DATABASE RELATIONS

LESSON

INTRODUCTION

- When creating a database, common sense dictates that we use separate tables for different types of entities. Some examples are: users, tasks, etc… But we also need to have relationships between these tables. For instance, users assign tasks. These relationships need to be represented in the database. Also, when fetching data with SQL, we need to use certain types of JOIN queries to get what we need.

TYPES OF DATABASE RELATIONSHIPS

- One to One Relationships
- One to Many and Many to One Relationships
- Many to Many Relationships
- Self Referencing Relationships

ONE TO ONE RELATIONSHIPS

Let's say you have a table for customers:

customer_id	customer_name	customer_address
101	John Doe	Lacplesa 37
102	Bruce Wayne	Skolas 21a

We can put the customer address information on a separate table:

CUSTOMERS

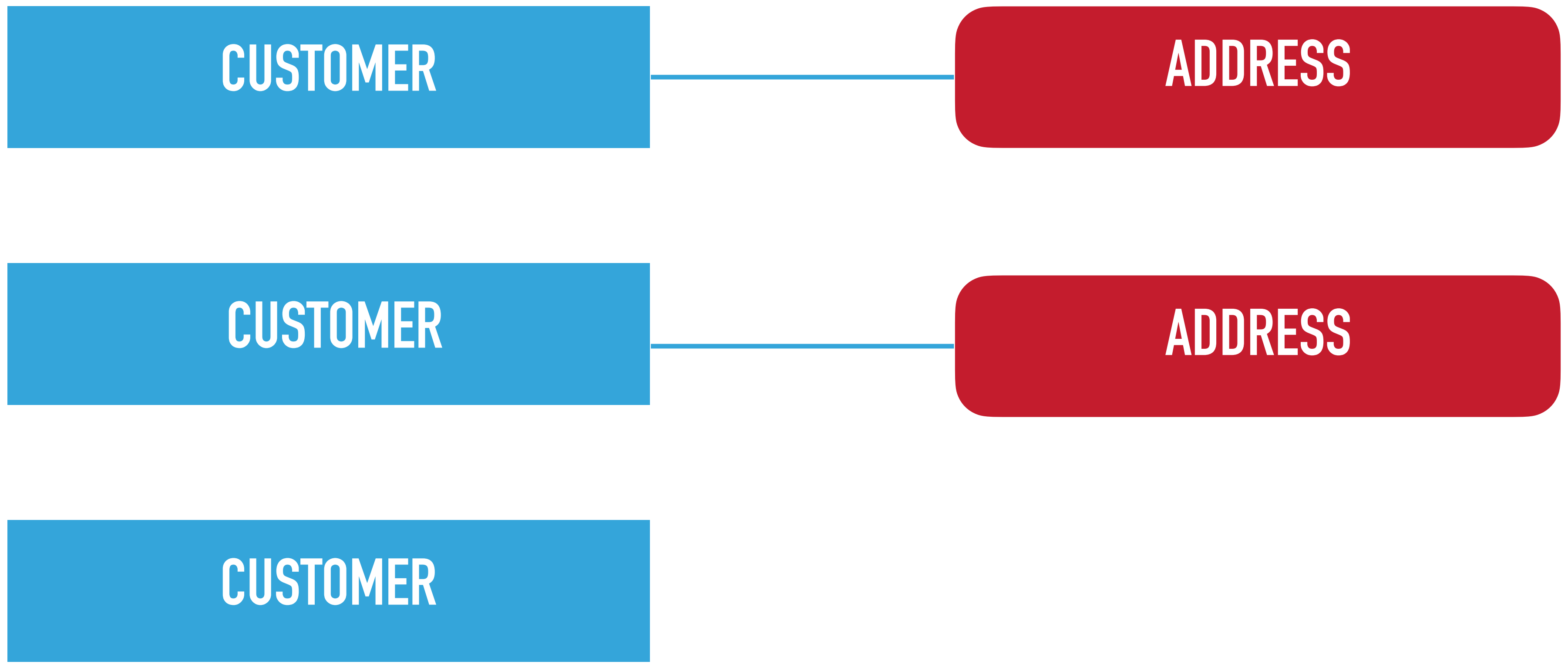
customer_id	customer_name	address_id
101	John Doe	301
102	Bruce Wayne	302

ADDRESSES

address_id	address
301	Lacplesa 37
302	Skolas 21

Now we have a relationship between the Customers table and the Addresses table. If each address can belong to only one customer, this relationship is "One to One". Keep in mind that this kind of relationship is not very common. Our initial table that included the address along with the customer could have worked fine in most cases.

Notice that now there is a field named "address_id" in the Customers table, that refers to the matching record in the Address table. This is called a "Foreign Key" and it is used for all kinds of database relationships.


CUSTOMER
ADDRESS
CUSTOMER
ADDRESS
CUSTOMER

ONE TO MANY AND MANY TO ONE RELATIONSHIPS

- This is the most commonly used type of relationship.
    - Customers can make many orders.
    - Orders can contain many items.
    - Items can have descriptions in many languages.
- In these cases we would need to create "One to Many" relationships.

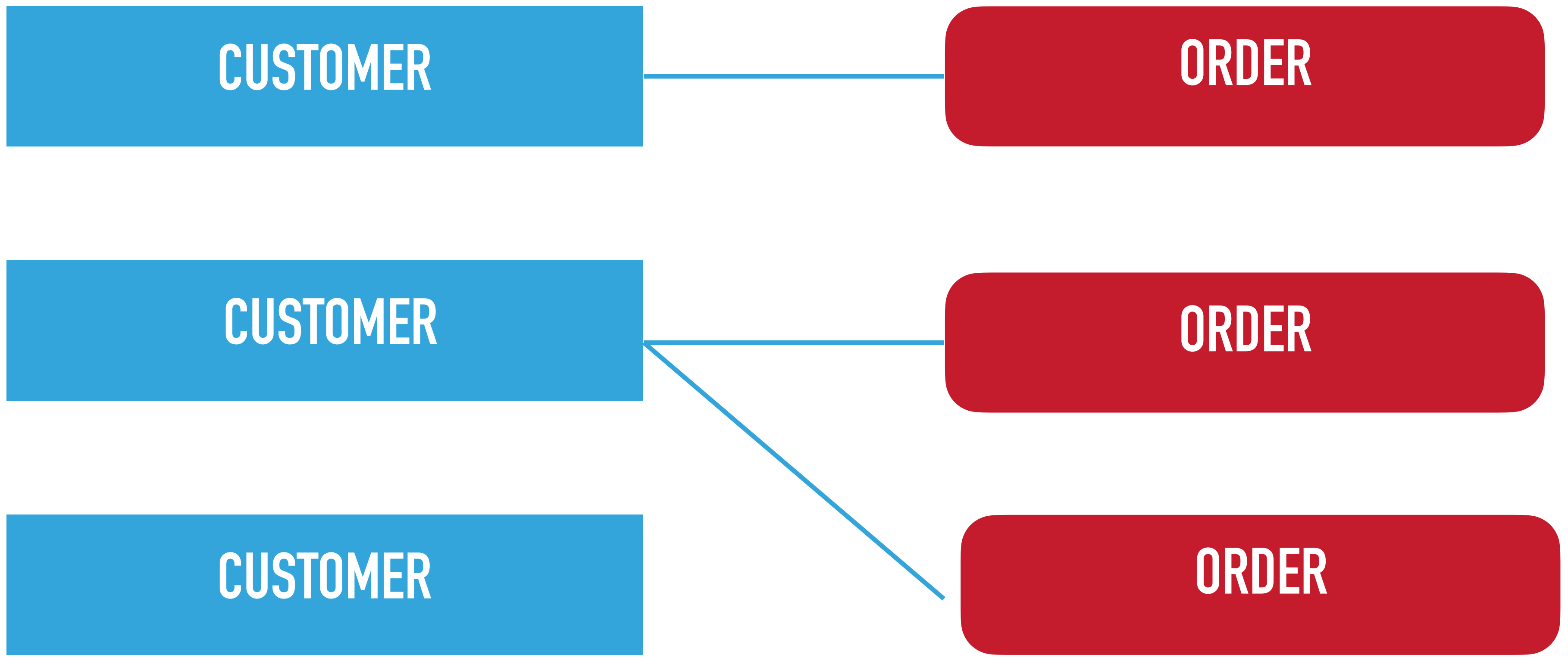
CUSTOMERS

customer_id	customer_name
101	John Doe
102	Bruce Wayne

ORDERS

order_id	customer_id	order_date	amount
201	101	11/03/2019	123$
202	102	12/03/2019	517$
203	101	12/03/2019	3154$

Each customer may have zero, one or multiple orders. But an order can belong to only one customer.



MANY TO MANY RELATIONSHIPS

- In some cases, you may need multiple instances on both sides of the relationship. For example, each order can contain multiple items. And each item can also be in multiple orders.
- For these relationships, we need to create an extra table.

ORDERS

id	customer_id	order_date	amount
201	101	11/03/2019	60$
202	102	12/03/2019	10$

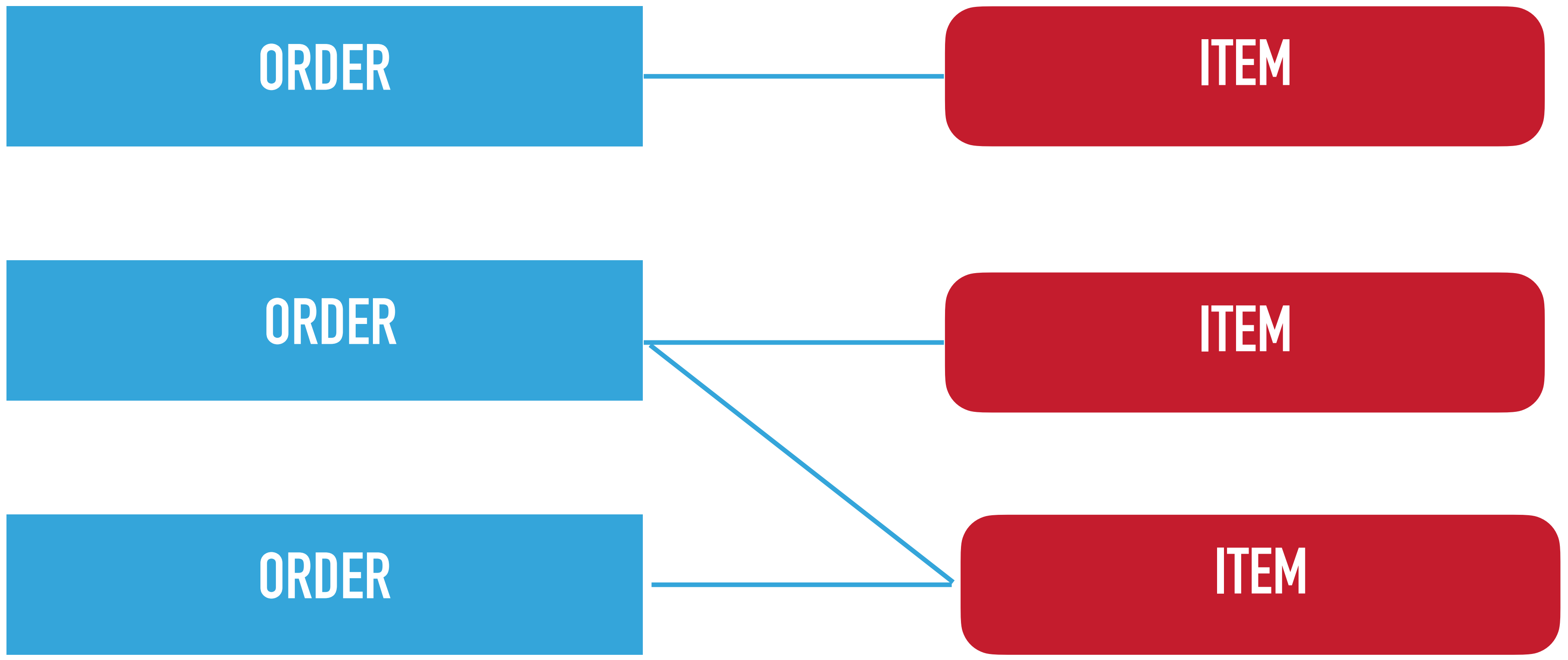
ITEMS

id	item_name	item_description
101	Rib-eye steak	juicy steak
102	Milka	milk chocolate
103	Evian	Mineral water

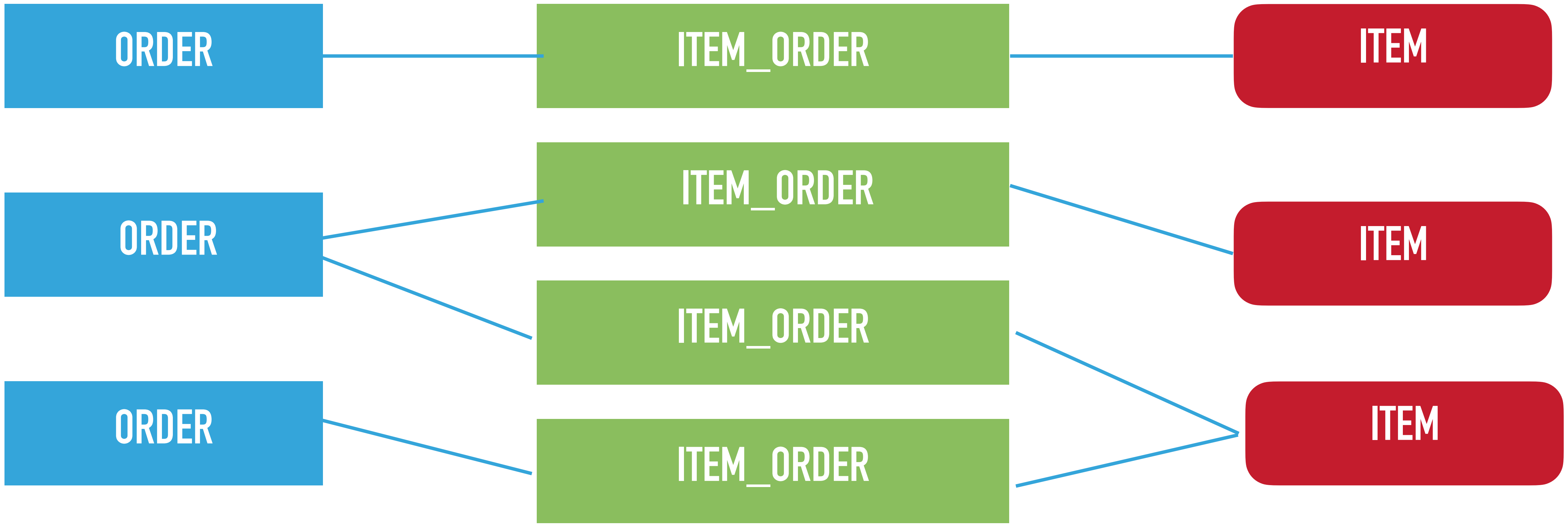
ITEM ORDERS

id	order_id	item_id
301	201	101
302	201	102
303	202	101

The Items_Orders table has only one purpose, and that is to create a "Many to Many" relationship between the items and the orders.



If you want to include the items_orders records in the graph, it may look like this:



PRIMARY AND FOREIGN KEYS

- A primary key uniquely identifies each record in the table. It is a type of candidate key that is usually the first column in a table and can be automatically generated by the database to ensure that it is unique.
- A foreign key is another candidate key (not the primary key) used to link a record to data in another table.

FOREIGN KEYS

- In the relationship examples above, we always had these "****_id" fields that referenced a column in another table. In this example, the customer_id column in the Orders table is a Foreign Key column.

CUSTOMERS

customer_id	customer_name
101	John Doe
102	Bruce Wayne

ORDERS

order_id	customer_id	order_date	amount
201	101	11/03/2019	123$
202	102	12/03/2019	517$
203	101	12/03/2019	3154$

DEFINING FOREIGN KEY

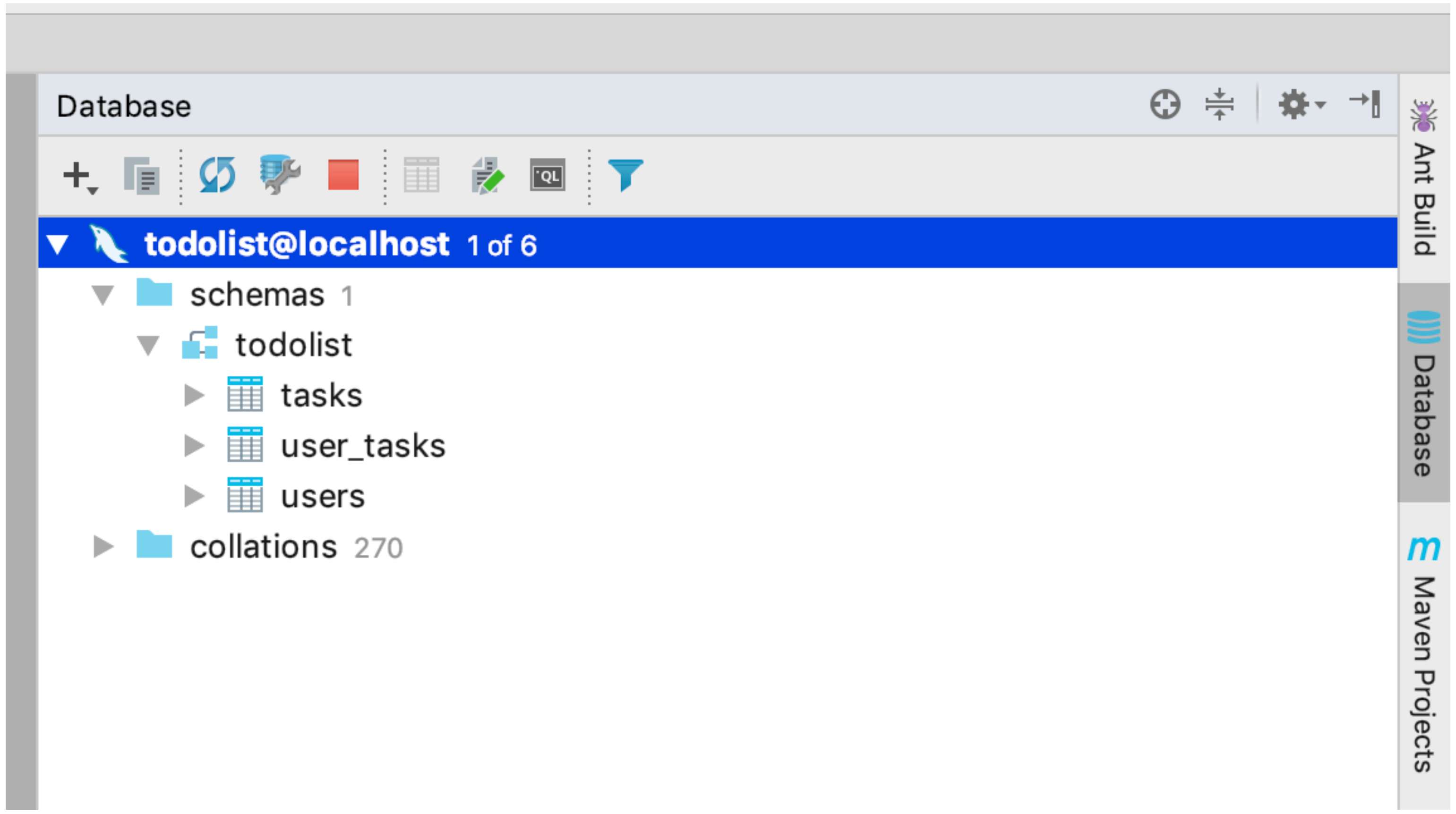
```sql
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_name VARCHAR(100)
);
```

```sql
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    amount DOUBLE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

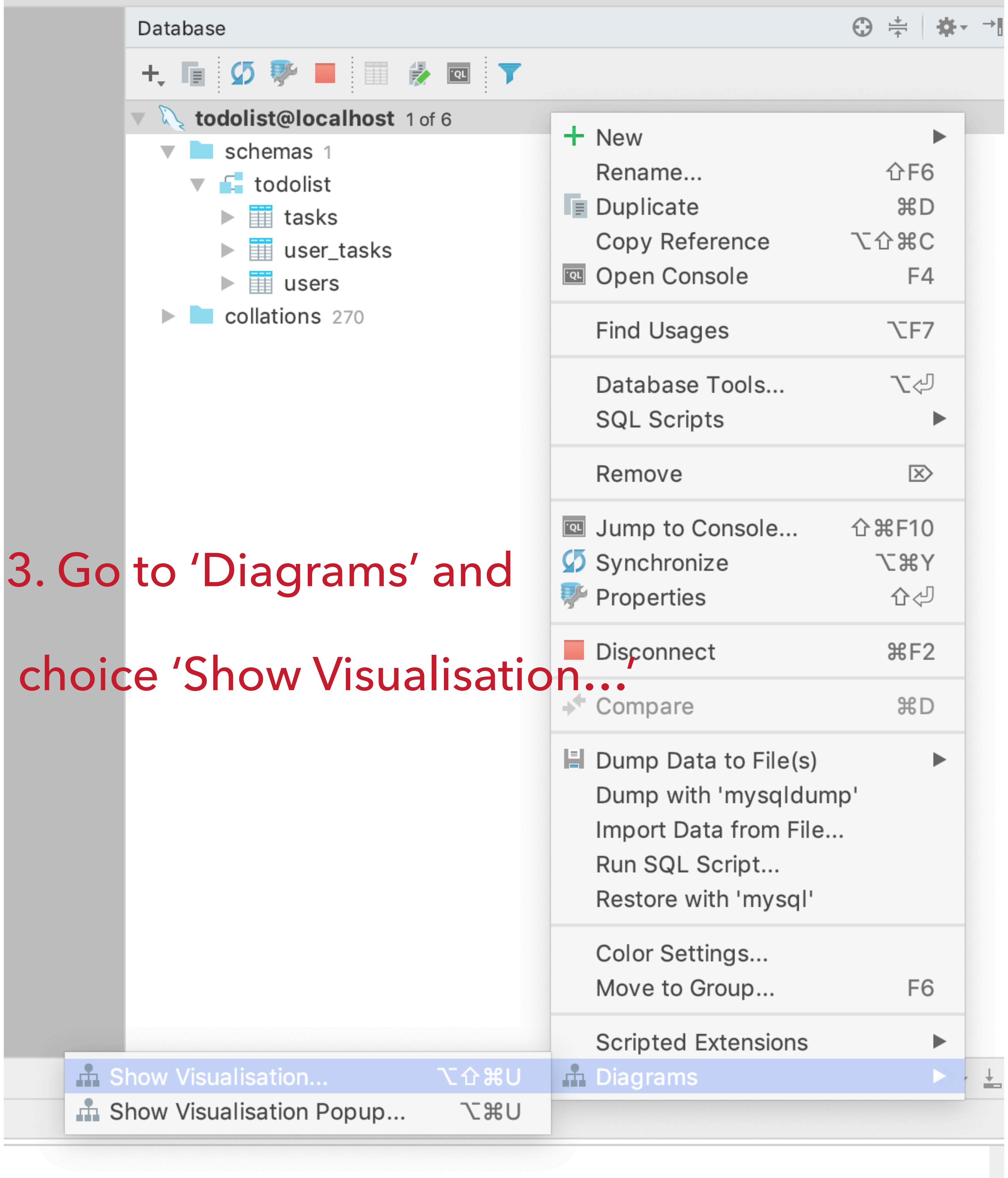
- Both columns (customers.customer_id and orders.customer_id) should be the same exact data structure. If one is INT, the other one should not be BIGINT for example.
- Please note that in MySQL only the InnoDB engine has full support for Foreign Keys. But other storage engines will still allow you to specify them without giving any errors. Also the Foreign Key column is indexed automatically, unless you specify another index for it.

VISUALISATION

1. Go to 'Database' in IntelliJ IDEA



2. Use right click on todolist@localhost



Database

todolist@localhost 1 of 6
- schemas 1
  - todolist
    - tasks
    - user_tasks
    - users
- collations 270

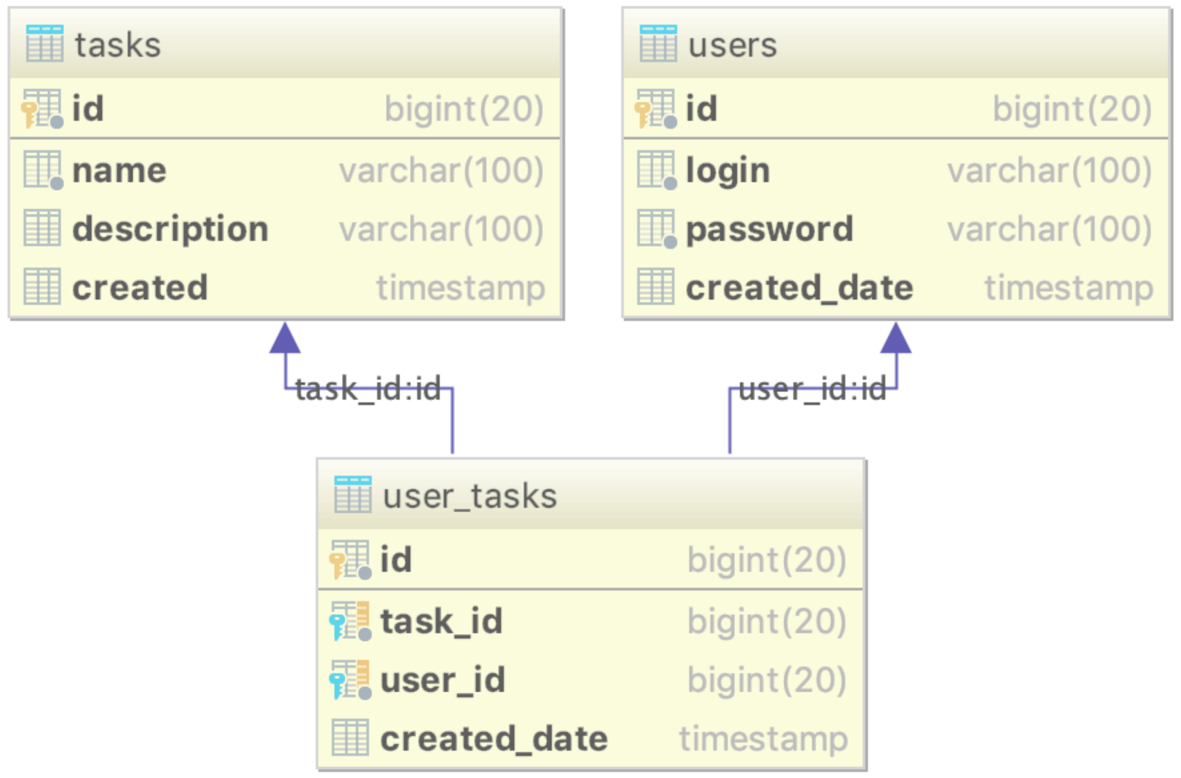
New	▶
Rename...	⇧F6
Duplicate	⌘D
Copy Reference	⌥⇧⌘C
Open Console	F4
Find Usages	⌥F7
Database Tools...	⌥⏎
SQL Scripts	▶
Remove	⌦
Jump to Console...	⇧⌘F10
Synchronize	⌥⌘Y
Properties	⇧⏎
Disconnect	⌘F2
Compare	⌘D
Dump Data to File(s)	▶
Dump with 'mysqldump'	
Import Data from File...	
Run SQL Script...	
Restore with 'mysql'	
Color Settings...	
Move to Group...	F6
Scripted Extensions	▶
Diagrams	▶

Show Visualisation...	⌥⇧⌘U
Show Visualisation Popup...	⌥⌘U

3. Go to 'Diagrams' and

choice 'Show Visualisation...'

Result



tasks
id bigint(20)
name varchar(100)
description varchar(100)
created timestamp
users
id bigint(20)
login varchar(100)
password varchar(100)
created_date timestamp
task_id:id
user_id:id
user_tasks
id bigint(20)
task_id bigint(20)
user_id bigint(20)
created_date timestamp

HIBERNATE ANNOTATIONS

@Column Annotation part 2

```
@Column(name = "login", nullable = false, unique = true)
```

```
/**
 * (Optional) Whether the column is a unique key.  This is a
 * shortcut for the UniqueConstraint annotation at the table
 * level and is useful for when the unique key constraint
 * corresponds to only a single column. This constraint applies
 * in addition to any constraint entailed by primary key mapping and
 * to constraints specified at the table level.
 */
boolean unique() default false;
```

```
/**
 * (Optional) Whether the database column is nullable.
 */
boolean nullable() default true;
```

@ManyToOne Annotation

```
* Defines a single-valued association to another entity class that
* has many-to-one multiplicity. It is not normally necessary to
* specify the target entity explicitly since it can usually be
* inferred from the type of the object being referenced.  If the
* relationship is bidirectional, the non-owning
* OneToMany entity side must used the
* mappedBy element to specify the relationship field or
* property of the entity that is the owner of the relationship.
```

@JoinColumn Annotation

```
@JoinColumn(name = "task_id", nullable = false)
```

```
/**
 * (Optional) The name of the foreign key column.
 * The table in which it is found depends upon the
 * context.
 * If the join is for a OneToOne or ManyToOne
 *  mapping using a foreign key mapping strategy,
 * the foreign key column is in the table of the
 * source entity or embeddable.
 * If the join is for a unidirectional OneToMany mapping
 * using a foreign key mapping strategy, the foreign key is in the
 * table of the target entity.
 * If the join is for a ManyToMany mapping or for a OneToOne
 * or bidirectional ManyToOne/OneToMany mapping using a join
 * table, the foreign key is in a join table.
 * If the join is for an element collection, the foreign
 * key is in a collection table.
 *
```

REFERENCES

- https://database.guide/the-3-types-of-relationships-in-database-design/
- https://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561
- https://www.lifewire.com/database-relationships-p2-1019758
- https://www.techrepublic.com/article/relational-databases-defining-relationships-between-database-tables/