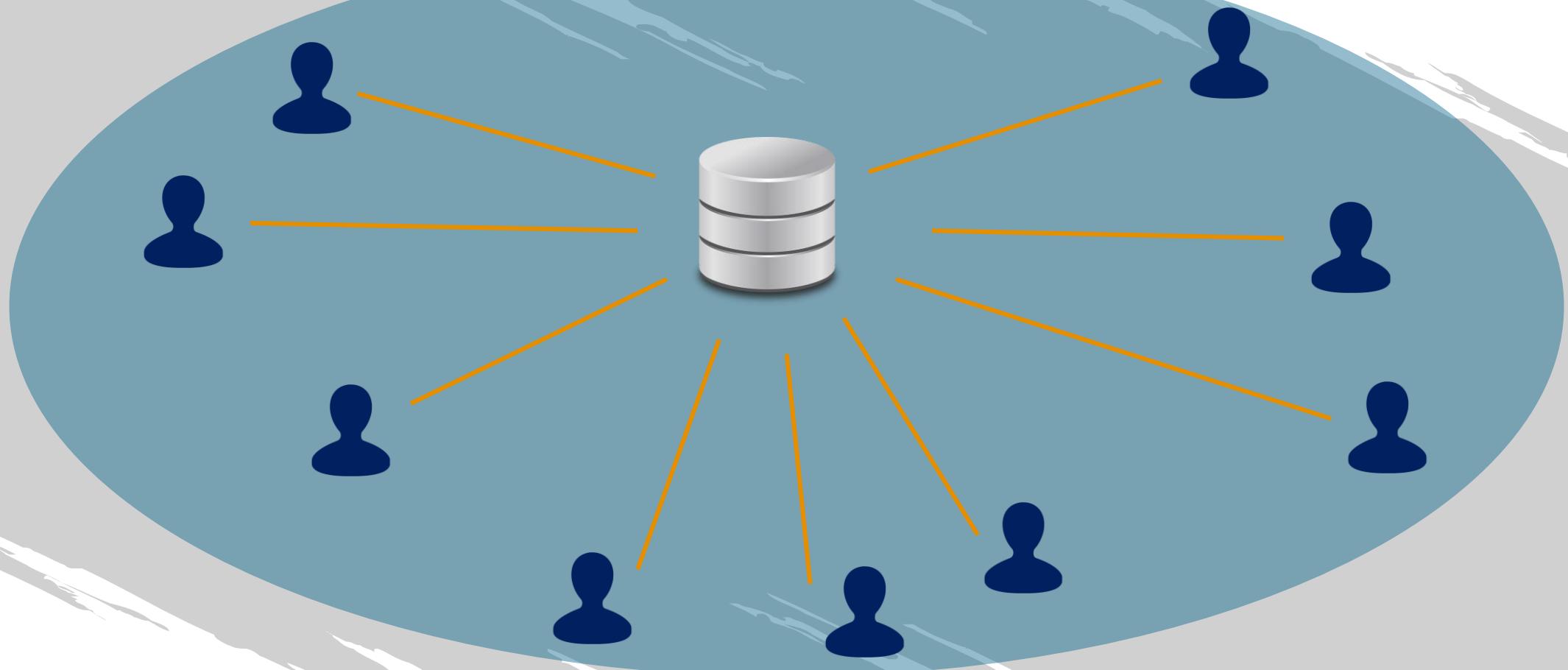
# Global Variables

# Global Variables

**global variables**

**global variables** apply to all connections related to a specific server

# Global Variables



# Global Variables



# Global Variables



```
SET GLOBAL var_name = value;
```

# Global Variables



SQL

```
SET GLOBAL var_name = value;
```



SQL

```
SET @@global.var_name = value;
```

# Global Variables

- you cannot set just any variable as global

# Global Variables

- you cannot set just any variable as global
  - a specific group of *pre-defined variables* in MySQL is suitable for this job. They are called system variables

# Global Variables

- you cannot set just any variable as global
    - a specific group of *pre-defined variables* in MySQL is suitable for this job. They are called system variables
-

# Global Variables

- you cannot set just any variable as global
    - a specific group of *pre-defined variables* in MySQL is suitable for this job. They are called system variables
- 

```
.max_connections()
```

# Global Variables

- you cannot set just any variable as global
    - a specific group of *pre-defined variables* in MySQL is suitable for this job. They are called system variables
- 

`.max_connections()`

`.max_join_size()`

# Global Variables

- you cannot set just any variable as global
    - a specific group of *pre-defined variables* in MySQL is suitable for this job. They are called system variables
- 

`.max_connections()`

- indicates the *maximum number of connections* to a server that can be established at a certain point in time

`.max_join_size()`

# Global Variables

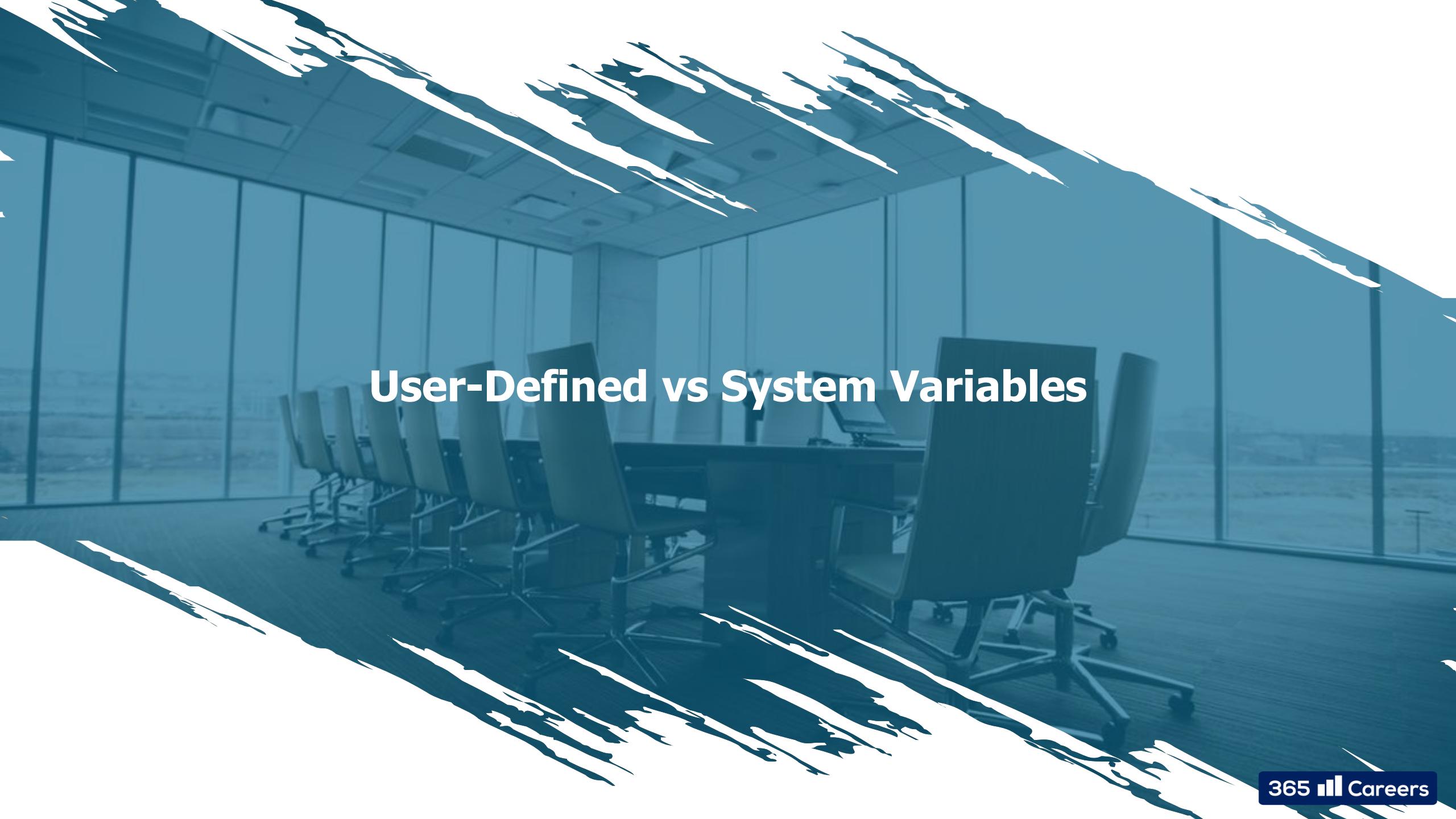
- you cannot set just any variable as global
    - a specific group of *pre-defined variables* in MySQL is suitable for this job. They are called system variables
- 

`.max_connections()`

- indicates the *maximum number of connections* to a server that can be established at a certain point in time

`.max_join_size()`

- sets the *maximum memory space* allocated for the joins created by a certain connection

A large conference room with rows of chairs facing a front wall with a large screen. The room has a modern design with a grid ceiling and large windows on the right side.

# User-Defined vs System Variables

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

user-defined

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

user-defined

system

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

**user-defined**

variables that can be *set by the user manually*

**system**

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

**user-defined**

variables that can be *set by the user manually*

**system**

variables that are *pre-defined* on our *system* –  
the MySQL server

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓		
system			

- local variables can be user-defined only

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓		
system	✗		

- local variables can be user-defined only

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓		
system	✗		✓

- only system variables can be set as global

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓		✗
system	✗		✓

- only system variables can be set as global

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓		✗
system	✗		✓

- both user-defined and system variables can be set as session variables

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓	✓	✗
system	✗	✓	✓

- both user-defined and system variables can be set as session variables

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓	✓	✗
system	✗	✓	✓

- both user-defined and system variables can be set as session variables  
there are limitations to this rule!

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓	✓*	✗
system	✗	✓*	✓

- both user-defined and system variables can be set as session variables  
there are limitations to this rule!

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓	✓*	✗
system	✗	✓*	✓

- some of the system variables can be defined as global only; they cannot be session variables

# User-Defined vs System Variables

- variables in MySQL can be characterized according to the way they have been created

	local	session	global
user-defined	✓	✓*	✗
system	✗	✓*	✓

- some of the system variables can be defined as global only; they cannot be session variables (e.g. `.max_connections()`)

A large, modern conference room with a long rectangular table in the center. Rows of black office chairs are arranged facing the table. The room has floor-to-ceiling windows on one side, providing a view of an outdoor area. The ceiling is white with a grid of recessed lights. The overall atmosphere is professional and spacious.

# MySQL Indexes

# MySQL Indexes



# MySQL Indexes



# MySQL Indexes

- the index of a table functions like the index of a book

# MySQL Indexes

- the index of a table functions like the index of a book
  - data is taken from a column of the table and is stored in a certain order in a distinct place, called an index

# MySQL Indexes

- the index of a table functions like the index of a book
  - data is taken from a column of the table and is stored in a certain order in a distinct place, called an index
- your datasets will typically contain 100,000+ or even 1,000,000+ records

# MySQL Indexes

- the index of a table functions like the index of a book
  - data is taken from a column of the table and is stored in a certain order in a distinct place, called an index
- your datasets will typically contain 100,000+ or even 1,000,000+ records
  - the *Larger* a database is, the *slower* the process of finding the record or records you need

# MySQL Indexes

- we can use an index that will increase the speed of searches related to a table

# MySQL Indexes

- we can use an index that will increase the speed of searches related to a table



SQL

```
CREATE INDEX index_name  
ON table_name (column_1, column_2, ...);
```

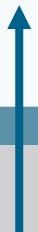
# MySQL Indexes

- we can use an index that will increase the speed of searches related to a table



SQL

```
CREATE INDEX index_name  
ON table_name (column_1, column_2, ...);
```



the parentheses serve us to indicate the column names on which our search will be based

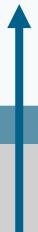
# MySQL Indexes

- we can use an index that will increase the speed of searches related to a table



SQL

```
CREATE INDEX index_name  
ON table_name (column_1, column_2, ...);
```



these must be fields from your data table you  
will search frequently

# MySQL Indexes

- composite indexes

# MySQL Indexes

- composite indexes

- applied to *multiple columns*, not just a single one

# MySQL Indexes

- composite indexes

- applied to *multiple columns*, not just a single one



SQL

```
CREATE INDEX index_name  
ON table_name (column_1, column_2, ...);
```

# MySQL Indexes

- composite indexes

- applied to *multiple columns*, not just a single one



SQL

```
CREATE INDEX index_name  
ON table_name (column_1, column_2, ...);
```

- carefully pick the *columns* that would optimize your search!

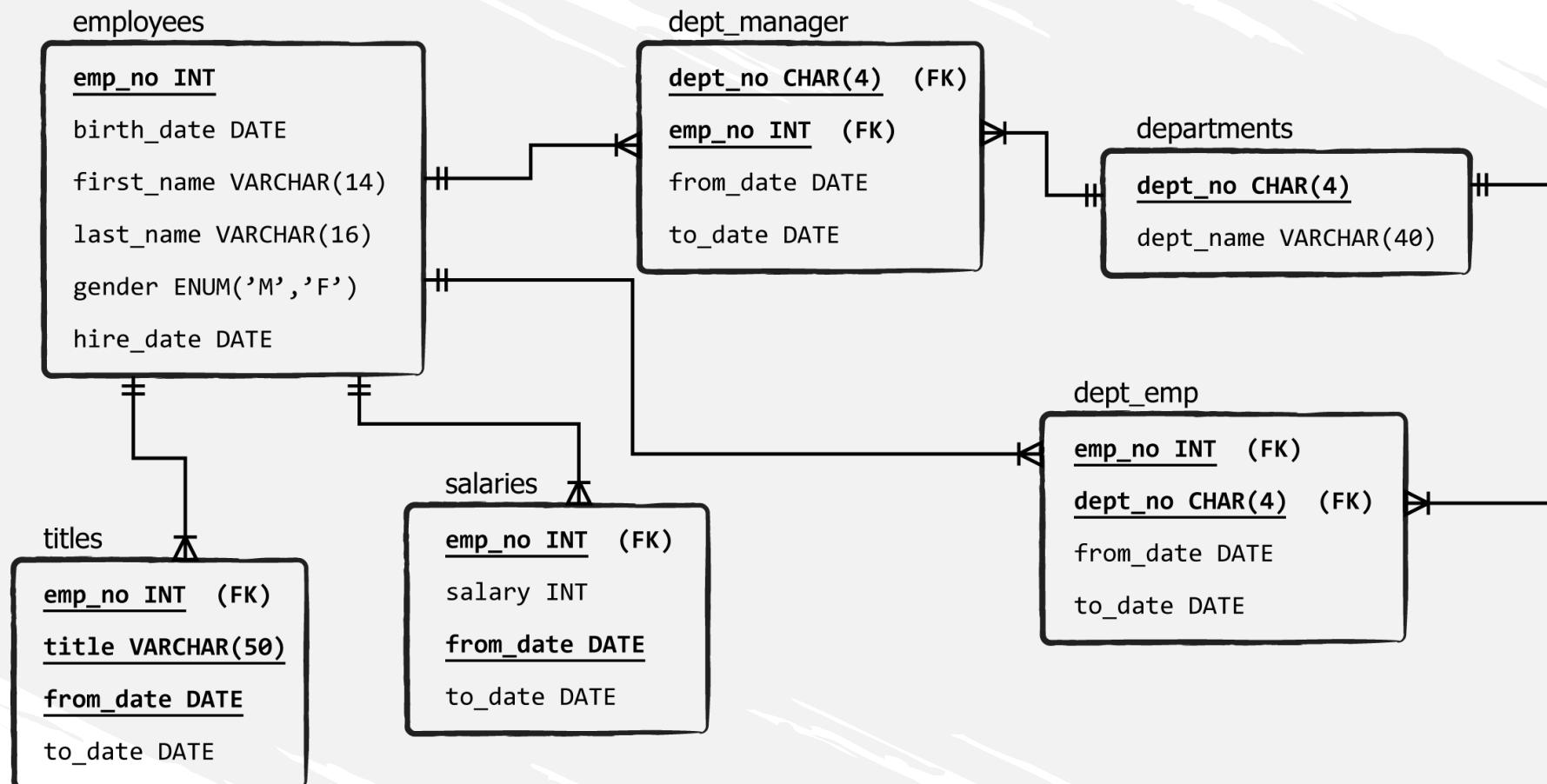
# MySQL Indexes

- *primary and unique keys* are MySQL indexes

# MySQL Indexes

- *primary* and *unique keys* are MySQL indexes
  - they represent columns on which a person would typically base their search

# MySQL Indexes



# MySQL Indexes

- SQL specialists are always aiming for a good balance between the *improvement of speed search and the resources used for its execution*

# MySQL Indexes

- SQL specialists are always aiming for a good balance between the *improvement of speed search and the resources used for its execution*

small datasets

the costs of having an index might be higher than the benefits

# MySQL Indexes

- SQL specialists are always aiming for a good balance between the improvement of speed search and the resources used for its execution

small datasets

the costs of having an index might be higher than the benefits

large datasets

a well-optimized index can make a positive impact on the search process