INNER JOIN

Let’s discuss these two queries:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT \*  FROM country, city  WHERE city.country\_id = country.id;    SELECT \*  FROM country  INNER JOIN city ON city.country\_id = country.id; |

The result they return is presented on the picture below:



Both queries return exactly the same result. This is not by accident but the result of the fact that this is the same query written in two different ways. Both ways are correct, and you can use any of them.

In the first query, we listed all tables we use in the FROM part of the query (FROM country, city) and then went with the join condition in the WHERE part of the query (WHERE city.country\_id = country.id). In case we forgot to write down this join condition, we would have the Cartesian product of both tables.

In the second query, we have only one table in the FROM part of the query (FROM country) and then we have the second table and the JOIN condition in the JOIN part of the query (INNER JOIN city ON city.country\_id = country.id).

While both queries are well-written, I would suggest that you always use INNER JOIN instead of listing tables and joining them in the WHERE part of the query. There are a few reasons for that:

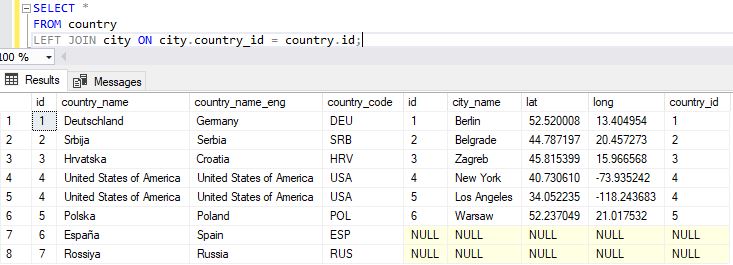
* Readability is much better because the table used and related JOIN condition are in the same line. You can easily see if you omitted the JOIN condition or not
* If you want to use other JOINs later (LEFT or RIGHT), you couldn’t do that (easily) unless you’ve used INNER JOIN before that

LEFT JOIN

I’ll repeat this – “We don’t have 2 countries on the list (Spain and Russia) because they don’t have any related city in the **city** table“. This shall prove crucial when comparing INNER JOIN vs LEFT JOIN.

In some cases, we want to have even these records in our results. For example, you simply want to see in the result that these countries don’t have related records in another table. This could be part of some control, or maybe just counting cases, etc. No matter what the motivation behind that desire is, we should be technically able to do that. And we are. In databases, LEFT JOIN does exactly that.

The result of LEFT JOIN shall be the same as the result of INNER JOIN + we’ll have rows, from the “left” table, without a pair in the “right” table. We’ll use the same INNER JOIN query and just replace the word INNER with LEFT. This is the result:

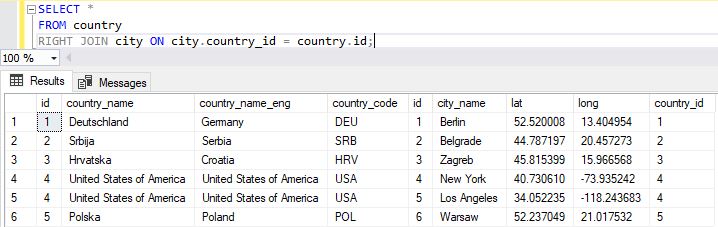


You can easily notice, that we have 2 more rows, compared to the result of the INNER JOIN query. These are rows for Russia and Spain. Since they both don’t have any related city, all city attributes in these two rows have NULL values (are not defined). That is the biggest difference when comparing INNER JOIN vs LEFT JOIN.

RIGHT JOIN

You’ll at least hear about the RIGHT JOIN. It’s rarely used because it returns the same result as the LEFT JOIN. On the other hand, queries which use LEFT JOIN are much easier to read because we simply list tables one after the other.

This is the equivalent of the previous query using the RIGHT JOIN:



You can notice that returned values are the same, only in this case values from the city table are in the first 5 columns, and country-related values come after them.

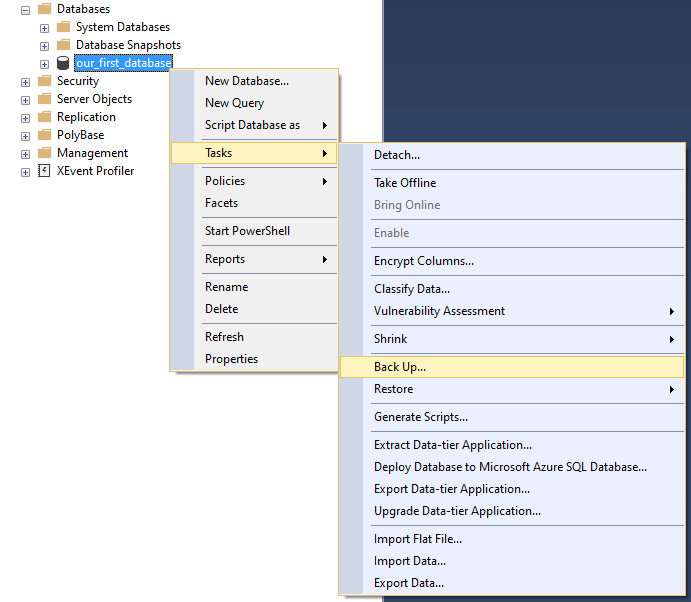
Backup and restore

Using scripts is usually related to making significant changes in the database. I might be paranoid about this, but I prefer to backup the database before these changes.

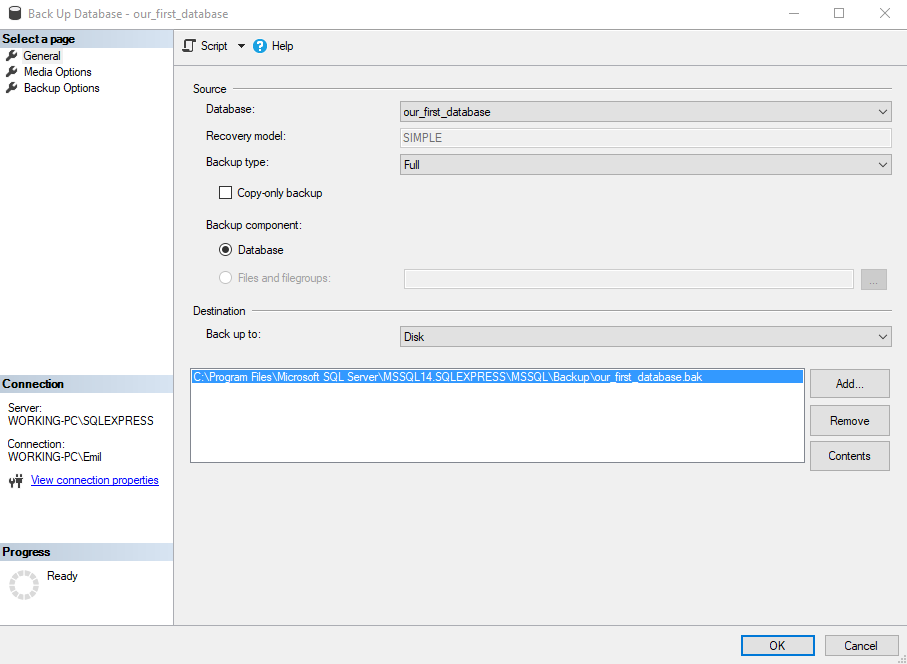
* ***Tip:****If you expect major changes in your databases, either in structure, either data changes, creating a backup is always a good idea. You could backup the entire database or only 1 table. That’s completely up to you and the changes you’re making.*

Oldie but goodie: *“There are two types of people: 1. People who do backup & 2. People who will start doing backup.”*

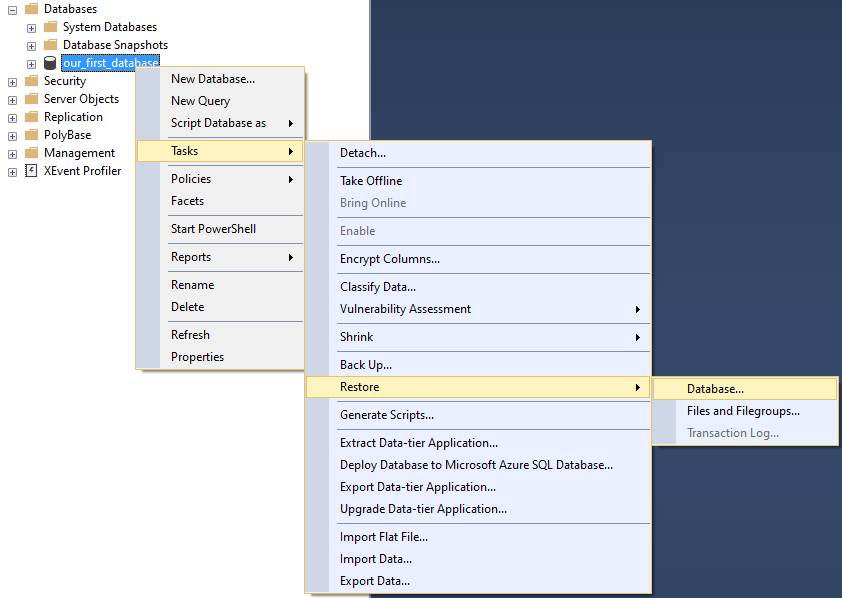
In SQL Server you can easily backup the whole database. First right-click on the database you want to backup, go to **Tasks**, and choose the **Back Up** option:



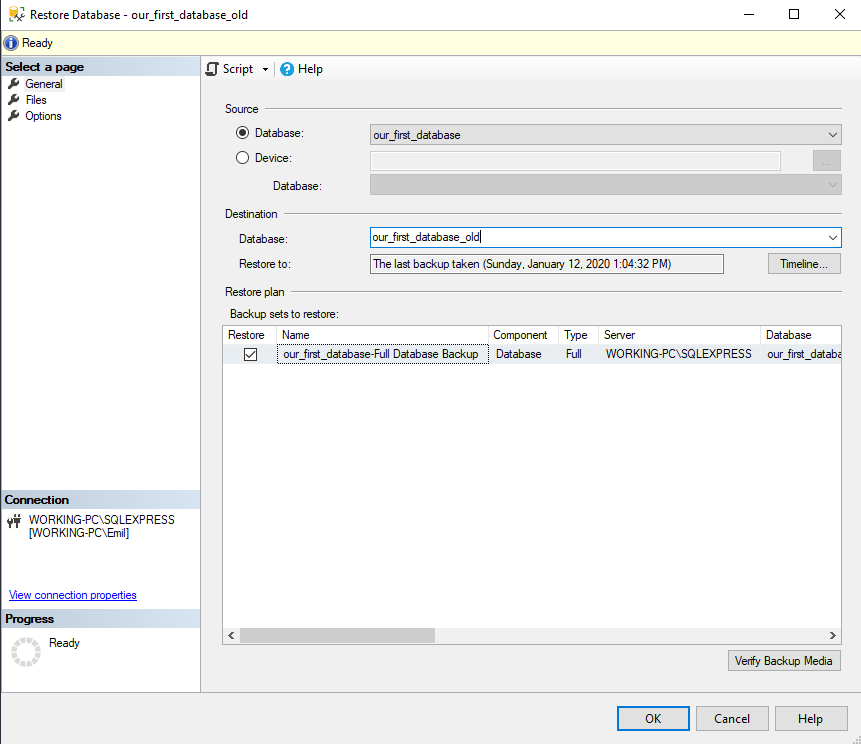
After choosing the backup option, you’ll get the pop-up as on the picture below:



Click **OK**, and your backup is created. Now we can create a copy of our database. We’ll do that by restoring from the backup. To do that, follow the process in the picture below:



You’ll need to enter the name of the restored database (**test\_database\_old**) and click **OK**:



The result of these actions shall be that now we have 2 identical databases – **test\_database** & **test\_database\_old**. We’ll use the first one as the one where we’ll run our SQL script. In case we would screw something up, we could always easily revert changes using our backup.

**SQL Script**

Now we’re ready to take a look at our script. We want to do two different things:

* Create new database objects (tables and relations) – DDL commands, and
* Populate these tables with data – DML commands

Of course, we’ll run DDL commands first and then run DML commands. Trying to insert data into a table that doesn’t exist would result in errors. So, let’s take a look at our script now:

-- tables

-- Table: call

CREATE TABLE call (

id int NOT NULL IDENTITY(1, 1),

employee\_id int NOT NULL,

customer\_id int NOT NULL,

start\_time datetime NOT NULL,

end\_time datetime NULL,

call\_outcome\_id int NULL,

CONSTRAINT call\_ak\_1 UNIQUE (employee\_id, start\_time),

CONSTRAINT call\_pk PRIMARY KEY (id)

);

-- Table: call\_outcome

CREATE TABLE call\_outcome (

id int NOT NULL IDENTITY(1, 1),

outcome\_text char(128) NOT NULL,

CONSTRAINT call\_outcome\_ak\_1 UNIQUE (outcome\_text),

CONSTRAINT call\_outcome\_pk PRIMARY KEY (id)

);

-- Table: customer

CREATE TABLE customer (

id int NOT NULL IDENTITY(1, 1),

customer\_name varchar(255) NOT NULL,

city\_id int NOT NULL,

customer\_address varchar(255) NOT NULL,

next\_call\_date date NULL,

ts\_inserted datetime NOT NULL,

CONSTRAINT customer\_pk PRIMARY KEY (id)

);

-- Table: employee

CREATE TABLE employee (

id int NOT NULL IDENTITY(1, 1),

first\_name varchar(255) NOT NULL,

last\_name varchar(255) NOT NULL,

CONSTRAINT employee\_pk PRIMARY KEY (id)

);

-- foreign keys

-- Reference: call\_call\_outcome (table: call)

ALTER TABLE call ADD CONSTRAINT call\_call\_outcome

FOREIGN KEY (call\_outcome\_id)

REFERENCES call\_outcome (id);

-- Reference: call\_customer (table: call)

ALTER TABLE call ADD CONSTRAINT call\_customer

FOREIGN KEY (customer\_id)

REFERENCES customer (id);

-- Reference: call\_employee (table: call)

ALTER TABLE call ADD CONSTRAINT call\_employee

FOREIGN KEY (employee\_id)

REFERENCES employee (id);

-- Reference: customer\_city (table: customer)

ALTER TABLE customer ADD CONSTRAINT customer\_city

FOREIGN KEY (city\_id)

REFERENCES city (id);

-- insert values

INSERT INTO call\_outcome (outcome\_text) VALUES ('call started');

INSERT INTO call\_outcome (outcome\_text) VALUES ('finished - successfully');

INSERT INTO call\_outcome (outcome\_text) VALUES ('finished - unsuccessfully');

INSERT INTO employee (first\_name, last\_name) VALUES ('Thomas (Neo)', 'Anderson');

INSERT INTO employee (first\_name, last\_name) VALUES ('Agent', 'Smith');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Jewelry Store', 4, 'Long Street 120', '2020/1/21', '2020/1/9 14:1:20');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Bakery', 1, 'Kurfürstendamm 25', '2020/2/21', '2020/1/9 17:52:15');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Café', 1, 'Tauentzienstraße 44', '2020/1/21', '2020/1/10 8:2:49');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Restaurant', 3, 'Ulica lipa 15', '2020/1/21', '2020/1/10 9:20:21');

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 4, '2020/1/11 9:0:15', '2020/1/11 9:12:22', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 2, '2020/1/11 9:14:50', '2020/1/11 9:20:1', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 3, '2020/1/11 9:2:20', '2020/1/11 9:18:5', 3);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 1, '2020/1/11 9:24:15', '2020/1/11 9:25:5', 3);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 3, '2020/1/11 9:26:23', '2020/1/11 9:33:45', 2);

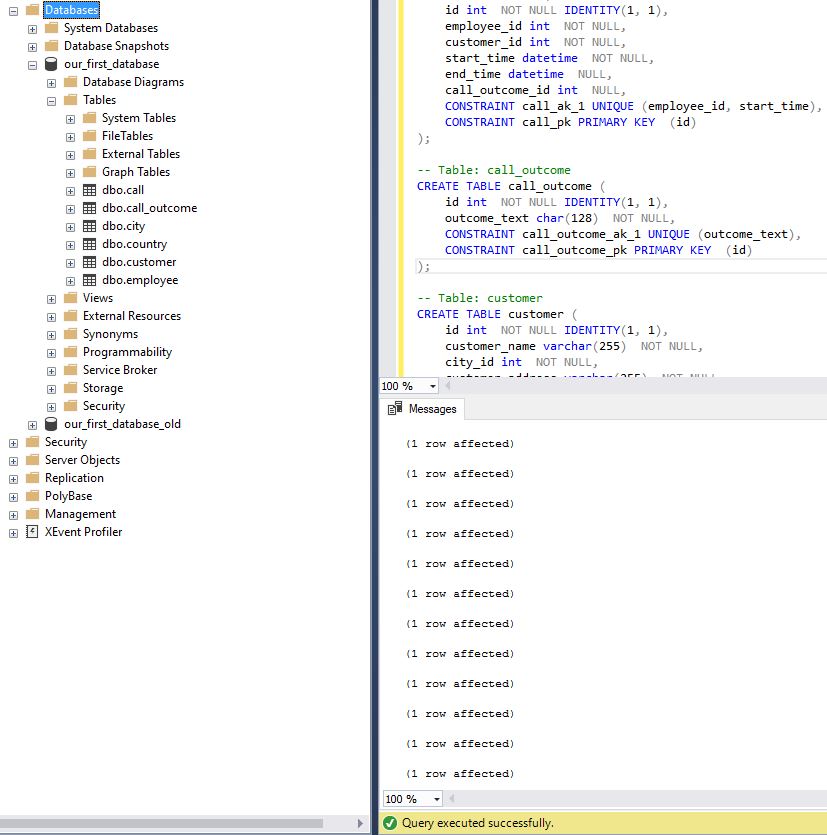
INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 2, '2020/1/11 9:40:31', '2020/1/11 9:42:32', 2);

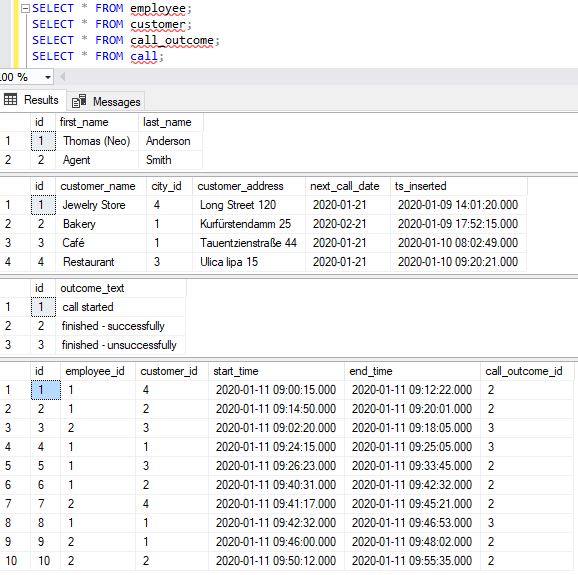
INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 4, '2020/1/11 9:41:17', '2020/1/11 9:45:21', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 1, '2020/1/11 9:42:32', '2020/1/11 9:46:53', 3);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 1, '2020/1/11 9:46:0', '2020/1/11 9:48:2', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 2, '2020/1/11 9:50:12', '2020/1/11 9:55:35', 2);

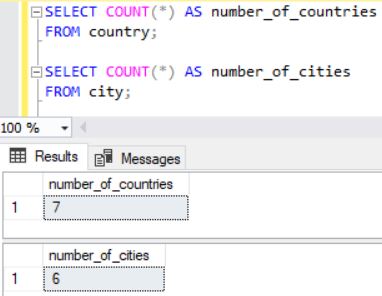




Aggregate Function

|  |  |
| --- | --- |
|  |  |

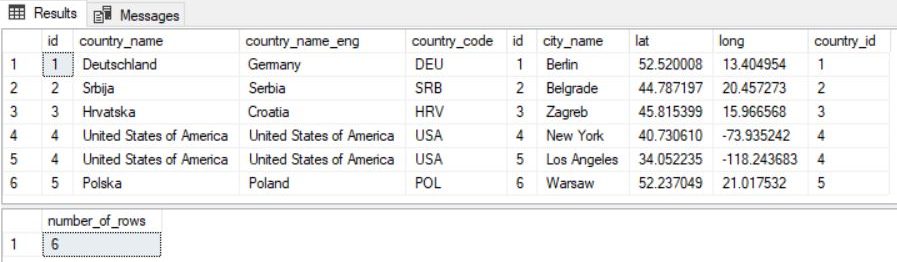




Aggregate Functions & JOINs

Now let’s try two more things. First, we’ll test how **COUNT** works when we’re joining tables. To do that, we’ll use the following queries:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT \*  FROM country  INNER JOIN city ON city.country\_id =  country.id;    SELECT COUNT(\*) AS number\_of\_rows  FROM country  INNER JOIN city ON city.country\_id =  country.id; |



SELECT \*

FROM country

LEFT JOIN city ON city.country\_id = country.id;

SELECT COUNT(\*) AS number\_of\_rows

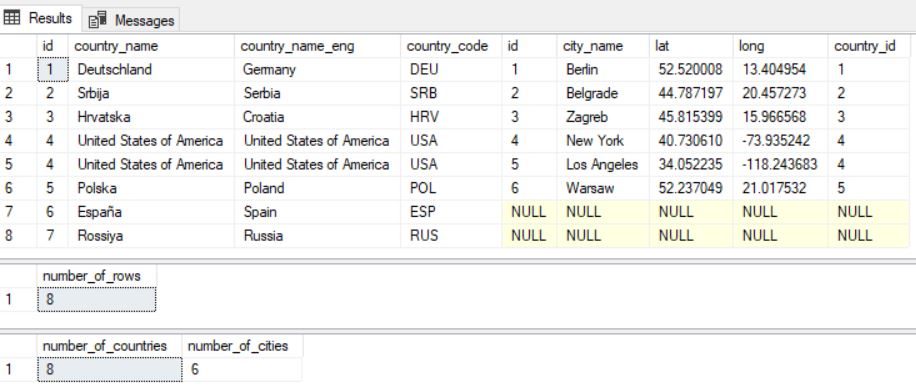
FROM country

LEFT JOIN city ON city.country\_id = country.id;

SELECT COUNT(country.country\_name) AS countries, COUNT(city.city\_name) AS cities

FROM country

LEFT JOIN city ON city.country\_id = country.id;



We can notice a few things:

* 1st query returned 8 rows. These are the same 6 rows as in a query using **INNER JOIN** and 2 more rows for countries that don’t have any related city (Russia & Spain)
* 2nd query counts the number of rows 1st query returns, so this number is 8
* 3rd query has two important things to comment on. The first one is that we’ve used aggregate function (**COUNT**), twice in the **SELECT** part of the query. This will usually be the case because you’re interested in more details about the group you want to analyze (number of records, average values, etc.). The second important thing is that these 2 counts used column names instead of “\*” and they returned different values. That happens because **COUNT** was created that way. If you put column names between brackets **COUNT** will count how many values are there (not including NULL values). All our records had value for country\_name, so the 1st COUNT returned 8. On the other hand, city\_name wasn’t defined 2 times (=NULL), so the 2nd COUNT returned 6 (8-2=6)
* ***Note:****This stands for other aggregate functions as well. If they run into NULL values, they will simply ignore them and calculate as they don’t exist.*

SQL Aggregate Functions

Now it’s time that we mention all T-SQL aggregate functions. The most commonly used are:

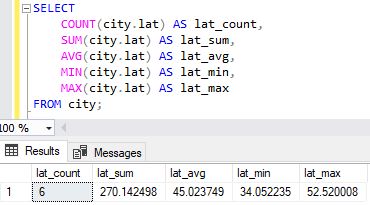
* **COUNT** – counts the number of elements in the group defined
* **SUM** – calculates the sum of the given attribute/expression in the group defined
* **AVG** – calculates the average value of the given attribute/expression in the group defined
* **MIN** – finds the minimum in the group defined
* **MAX** – finds the maximum in the group defined

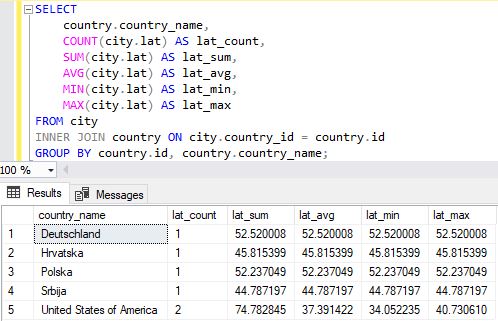
While all aggregate functions could be used without the **GROUP BY** clause, the whole point is to use the **GROUP BY** clause. That clause serves as the place where you’ll define the condition on how to create a group. When the group is created, you’ll calculate aggregated values.

* ***Example:****Imagine that you have a list of professional athletes and you know which sport each one of them plays. You could ask yourself something like – From my list, return the minimal, maximal and average height of players, grouped by the sport they play. The result would be, of course, MIN, MAX, and AVG height for groups – “football players”, “basketball players”, etc.*

Aggregate Functions – Examples

Now, let’s take a look at how these functions work on a single table. They are rarely used this way, but it’s good to see it, at least for educational purposes:





This is a much “smarter” query than the previous one. It returned the list of all countries, with a number of cities in them, as well as SUM, AVG, MIN, and MAX of their **lat** values.

Please notice that we’ve used the **GROUP BY** clause. By placing **country.id** and **country. country\_name**, we’ve defined a group. All cities belonging to the same country will be in the same group. After the group is created, aggregated values are calculated.

Complex Query

SELECT

country.country\_name\_eng,

SUM(CASE WHEN call.id IS NOT NULL THEN 1 ELSE 0 END) AS calls,

AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) AS avg\_difference

FROM country

LEFT JOIN city ON city.country\_id = country.id

LEFT JOIN customer ON city.id = customer.city\_id

LEFT JOIN call ON call.customer\_id = customer.id

GROUP BY

country.id,

country.country\_name\_eng

HAVING AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) > (SELECT AVG(DATEDIFF(SECOND, call.start\_time, call.end\_time)) FROM call)

ORDER BY calls DESC, country.id ASC;

SELECT

\*

FROM country

LEFT JOIN city ON city.country\_id = country.id

LEFT JOIN customer ON city.id = customer.city\_id

LEFT JOIN call ON call.customer\_id = customer.id;

Now we’ll write down the query which returns aggregated values for all countries.

SELECT

country.country\_name\_eng,

SUM(CASE WHEN call.id IS NOT NULL THEN 1 ELSE 0 END) AS calls,

AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) AS avg\_difference

FROM country

LEFT JOIN city ON city.country\_id = country.id

LEFT JOIN customer ON city.id = customer.city\_id

LEFT JOIN call ON call.customer\_id = customer.id

GROUP BY

country.id,

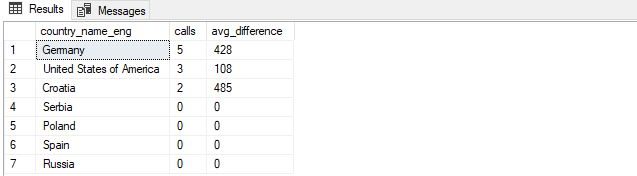
country.country\_name\_eng

ORDER BY calls DESC, country.id ASC;

I would like to point out two things here:

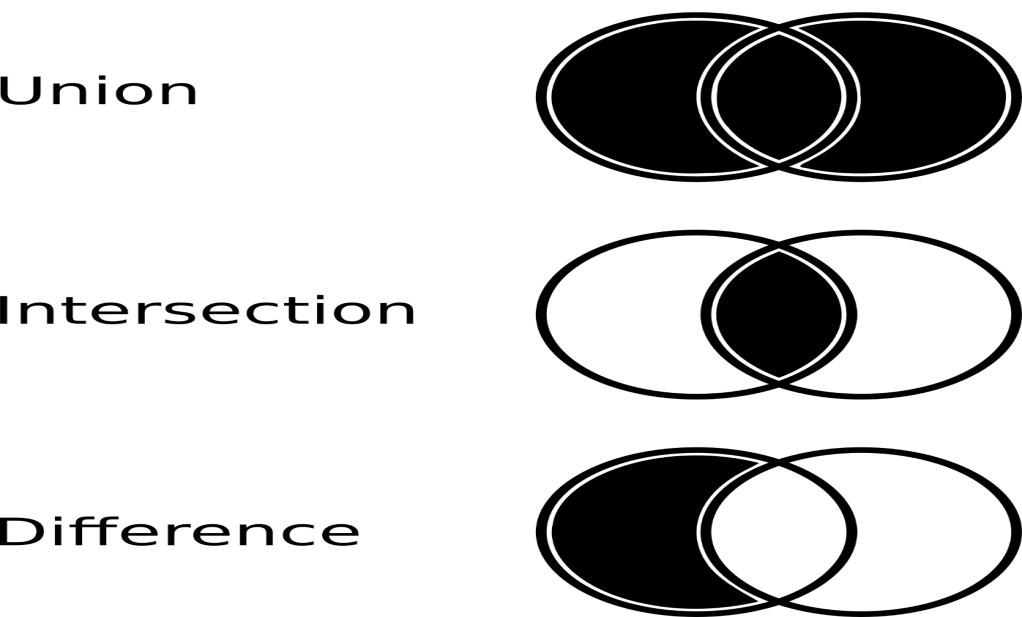
* SUM(CASE WHEN call.id IS NOT NULL THEN 1 ELSE 0 END) – This will sum up only existing calls. Since we’ve used LEFT JOIN, we’ll also join countries without any call. In case we’ve used COUNT, we would have value 1 returned for countries without any call, and we want 0 there (we want to see that info)
* AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) – This is very similar to the previously mentioned AVG. The difference here is that I’ve used ISNULL(…,0). This simply tests if the calculated value IS NULL, and if so, replaces it with 0. Calculated value could be NULL if there is not data (we’ve used LEFT JOIN)

Let’s see what this query returns.



## Set Theory and Venn Diagrams

In SQL Server we have 3 important operators at our disposal – UNION (ALL), INTERSECT, and EXCEPT. They return the result of related operators from the set theory (on the picture below).



The easiest way to explain this is:

* UNION – Returns elements from both sets (if there are duplicates, they are in the final set, only once)
* UNION ALL – Same as the UNION operator, but will contain all duplicates
* INTERSECT – Returns a set containing elements that are present in both sets
* EXCEPT/MINUS (difference) – A MINUS B is a set containing elements from the set A that are not elements of the set B (so A MINUS (A INTERSECT B))

We won’t analyze situations where sets don’t have any common elements (A UNION B = all elements from A and B, A INTERSECT B = {}, A EXCEPT B = A, B EXCEPT A = B) and where set A = set B (A UNION B = A = B, A INTRSECT B = A = B, A EXCEPT B = B EXCEPT A = {}).

## Set Theory and SQL

We talked a lot about the set theory so far, and now it’s time for some practice. We’ll write down a few queries which will show how UNION (ALL), INTERSECT and EXCEPT operators work.

**#1 First we’ll test two separate queries and analyze the result set they return**

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

);

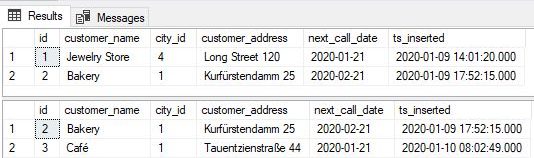
**-- list all customers from Berlin**

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';



You should notice a few things:

* The first query returns all customers having exactly 3 calls
* The second query returns all customers from Berlin
* Both queries return the same columns, but rows returned are not the same. This is important because you can use operators working with sets only if these two sets are composed of elements with the same structure
* Each result set has 2 rows. “Bakery” is present in both result sets, and each set has one other row

### #2 UNION and UNION ALL

Now we’ll use two available UNION operators. Any of these operators (UNION (ALL), INTERSECT, EXCEPT) is used in a way you just place it between queries

-- UNION

**-- list all customers with exactly 3 calls**

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

UNION

-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';

-- UNION ALL

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

UNION ALL

-- list all customers from Berlin

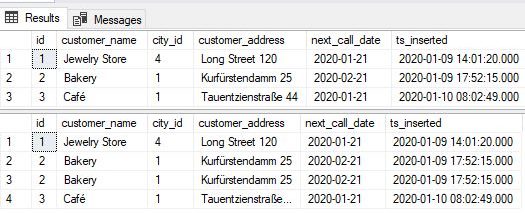
select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';.

|  |  |
| --- | --- |
|  |  |



For these two result sets returned, you should notice the following:

* Each query, the first one using UNION, and the second one using UNION ALL returns 1 result set
* The result set returned by the UNION query returned all rows returned by the two queries used. The only difference is that the duplicated row had been eliminated
* The query using UNION ALL returned all rows from both queries, without removing duplicates
* The UNION is used more often, and you’ll probably use it when you have a few complex queries and you simply want to “join” their results without writing a single more complex query

### #3 INTERSECT

The INTERSECT should return elements/rows which appear in both sets.

-- INTERSECT

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

INTERSECT

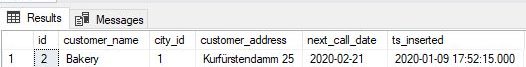
-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';



Everything went as expected and you can see that the “Bakery” row was returned as a result.

### #4 EXCEPT

The EXCEPT operator returns all elements/rows from the first set, except those that are in the second set.

-- A EXCEPT B

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

EXCEPT

-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';

-- B EXCEPT A

-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin'

EXCEPT

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

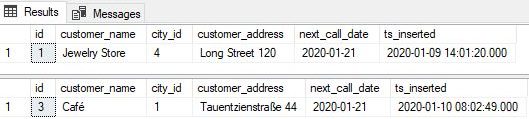
from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

);



The most important thing you should notice here is:

* A EXCEPT B is not the same as B EXCEPT A (A and B are names of the sets)
* The first query returns all customers having exactly 3 calls except those from Berlin, while the second query finds and returns customers from Berlin except those with exactly 3 calls

INNER JOIN

Let’s discuss these two queries:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT \*  FROM country, city  WHERE city.country\_id = country.id;    SELECT \*  FROM country  INNER JOIN city ON city.country\_id = country.id; |

The result they return is presented on the picture below:



Both queries return exactly the same result. This is not by accident but the result of the fact that this is the same query written in two different ways. Both ways are correct, and you can use any of them.

In the first query, we listed all tables we use in the FROM part of the query (FROM country, city) and then went with the join condition in the WHERE part of the query (WHERE city.country\_id = country.id). In case we forgot to write down this join condition, we would have the Cartesian product of both tables.

In the second query, we have only one table in the FROM part of the query (FROM country) and then we have the second table and the JOIN condition in the JOIN part of the query (INNER JOIN city ON city.country\_id = country.id).

While both queries are well-written, I would suggest that you always use INNER JOIN instead of listing tables and joining them in the WHERE part of the query. There are a few reasons for that:

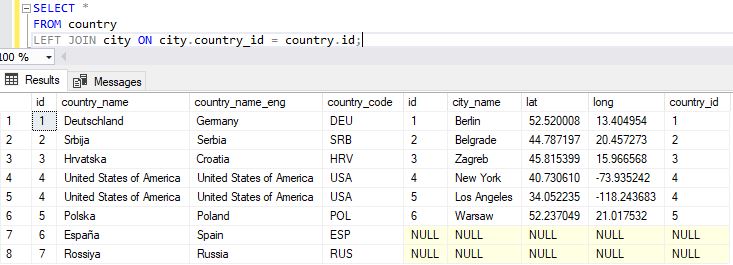
* Readability is much better because the table used and related JOIN condition are in the same line. You can easily see if you omitted the JOIN condition or not
* If you want to use other JOINs later (LEFT or RIGHT), you couldn’t do that (easily) unless you’ve used INNER JOIN before that

LEFT JOIN

I’ll repeat this – “We don’t have 2 countries on the list (Spain and Russia) because they don’t have any related city in the **city** table“. This shall prove crucial when comparing INNER JOIN vs LEFT JOIN.

In some cases, we want to have even these records in our results. For example, you simply want to see in the result that these countries don’t have related records in another table. This could be part of some control, or maybe just counting cases, etc. No matter what the motivation behind that desire is, we should be technically able to do that. And we are. In databases, LEFT JOIN does exactly that.

The result of LEFT JOIN shall be the same as the result of INNER JOIN + we’ll have rows, from the “left” table, without a pair in the “right” table. We’ll use the same INNER JOIN query and just replace the word INNER with LEFT. This is the result:

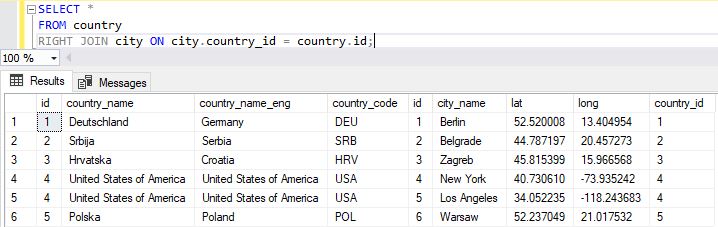


You can easily notice, that we have 2 more rows, compared to the result of the INNER JOIN query. These are rows for Russia and Spain. Since they both don’t have any related city, all city attributes in these two rows have NULL values (are not defined). That is the biggest difference when comparing INNER JOIN vs LEFT JOIN.

RIGHT JOIN

You’ll at least hear about the RIGHT JOIN. It’s rarely used because it returns the same result as the LEFT JOIN. On the other hand, queries which use LEFT JOIN are much easier to read because we simply list tables one after the other.

This is the equivalent of the previous query using the RIGHT JOIN:



You can notice that returned values are the same, only in this case values from the city table are in the first 5 columns, and country-related values come after them.

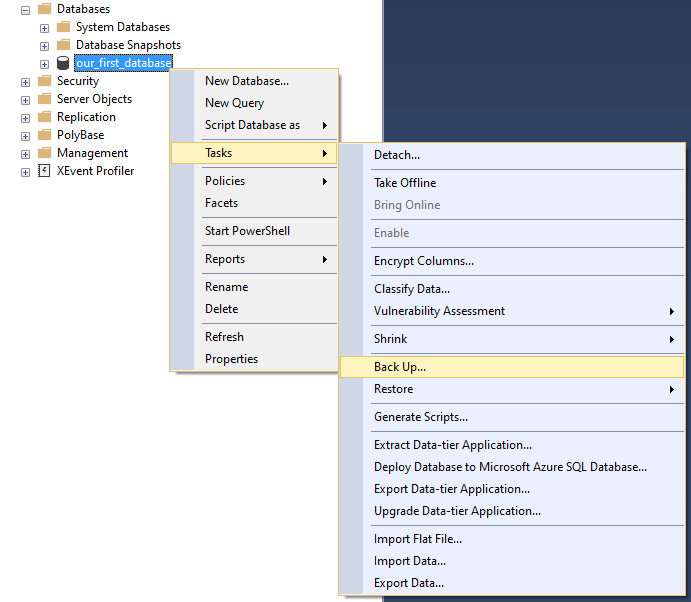
Backup and restore

Using scripts is usually related to making significant changes in the database. I might be paranoid about this, but I prefer to backup the database before these changes.

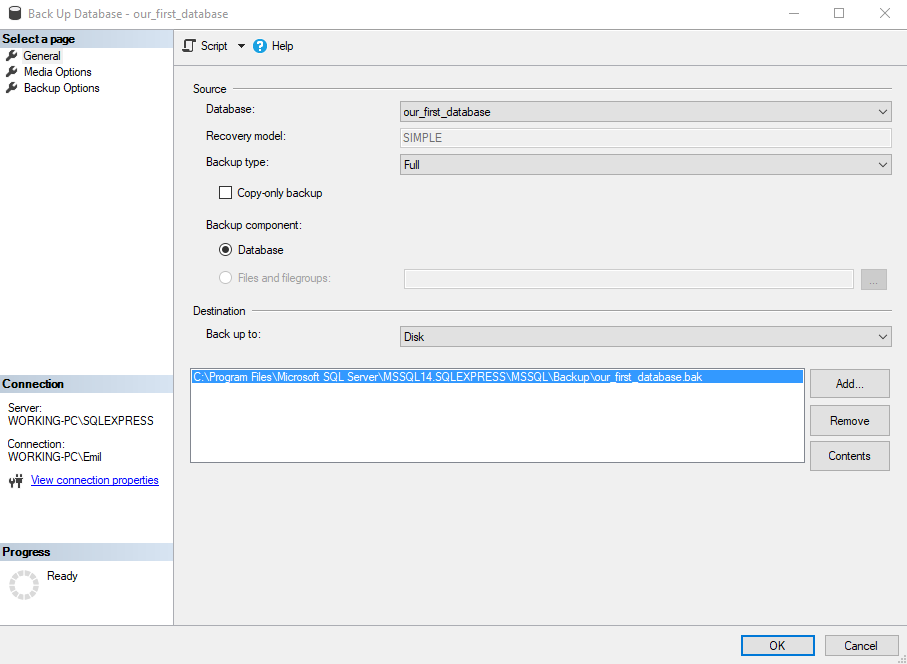
* ***Tip:****If you expect major changes in your databases, either in structure, either data changes, creating a backup is always a good idea. You could backup the entire database or only 1 table. That’s completely up to you and the changes you’re making.*

Oldie but goodie: *“There are two types of people: 1. People who do backup & 2. People who will start doing backup.”*

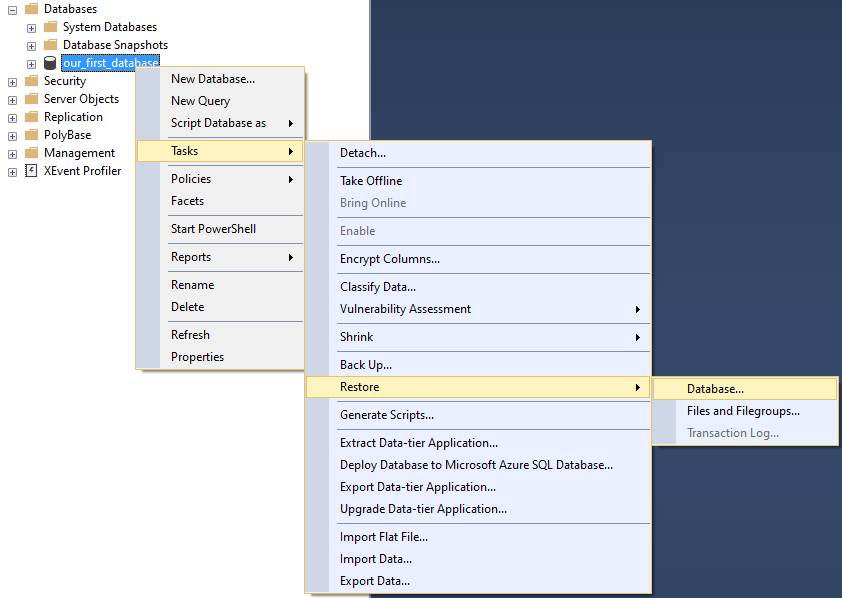
In SQL Server you can easily backup the whole database. First right-click on the database you want to backup, go to **Tasks**, and choose the **Back Up** option:



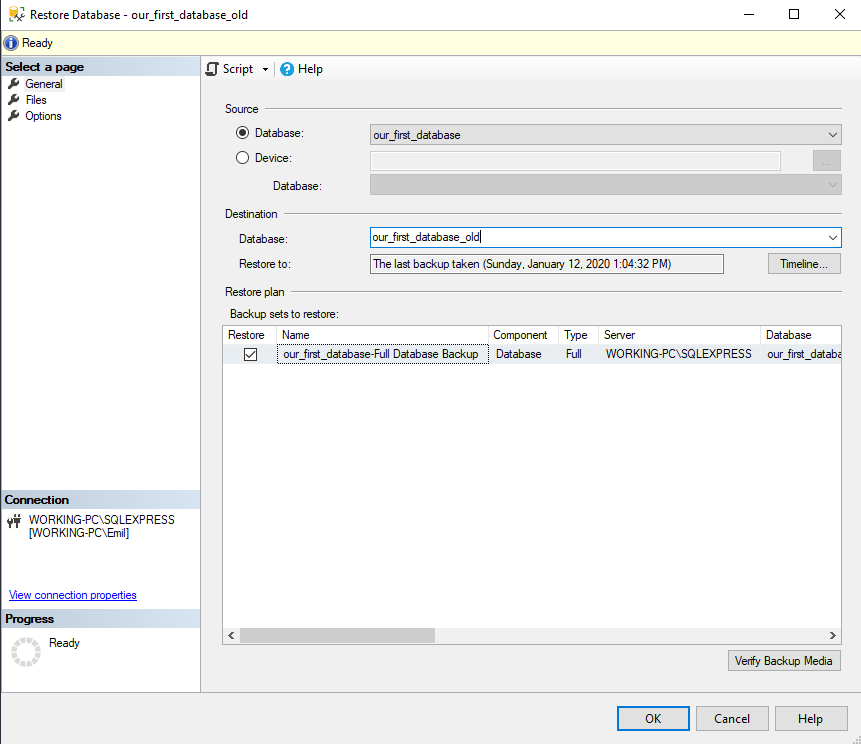
After choosing the backup option, you’ll get the pop-up as on the picture below:



Click **OK**, and your backup is created. Now we can create a copy of our database. We’ll do that by restoring from the backup. To do that, follow the process in the picture below:



You’ll need to enter the name of the restored database (**test\_database\_old**) and click **OK**:



The result of these actions shall be that now we have 2 identical databases – **test\_database** & **test\_database\_old**. We’ll use the first one as the one where we’ll run our SQL script. In case we would screw something up, we could always easily revert changes using our backup.

**SQL Script**

Now we’re ready to take a look at our script. We want to do two different things:

* Create new database objects (tables and relations) – DDL commands, and
* Populate these tables with data – DML commands

Of course, we’ll run DDL commands first and then run DML commands. Trying to insert data into a table that doesn’t exist would result in errors. So, let’s take a look at our script now:

-- tables

-- Table: call

CREATE TABLE call (

id int NOT NULL IDENTITY(1, 1),

employee\_id int NOT NULL,

customer\_id int NOT NULL,

start\_time datetime NOT NULL,

end\_time datetime NULL,

call\_outcome\_id int NULL,

CONSTRAINT call\_ak\_1 UNIQUE (employee\_id, start\_time),

CONSTRAINT call\_pk PRIMARY KEY (id)

);

-- Table: call\_outcome

CREATE TABLE call\_outcome (

id int NOT NULL IDENTITY(1, 1),

outcome\_text char(128) NOT NULL,

CONSTRAINT call\_outcome\_ak\_1 UNIQUE (outcome\_text),

CONSTRAINT call\_outcome\_pk PRIMARY KEY (id)

);

-- Table: customer

CREATE TABLE customer (

id int NOT NULL IDENTITY(1, 1),

customer\_name varchar(255) NOT NULL,

city\_id int NOT NULL,

customer\_address varchar(255) NOT NULL,

next\_call\_date date NULL,

ts\_inserted datetime NOT NULL,

CONSTRAINT customer\_pk PRIMARY KEY (id)

);

-- Table: employee

CREATE TABLE employee (

id int NOT NULL IDENTITY(1, 1),

first\_name varchar(255) NOT NULL,

last\_name varchar(255) NOT NULL,

CONSTRAINT employee\_pk PRIMARY KEY (id)

);

-- foreign keys

-- Reference: call\_call\_outcome (table: call)

ALTER TABLE call ADD CONSTRAINT call\_call\_outcome

FOREIGN KEY (call\_outcome\_id)

REFERENCES call\_outcome (id);

-- Reference: call\_customer (table: call)

ALTER TABLE call ADD CONSTRAINT call\_customer

FOREIGN KEY (customer\_id)

REFERENCES customer (id);

-- Reference: call\_employee (table: call)

ALTER TABLE call ADD CONSTRAINT call\_employee

FOREIGN KEY (employee\_id)

REFERENCES employee (id);

-- Reference: customer\_city (table: customer)

ALTER TABLE customer ADD CONSTRAINT customer\_city

FOREIGN KEY (city\_id)

REFERENCES city (id);

-- insert values

INSERT INTO call\_outcome (outcome\_text) VALUES ('call started');

INSERT INTO call\_outcome (outcome\_text) VALUES ('finished - successfully');

INSERT INTO call\_outcome (outcome\_text) VALUES ('finished - unsuccessfully');

INSERT INTO employee (first\_name, last\_name) VALUES ('Thomas (Neo)', 'Anderson');

INSERT INTO employee (first\_name, last\_name) VALUES ('Agent', 'Smith');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Jewelry Store', 4, 'Long Street 120', '2020/1/21', '2020/1/9 14:1:20');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Bakery', 1, 'Kurfürstendamm 25', '2020/2/21', '2020/1/9 17:52:15');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Café', 1, 'Tauentzienstraße 44', '2020/1/21', '2020/1/10 8:2:49');

INSERT INTO customer (customer\_name, city\_id, customer\_address, next\_call\_date, ts\_inserted) VALUES ('Restaurant', 3, 'Ulica lipa 15', '2020/1/21', '2020/1/10 9:20:21');

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 4, '2020/1/11 9:0:15', '2020/1/11 9:12:22', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 2, '2020/1/11 9:14:50', '2020/1/11 9:20:1', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 3, '2020/1/11 9:2:20', '2020/1/11 9:18:5', 3);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 1, '2020/1/11 9:24:15', '2020/1/11 9:25:5', 3);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 3, '2020/1/11 9:26:23', '2020/1/11 9:33:45', 2);

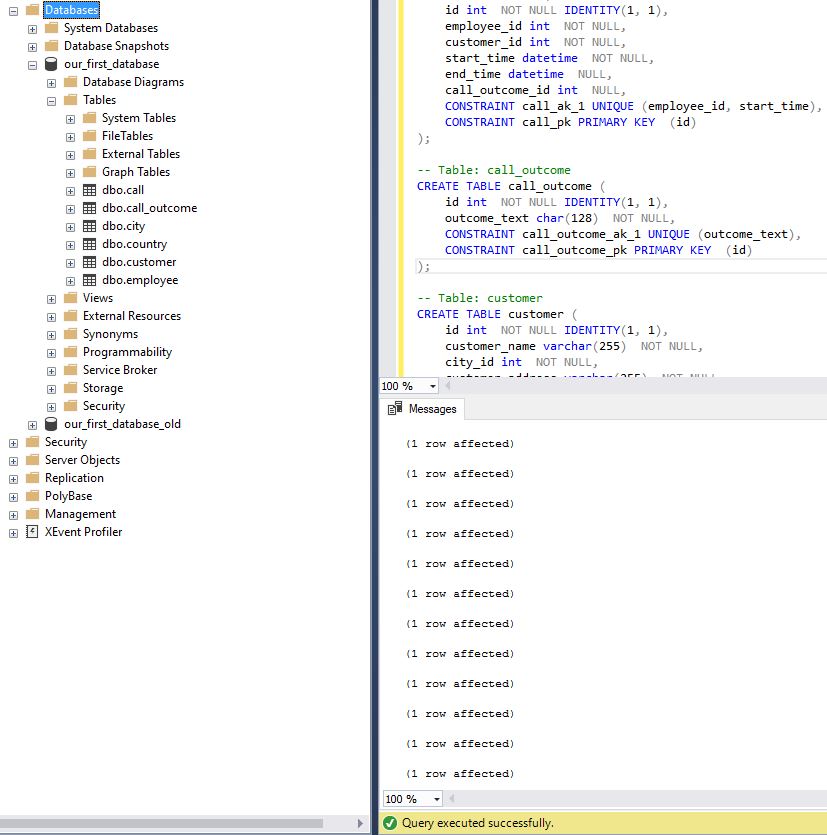
INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 2, '2020/1/11 9:40:31', '2020/1/11 9:42:32', 2);

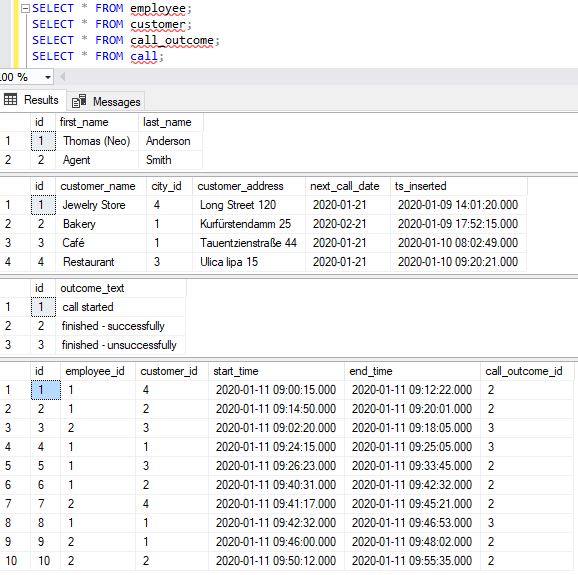
INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 4, '2020/1/11 9:41:17', '2020/1/11 9:45:21', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (1, 1, '2020/1/11 9:42:32', '2020/1/11 9:46:53', 3);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 1, '2020/1/11 9:46:0', '2020/1/11 9:48:2', 2);

INSERT INTO call (employee\_id, customer\_id, start\_time, end\_time, call\_outcome\_id) VALUES (2, 2, '2020/1/11 9:50:12', '2020/1/11 9:55:35', 2);

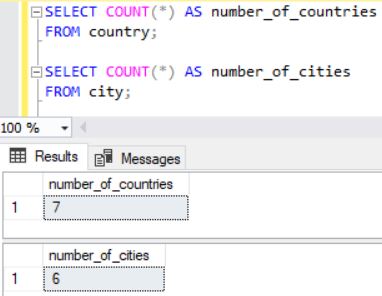




Aggregate Function

|  |  |
| --- | --- |
|  |  |

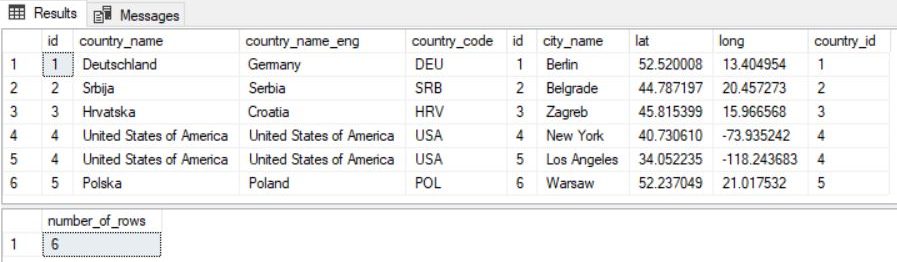




Aggregate Functions & JOINs

Now let’s try two more things. First, we’ll test how **COUNT** works when we’re joining tables. To do that, we’ll use the following queries:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT \*  FROM country  INNER JOIN city ON city.country\_id =  country.id;    SELECT COUNT(\*) AS number\_of\_rows  FROM country  INNER JOIN city ON city.country\_id =  country.id; |



SELECT \*

FROM country

LEFT JOIN city ON city.country\_id = country.id;

SELECT COUNT(\*) AS number\_of\_rows

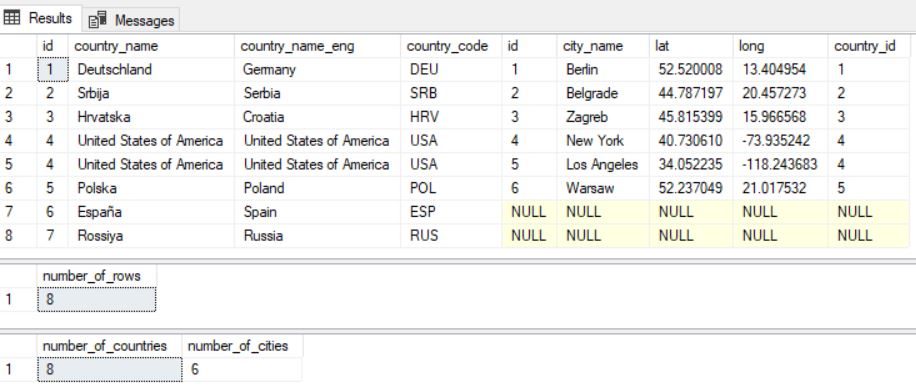
FROM country

LEFT JOIN city ON city.country\_id = country.id;

SELECT COUNT(country.country\_name) AS countries, COUNT(city.city\_name) AS cities

FROM country

LEFT JOIN city ON city.country\_id = country.id;



We can notice a few things:

* 1st query returned 8 rows. These are the same 6 rows as in a query using **INNER JOIN** and 2 more rows for countries that don’t have any related city (Russia & Spain)
* 2nd query counts the number of rows 1st query returns, so this number is 8
* 3rd query has two important things to comment on. The first one is that we’ve used aggregate function (**COUNT**), twice in the **SELECT** part of the query. This will usually be the case because you’re interested in more details about the group you want to analyze (number of records, average values, etc.). The second important thing is that these 2 counts used column names instead of “\*” and they returned different values. That happens because **COUNT** was created that way. If you put column names between brackets **COUNT** will count how many values are there (not including NULL values). All our records had value for country\_name, so the 1st COUNT returned 8. On the other hand, city\_name wasn’t defined 2 times (=NULL), so the 2nd COUNT returned 6 (8-2=6)
* ***Note:****This stands for other aggregate functions as well. If they run into NULL values, they will simply ignore them and calculate as they don’t exist.*

SQL Aggregate Functions

Now it’s time that we mention all T-SQL aggregate functions. The most commonly used are:

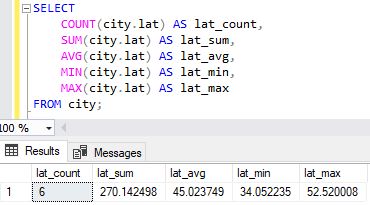
* **COUNT** – counts the number of elements in the group defined
* **SUM** – calculates the sum of the given attribute/expression in the group defined
* **AVG** – calculates the average value of the given attribute/expression in the group defined
* **MIN** – finds the minimum in the group defined
* **MAX** – finds the maximum in the group defined

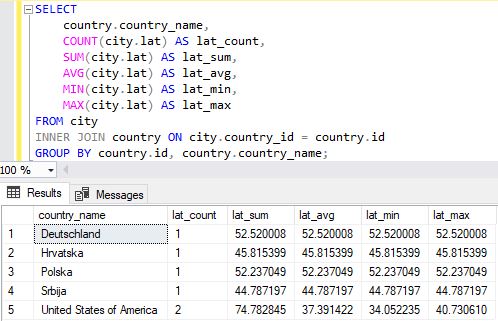
While all aggregate functions could be used without the **GROUP BY** clause, the whole point is to use the **GROUP BY** clause. That clause serves as the place where you’ll define the condition on how to create a group. When the group is created, you’ll calculate aggregated values.

* ***Example:****Imagine that you have a list of professional athletes and you know which sport each one of them plays. You could ask yourself something like – From my list, return the minimal, maximal and average height of players, grouped by the sport they play. The result would be, of course, MIN, MAX, and AVG height for groups – “football players”, “basketball players”, etc.*

Aggregate Functions – Examples

Now, let’s take a look at how these functions work on a single table. They are rarely used this way, but it’s good to see it, at least for educational purposes:





This is a much “smarter” query than the previous one. It returned the list of all countries, with a number of cities in them, as well as SUM, AVG, MIN, and MAX of their **lat** values.

Please notice that we’ve used the **GROUP BY** clause. By placing **country.id** and **country. country\_name**, we’ve defined a group. All cities belonging to the same country will be in the same group. After the group is created, aggregated values are calculated.

Complex Query

SELECT

country.country\_name\_eng,

SUM(CASE WHEN call.id IS NOT NULL THEN 1 ELSE 0 END) AS calls,

AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) AS avg\_difference

FROM country

LEFT JOIN city ON city.country\_id = country.id

LEFT JOIN customer ON city.id = customer.city\_id

LEFT JOIN call ON call.customer\_id = customer.id

GROUP BY

country.id,

country.country\_name\_eng

HAVING AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) > (SELECT AVG(DATEDIFF(SECOND, call.start\_time, call.end\_time)) FROM call)

ORDER BY calls DESC, country.id ASC;

SELECT

\*

FROM country

LEFT JOIN city ON city.country\_id = country.id

LEFT JOIN customer ON city.id = customer.city\_id

LEFT JOIN call ON call.customer\_id = customer.id;

Now we’ll write down the query which returns aggregated values for all countries.

SELECT

country.country\_name\_eng,

SUM(CASE WHEN call.id IS NOT NULL THEN 1 ELSE 0 END) AS calls,

AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) AS avg\_difference

FROM country

LEFT JOIN city ON city.country\_id = country.id

LEFT JOIN customer ON city.id = customer.city\_id

LEFT JOIN call ON call.customer\_id = customer.id

GROUP BY

country.id,

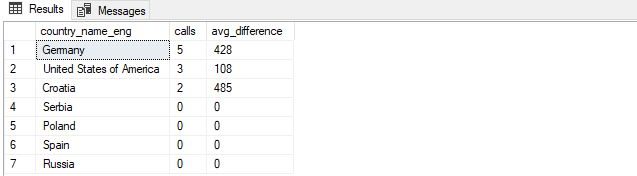
country.country\_name\_eng

ORDER BY calls DESC, country.id ASC;

I would like to point out two things here:

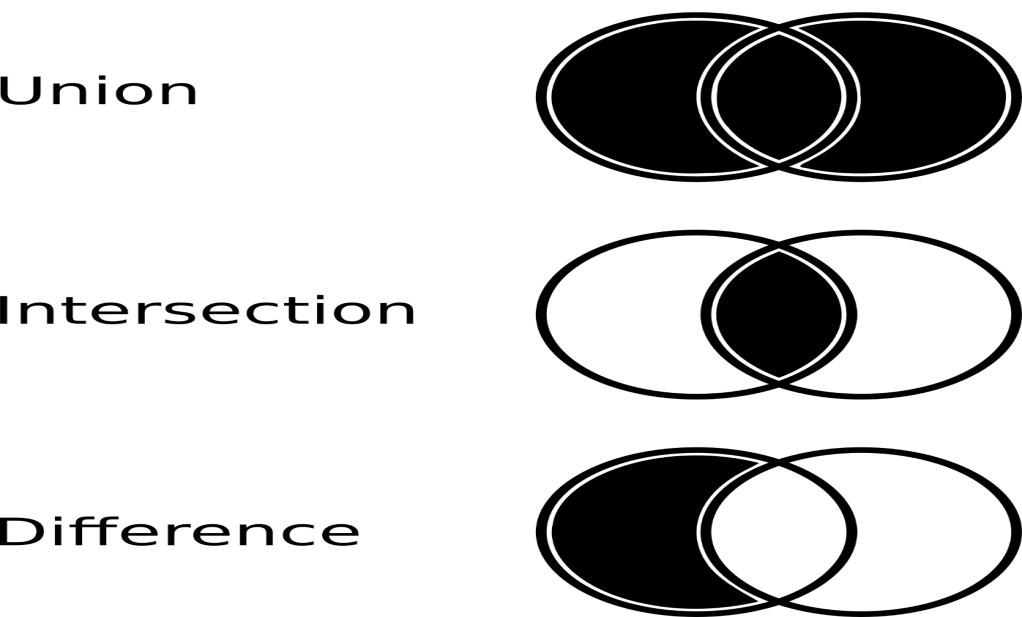
* SUM(CASE WHEN call.id IS NOT NULL THEN 1 ELSE 0 END) – This will sum up only existing calls. Since we’ve used LEFT JOIN, we’ll also join countries without any call. In case we’ve used COUNT, we would have value 1 returned for countries without any call, and we want 0 there (we want to see that info)
* AVG(ISNULL(DATEDIFF(SECOND, call.start\_time, call.end\_time),0)) – This is very similar to the previously mentioned AVG. The difference here is that I’ve used ISNULL(…,0). This simply tests if the calculated value IS NULL, and if so, replaces it with 0. Calculated value could be NULL if there is not data (we’ve used LEFT JOIN)

Let’s see what this query returns.



## Set Theory and Venn Diagrams

In SQL Server we have 3 important operators at our disposal – UNION (ALL), INTERSECT, and EXCEPT. They return the result of related operators from the set theory (on the picture below).



The easiest way to explain this is:

* UNION – Returns elements from both sets (if there are duplicates, they are in the final set, only once)
* UNION ALL – Same as the UNION operator, but will contain all duplicates
* INTERSECT – Returns a set containing elements that are present in both sets
* EXCEPT/MINUS (difference) – A MINUS B is a set containing elements from the set A that are not elements of the set B (so A MINUS (A INTERSECT B))

We won’t analyze situations where sets don’t have any common elements (A UNION B = all elements from A and B, A INTERSECT B = {}, A EXCEPT B = A, B EXCEPT A = B) and where set A = set B (A UNION B = A = B, A INTRSECT B = A = B, A EXCEPT B = B EXCEPT A = {}).

## Set Theory and SQL

We talked a lot about the set theory so far, and now it’s time for some practice. We’ll write down a few queries which will show how UNION (ALL), INTERSECT and EXCEPT operators work.

**#1 First we’ll test two separate queries and analyze the result set they return**

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

);

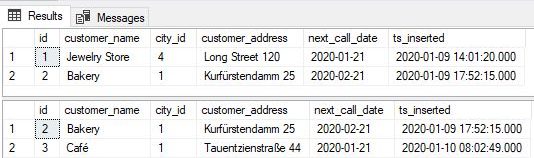
**-- list all customers from Berlin**

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';



You should notice a few things:

* The first query returns all customers having exactly 3 calls
* The second query returns all customers from Berlin
* Both queries return the same columns, but rows returned are not the same. This is important because you can use operators working with sets only if these two sets are composed of elements with the same structure
* Each result set has 2 rows. “Bakery” is present in both result sets, and each set has one other row

### #2 UNION and UNION ALL

Now we’ll use two available UNION operators. Any of these operators (UNION (ALL), INTERSECT, EXCEPT) is used in a way you just place it between queries

-- UNION

**-- list all customers with exactly 3 calls**

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

UNION

-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';

-- UNION ALL

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

UNION ALL

-- list all customers from Berlin

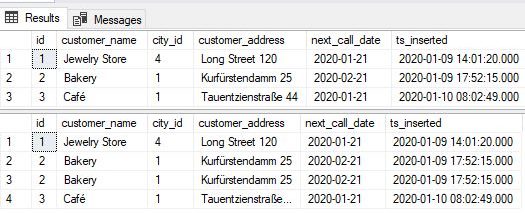
select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';.

|  |  |
| --- | --- |
|  |  |



For these two result sets returned, you should notice the following:

* Each query, the first one using UNION, and the second one using UNION ALL returns 1 result set
* The result set returned by the UNION query returned all rows returned by the two queries used. The only difference is that the duplicated row had been eliminated
* The query using UNION ALL returned all rows from both queries, without removing duplicates
* The UNION is used more often, and you’ll probably use it when you have a few complex queries and you simply want to “join” their results without writing a single more complex query

### #3 INTERSECT

The INTERSECT should return elements/rows which appear in both sets.

-- INTERSECT

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

INTERSECT

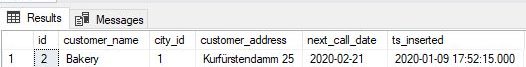
-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';



Everything went as expected and you can see that the “Bakery” row was returned as a result.

### #4 EXCEPT

The EXCEPT operator returns all elements/rows from the first set, except those that are in the second set.

-- A EXCEPT B

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

)

EXCEPT

-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin';

-- B EXCEPT A

-- list all customers from Berlin

select customer.\*

from customer

inner join city on customer.city\_id = city.id

where city.city\_name = 'Berlin'

EXCEPT

-- list all customers with exactly 3 calls

select customer.\*

from customer

where id in (

select customer.id

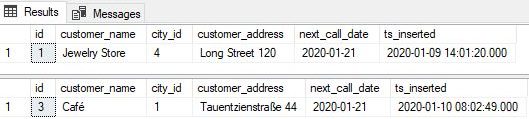
from customer

inner join call on customer.id = call.customer\_id

group by customer.id

having count(\*) = 3

);



The most important thing you should notice here is:

* A EXCEPT B is not the same as B EXCEPT A (A and B are names of the sets)
* The first query returns all customers having exactly 3 calls except those from Berlin, while the second query finds and returns customers from Berlin except those with exactly 3 calls