Lecture 5: Transactions, ACID, subqueries

Contents

DSCI 513
Databases & Data Retrieval

Lecture outline

- Transactions
- ACID properties
- Subqueries

Transactions: all or nothing

Let's consider the classical example of a bank database. Suppose that I bought a vintage guitar from my friend, Joel, for $1000, and now I want to transfer $1000 from my bank account to Joel's account using the banking app on my phone. This operation involves:

Skip to main content

2. Adding (i.e. crediting) $1000 to Joel's account.

We've already learned how to do this in SQL:

```
-- debiting source account
UPDATE
    accounts
SET
    balance = balance - 1000.0
WHERE
    account_number = 1132;
```

```
-- crediting target account
UPDATE
    accounts
SET
    balance = balance + 1000.0
WHERE
    account_number = 1279;
```

Now, what if a power outage happens right between the two `UPDATE` statements? In that case, my account would be left debited $1000, while Joel has also received no money. We'd both be furious!

This example nicely illustrates a common situation where we would want either **all** or **none** of the changes to be made to the database. In other words, the changes need to be considered and treated as **a single logical unit of work**: either all changes are successfully made, or all of them should fail.

In database (and in banking) terms, each unit of work is called a **transaction**.

Transactions in Postgres

In order to bundle the above update statements into one single transaction, we can use the

Skip to main content

```
BEGIN TRANSACTION;

-- debiting source account
UPDATE
    accounts
SET
    balance = balance - 1000.0
WHERE
    account_number = 1132;

-- crediting target account
UPDATE
    accounts
SET
    balance = balance + 1000.0
WHERE
    account_number = 1279;

COMMIT;
```

With the above syntax, now the DBMS knows how to treat the SQL commands: both update commands are a part of a larger unit of work. Either both changes will be made, or none.

The overall structure of a transaction is as follows:

```
[{BEGIN|START} [TRANSACTION]];

<SQL statements>

{COMMIT|ROLLBACK};
```

The `BEGIN` and `START` can be used interchangeably, although `BEGIN` is much more common. Moreover, the `BEGIN TRANSACTION` statement is optional. You can also start a transaction just using the command `BEGIN;` without the `TRANSACTION` keyword.

Commit: All data modifications during the transaction should be made permanent in the

Skip to main content

Rollback: All changes made during the transaction should be discarded, and the database is restored to its original state before the transaction.

```
import pandas as pd
import json
import psycopg2
import urllib.parse

with open('data/credentials.json') as f:
    login = json.load(f)

username = login['user']
password = urllib.parse.quote(login['password'])
host = login['host']
port = login['port']
```

```
%load_ext sql
%config SqlMagic.displaylimit = 30
```

```
%sql postgresql://{username}:{password}@{host}:{port}/mds
```

```
'Connected: postgres@mds'
```

```
%%sql

DROP TABLE IF EXISTS
    instructor,
    instructor_course,
    course_cohort
;

CREATE TABLE instructor (
    id INTEGER PRIMARY KEY,
    name TEXT,
    email TEXT,
    phone VARCHAR(12),
    department VARCHAR(50)
    )
.
```

Skip to main content

```
    instructor (id, name, email, phone, department)
VALUES
    (1, 'Mike', 'mike@mds.ubc.ca', '605-332-2343', 'Computer Science'),
    (2, 'Tiffany', 'tiff@mds.ubc.ca', '445-794-2233', 'Neuroscience'),
    (3, 'Arman', 'arman@mds.ubc.ca', '935-738-5796', 'Physics'),
    (4, 'Varada', 'varada@mds.ubc.ca', '243-924-4446', 'Computer Science'),
    (5, 'Quan', 'quan@mds.ubc.ca', '644-818-0254', 'Economics'),
    (6, 'Joel', 'joel@mds.ubc.ca', '773-432-7669', 'Biomedical Engineering'),
    (7, 'Florencia', 'flor@mds.ubc.ca', '773-926-2837', 'Biology'),
    (8, 'Alexi', 'alexiu@mds.ubc.ca', '421-888-4550', 'Statistics'),
    (15, 'Vincenzo', 'vincenzo@mds.ubc.ca', '776-543-1212', 'Statistics'),
    (19, 'Gittu', 'gittu@mds.ubc.ca', '776-334-1132', 'Biomedical Engineering'
    (16, 'Jessica', 'jessica@mds.ubc.ca', '211-990-1762', 'Computer Science')
;


CREATE TABLE instructor_course (
    id SERIAL PRIMARY KEY,
    instructor_id INTEGER,
    course TEXT,
    enrollment INTEGER,
    begins DATE
    )
;

INSERT INTO
    instructor_course (instructor_id, course, enrollment, begins)
VALUES
    (8, 'Statistical Inference and Computation I', 125, '2021-10-01'),
    (8, 'Regression II', 102, '2022-02-05'),
    (1, 'Descriptive Statistics and Probability', 79, '2021-09-10'),
    (1, 'Algorithms and Data Structures', 25, '2021-10-01'),
    (3, 'Algorithms and Data Structures', 25, '2021-10-01'),
    (3, 'Python Programming', 133, '2021-09-07'),
    (3, 'Databases & Data Retrieval', 118, '2021-11-16'),
    (6, 'Visualization I', 155, '2021-10-01'),
    (6, 'Privacy, Ethics & Security', 148, '2022-03-01'),
    (2, 'Programming for Data Manipulation', 160, '2021-09-08'),
    (7, 'Data Science Workflows', 98, '2021-09-15'),
    (2, 'Data Science Workflows', 98, '2021-09-15'),
    (12, 'Web & Cloud Computing', 78, '2022-02-10'),
    (10, 'Introduction to Optimization', NULL, '2022-09-01'),
    (9, 'Parallel Computing', NULL, '2023-01-10'),
    (13, 'Natural Language Processing', NULL, '2023-09-10')
;

CREATE TABLE course_cohort (
    id INTEGER,
    cohort VARCHAR(7)
    )
;

INSERT INTO
    course_cohort (id, cohort)
```

Skip to main content

```
    (8, 'MDS-CL'),
    (1, 'MDS-CL'),
    (3, 'MDS-CL'),
    (1, 'MDS-V'),
    (9, 'MDS-V'),
    (9, 'MDS-V'),
    (3, 'MDS-V')
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
Done.
11 rows affected.
Done.
16 rows affected.
Done.
8 rows affected.
```

```
[]
```

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
11 rows affected.
```

Skip to main content

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science

```
conn = psycopg2.connect(database='mds', **login)
```

Let's take a look at the following example. Here I begin a transaction and try to set all phone numbers in the `instructor` table to `NULL`:

```
cur = conn.cursor()
cur.execute("""
    BEGIN TRANSACTION;

    UPDATE
        instructor
    SET
        phone = NULL;
""")
```

Within the current database session/connection, the changes are visible:

Skip to main content

```
cur.execute("SELECT * FROM instructor")
cur.fetchall()
```

```
[(1, 'Mike', 'mike@mds.ubc.ca', None, 'Computer Science'),
 (2, 'Tiffany', 'tiff@mds.ubc.ca', None, 'Neuroscience'),
 (3, 'Arman', 'arman@mds.ubc.ca', None, 'Physics'),
 (4, 'Varada', 'varada@mds.ubc.ca', None, 'Computer Science'),
 (5, 'Quan', 'quan@mds.ubc.ca', None, 'Economics'),
 (6, 'Joel', 'joel@mds.ubc.ca', None, 'Biomedical Engineering'),
 (7, 'Florencia', 'flor@mds.ubc.ca', None, 'Biology'),
 (8, 'Alexi', 'alexiu@mds.ubc.ca', None, 'Statistics'),
 (15, 'Vincenzo', 'vincenzo@mds.ubc.ca', None, 'Statistics'),
 (19, 'Gittu', 'gittu@mds.ubc.ca', None, 'Biomedical Engineering'),
 (16, 'Jessica', 'jessica@mds.ubc.ca', None, 'Computer Science')]
```

We can see that the phone number column in all of the above tuples returned by psycopg2 is set to `NULL` (shown as `None` in Python).

However, if we look at the database from the viewpoint of any other connection/session/user, no changes seem to have been made:

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
11 rows affected.
```

Skip to main content

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science

Well, that's because the changes we've made in our original connection are not made permanent until we issue a `COMMIT` command:

```
cur.execute("COMMIT;")
```

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
11 rows affected.
```

Skip to main content

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	None	Computer Science
2	Tiffany	tiff@mds.ubc.ca	None	Neuroscience
3	Arman	arman@mds.ubc.ca	None	Physics
4	Varada	varada@mds.ubc.ca	None	Computer Science
5	Quan	quan@mds.ubc.ca	None	Economics
6	Joel	joel@mds.ubc.ca	None	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	None	Biology
8	Alexi	alexiu@mds.ubc.ca	None	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	None	Statistics
19	Gittu	gittu@mds.ubc.ca	None	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	None	Computer Science

Voila, the changes are now permanently recorded in the database, and visible by any user/connection. Note that instead of running `COMMIT;` as a SQL statement, we could have run `conn.commit()`.

Here's an example of `ROLLBACK`. I'm going to delete all rows from the `instructor` table in a transaction:

```
cur.execute("""
    BEGIN TRANSACTION;

    DELETE FROM instructor;
    """)
```

```
cur.execute("SELECT * FROM instructor")
cur.fetchall()
```

```
[]
```

Skip to main content

As expected, there are no rows left in the instructor table.

Since the transaction is still ongoing and changes have not been made permanent, I can **abort the transaction** and rollback to bring the database back to its original state before the transaction:

```
cur.execute("ROLLBACK;")
```

```
cur.execute("SELECT * FROM instructor")
cur.fetchall()
```

```
[(1, 'Mike', 'mike@mds.ubc.ca', None, 'Computer Science'),
 (2, 'Tiffany', 'tiff@mds.ubc.ca', None, 'Neuroscience'),
 (3, 'Arman', 'arman@mds.ubc.ca', None, 'Physics'),
 (4, 'Varada', 'varada@mds.ubc.ca', None, 'Computer Science'),
 (5, 'Quan', 'quan@mds.ubc.ca', None, 'Economics'),
 (6, 'Joel', 'joel@mds.ubc.ca', None, 'Biomedical Engineering'),
 (7, 'Florencia', 'flor@mds.ubc.ca', None, 'Biology'),
 (8, 'Alexi', 'alexiu@mds.ubc.ca', None, 'Statistics'),
 (15, 'Vincenzo', 'vincenzo@mds.ubc.ca', None, 'Statistics'),
 (19, 'Gittu', 'gittu@mds.ubc.ca', None, 'Biomedical Engineering'),
 (16, 'Jessica', 'jessica@mds.ubc.ca', None, 'Computer Science')]
```

For us Postgres users through psycopg2 in Python, it is important not to leave transactions open unintentionally. By default, even a `SELECT` statement begins a transaction (see here), and unless the transaction is committed or rolled back explicitly, the session remains in the undesirable idle transaction state.

In order to avoid an unintentionally long-running transaction, we can do one of the following things:

- Use auto-commit mode by running `conn.autocommit = True`. In this mode, every executed statement is automatically committed if successful, and rolled back if an error occurs.
- Use `with` context manager for the connection and the cursor:

  ```
  with conn, conn.cursor() as cur:
      cur.execute("SELECT * FROM instructor;")
  ```

Skip to main content

The transaction is committed when a connection exits the `with` block with no raised exceptions, otherwise the transaction is rolled back.

```
with conn, conn.cursor() as cur:
    cur.execute("DELETE FROM instructor;")
```

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
0 rows affected.
```

id	name	email	phone	department

Note: Exiting the `with` block does not close a connection (as opposed to reading files using `with`). To close a connection, you need to explicitly run `conn.close()`.

Note: `psql` and `ipython-sql` run in auto-commit mode by default.

```
%config SqlMagic.autocommit
```

```
True
```

ACID

In the database world, a transaction must have a number of properties to guarantee data integrity and validity, which ensure that in the event of any sort of database failure, the data will not be corrupt or lost. These properties are **atomicity, consistency, isolation, and durability (ACID)**.

Skip to main content

Atomicity

This is the all-or-nothing property of a transaction that we've seen earlier. Transactions should be atomic, meaning that the whole transaction is supposed to be treated as a single unit of work. Either all or none of the operations inside a transaction are executed.

Example: We have a database with one table (*Values*) and two columns (**a** and **b**). We have an operation where, on a particular row, a value is subtracted from **a** and added to **b**. This might be similar to a banking transaction, where I withdraw money from my account to pay for a new pair of slippers. The operation takes place in two steps:

1. Remove value from **a**
2. Place value in **b**

Atomicity ensures that if a system failure occurs between steps one and two, or if either step one or step two is invalid for any other reason, the entire transaction will not be processed.

Consistency

You've seen in the previous section that some constraints were temporarily violated inside a transaction, but by the end of the transaction all constraints were respected. It is the responsibility of a transaction to maintain data integrity. If a transaction makes the database inconsistent or invalid (i.e. in violation of the constraints), the transaction is aborted and the database is brought back to the consistent state before the transaction.

Example: We have a table *marks* that gives the breakdown of marking schemes for various courses. It is divided into columns for **quiz**, **participation**, **midterm** and **final**. All fields must be integers, and, because these values make up the course marking scheme, each row must

Skip to main content

A database that ensures consistency would reject any transaction where the final result of the transaction results in a *marks* sum <> 100.

Isolation

Each database modification operation is either implicitly or explicitly run within a transaction. The DBMS guarantees that each transaction is kept separate and isolated from any other transaction. No other transaction can interfere with the current transaction that is in progress. Although it would be ideal to have the highest levels of isolation, this would cause serious performance issues in real-world applications. So depending on the situation, there are various levels of isolation that can be used as a trade-off between full isolation of transactions and concurrency.

Example: In that same *marks* table as before, let's assume that a department has made a decision that all **final** exam scores will be reduced by 5% and that that value will be shifted to **participation**. At the same time, an individual instructor in the program has decided to increase the weighting of the **final** and reduce the weighting of the **midterm**.

If these operations are not isolated, it is possible that the program-level transaction reduces the weighting of the final exam.

Durability

The changes that are made by a successful transaction are permanent, even if there is a failure and the database shuts down. In Postgres and other relational databases, this is achieved by using transaction logs.

Example: Data that goes in following a successfully completed transaction doesn't come out

Skip to main content

Subqueries

There is a particular type of question that we have avoided so far for our SQL queries, and that is one for which we need the result of a second query to be able to run the first query. Take the following question as an example:

Example: Using the `world` database, find the countries with surface area above the average value of all countries in the world.

This query looks simple. You might be tempted to try

```
-- This will NOT work
SELECT
    name
FROM
    country
WHERE
    surfacearea > AVG(surfacearea)
;
```

but we've learned before that aggregate functions **cannot** be used within a `WHERE` clause.

To answer this question, we need to query the database twice: once to obtain the average surface area, and once to actually retrieve the rows that satisfy the condition. However, we don't need to do that in two separate queries and manually take the data coming from the first query and use it in the second. We can use a **subquery** to do that for us.

Skip to main content

A subquery is a `SELECT` statement that is incorporated into another SQL statement. For example, the query that computes the average surface area is:

```
SELECT
    AVG(surfacearea)
FROM
    country
;
```

We can use this intermediate information in our original query by embedding the above query in the `WHERE` clause of the original query:

```
SELECT
    name
FROM
    country
WHERE
    surfacearea > (
        SELECT AVG(surfacearea) FROM country
    )
;
```

```
%sql postgresql://{username}:{password}@{host}:{port}/world
```

```
'Connected: postgres@world'
```

```
%%sql

SELECT
    name
FROM
    country
WHERE
    surfacearea > (
        SELECT
            AVG(surfacearea)
        FROM
            country
    )
;
```

Skip to main content

```
43 rows affected.
```

Skip to main content

name
Afghanistan
Algeria
Angola
Argentina
Australia
Bolivia
Brazil
Chile
Egypt
South Africa
Ethiopia
Greenland
Indonesia
India
Iran
Canada
Kazakstan
China
Colombia
Congo, The Democratic Republic of the
Libyan Arab Jamahiriya
Mali
Mauritania
Mexico
Mongolia
Mozambique
Myanmar

Skip to main content

name
Namibia
Niger
Nigeria

43 rows, truncated to displaylimit of 30

Note that:

- A subquery should always be enclosed in parentheses, e.g. `(SELECT ...)`
- Subqueries should **not** be terminated by a semi-colon, as opposed to regular queries
- Sometimes the main SQL statement is called the **outer query** and the subquery is called the **inner query**

A subquery can be used in the `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses. Subqueries are most commonly used in the `WHERE` clause.

In our last query, we could have alternatively used a subquery in the `SELECT` clause to check if the returned rows do satisfy the `surfacearea > AVG(surfacearea)` condition:

```
%%sql

SELECT
    name,
    ROUND(surfacearea::NUMERIC / (SELECT AVG(surfacearea) FROM country)::NUMER
        AS ratio
FROM
    country
WHERE
    surfacearea > (
        SELECT AVG(surfacearea) FROM country
    )
ORDER BY
```

Skip to main content

```
    ratio
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
43 rows affected.
```

Skip to main content

name	ratio
Somalia	1.02
Afghanistan	1.05
Myanmar	1.09
Zambia	1.21
Chile	1.21
Turkey	1.24
Pakistan	1.28
Mozambique	1.29
Namibia	1.32
Tanzania	1.42
Venezuela	1.46
Nigeria	1.48
Egypt	1.61
Mauritania	1.65
Bolivia	1.76
Ethiopia	1.77
Colombia	1.83
South Africa	1.96
Mali	1.99
Angola	2.00
Niger	2.03
Chad	2.06
Peru	2.06
Mongolia	2.51
Iran	2.64
Libyan Arab Jamahiriya	2.82
Indonesia	3.06

Skip to main content

name	ratio
Mexico	3.14
Saudi Arabia	3.45
Greenland	3.48

43 rows, truncated to displaylimit of 30

> **Note:** A subquery in the `SELECT` clause should always return a single value, not a column or rows of values.

Example: Retrieve the name of countries whose capital cities have a population larger than 5 million.

```
%%sql

SELECT
    name
FROM
    country
WHERE
    capital IN (
        SELECT id
        FROM city
        WHERE population > 5000000
    )
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
13 rows affected.
```

Skip to main content

name
United Kingdom
Egypt
Indonesia
Iran
Japan
China
Colombia
Congo, The Democratic Republic of the
South Korea
Mexico
Peru
Thailand
Russian Federation

Well, as you might have guessed, we can rewrite this query using a **join**:

```
%%sql

SELECT
    co.name
FROM
    country co
JOIN
    city ci
ON
    co.capital = ci.id
WHERE
    ci.population > 5000000
;
```

```
postgresql://postgres:***@localhost:5432/mds
```

Skip to main content

```
13 rows affected.
```

name
United Kingdom
Egypt
Indonesia
Iran
Japan
China
Colombia
Congo, The Democratic Republic of the
South Korea
Mexico
Peru
Thailand
Russian Federation

Using a subquery is actually another way to gain access to data stored in other tables.

Typically, joins can be rewritten as a subquery and vice-versa, so what's the difference?

- Subqueries tend to be more readable and more intuitive
- Subqueries cannot be used if you need to include columns from the inner query in your results

Example Retrieve the names of countries where English is an official language and has a

Skip to main content

```
%%sql

SELECT
    name
FROM
    country
WHERE
    population > 1000000
    AND
    code IN (
        SELECT
            countrycode
        FROM
            countrylanguage
        WHERE
            language = 'English'
            AND
            isofficial = True
    )
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
10 rows affected.
```

name
Australia
United Kingdom
South Africa
Hong Kong
Ireland
Canada
Lesotho
New Zealand
United States
Zimbabwe

Skip to main content

Correlated subqueries

Subqueries that we have seen so far are called **simple or uncorrelated subqueries**, as they are executed **once** and **independently** of the outer query.

It sometimes happens that we need data from each row of the outer query in the subquery. This is an instance of what's called a **correlated subquery**. In a correlated subquery, the subquery takes the the current row of the outer query, and executes over all rows of the inner query. When finished, the next row from the outer query is selected, and the subquery is executed entirely for that outer row again, and so on.

Example: Which countries have the largest population in the continent where they are located?

The query for the above example requires the population of each country to be compared with all other countries that are in the same continent. The comparison of each row, with every other row that meet a particular condition can be achieved with a correlated subquery:

```
%%sql

SELECT
    c1.name, c1.continent
FROM
    country c1
WHERE
    c1.population = (
        SELECT
            MAX(c2.population)
        FROM
            country c2
        WHERE
            c2.continent = c1.continent
    )
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
11 rows affected.
```

Skip to main content

name	continent
Australia	Oceania
Brazil	South America
China	Asia
Nigeria	Africa
Russian Federation	Europe
United States	North America
Antarctica	Antarctica
Bouvet Island	Antarctica
South Georgia and the South Sandwich Islands	Antarctica
Heard Island and McDonald Islands	Antarctica
French Southern territories	Antarctica

Example: Write a query that returns the most populated city listed for each country code in the `city` table.

```
%%sql

SELECT
    ci1.name, ci1.countrycode
FROM
    city ci1
WHERE
    ci1.population = (
        SELECT
            MAX(ci2.population)
        FROM
            city ci2
        WHERE
            ci2.countrycode = ci1.countrycode
    )
;
```

Skip to main content

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
232 rows affected.
```

Skip to main content

name	countrycode
Kabul	AFG
Amsterdam	NLD
Willemstad	ANT
Tirana	ALB
Alger	DZA
Tafuna	ASM
Andorra la Vella	AND
Luanda	AGO
South Hill	AIA
Saint John´s	ATG
Dubai	ARE
Buenos Aires	ARG
Yerevan	ARM
Oranjestad	ABW
Sydney	AUS
Baku	AZE
Nassau	BHS
al-Manama	BHR
Dhaka	BGD
Bridgetown	BRB
Antwerpen	BEL
Belize City	BLZ
Cotonou	BEN
Saint George	BMU
Thimphu	BTN
Santa Cruz de la Sierra	BOL
Sarajevo	BIH

Skip to main content

name	countrycode
Gaborone	BWA
São Paulo	BRA
London	GBR

232 rows, truncated to displaylimit of 30

Example: Use the above query to return the name of countries whose capital city is not their most populated city.

The outer query in a subquery can be the result of joining other tables. In this example, we first need to join `country` and `city` to list capital cities of each country:

```
%%sql

SELECT
    co.name, ci.name
FROM
    country co
JOIN
    city ci
ON
    co.capital = ci.id
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
232 rows affected.
```

Skip to main content

name	name_1
Afghanistan	Kabul
Netherlands	Amsterdam
Netherlands Antilles	Willemstad
Albania	Tirana
Algeria	Alger
American Samoa	Fagatogo
Andorra	Andorra la Vella
Angola	Luanda
Anguilla	The Valley
Antigua and Barbuda	Saint John´s
United Arab Emirates	Abu Dhabi
Argentina	Buenos Aires
Armenia	Yerevan
Aruba	Oranjestad
Australia	Canberra
Azerbaijan	Baku
Bahamas	Nassau
Bahrain	al-Manama
Bangladesh	Dhaka
Barbados	Bridgetown
Belgium	Bruxelles [Brussel]
Belize	Belmopan
Benin	Porto-Novo
Bermuda	Hamilton
Bhutan	Thimphu
Bolivia	La Paz
Bosnia and Herzegovina	Sarajevo

Skip to main content

name	name_1
Botswana	Gaborone
Brazil	Brasília
United Kingdom	London

232 rows, truncated to displaylimit of 30

In the next step, we need to check whether the population of that capital city is the maximum population of cities of that particular country or not. This can be achieved via a correlated subquery:

```
%%sql

SELECT
    co.name
FROM
    country co
JOIN
    city ci
ON
    co.capital = ci.id
WHERE
    ci.population <> (
        SELECT
            MAX(ci2.population)
        FROM
            city ci2
        WHERE
            ci.countrycode = ci2.countrycode
    )
ORDER BY
    co.population DESC
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
48 rows affected.
```

Skip to main content

name
China
India
United States
Brazil
Pakistan
Nigeria
Vietnam
Philippines
Turkey
South Africa
Tanzania
Canada
Sudan
Morocco
Australia
Kazakstan
Cameroon
Côte d Ivoire
Ecuador
Malawi
Belgium
Senegal
Bolivia
Switzerland
Hong Kong
Benin
New Zealand

Skip to main content

name
Jamaica
Oman
United Arab Emirates

48 rows, truncated to displaylimit of 30

Correlated subqueries are usually quite inefficient (remember time complexity of nested loops from DSCI 512?), but they can answer some interesting and complex questions which might otherwise be impossible to answer using SQL.

`ANY` and `ALL`

Syntax:

```
SELECT
    column_name(s)
FROM
    table_name
WHERE
    column_name operator {ALL|ANY} (
        SELECT
            column_name
        FROM
            table_name
        WHERE
            condition
    );
```

Example: Find all non-European countries whose population is larger than every European country.

Skip to main content

```
%%sql

SELECT
    name
FROM
    country
WHERE
    continent <> 'Europe'
    AND
    population > ALL (
        SELECT
            population
        FROM
            country
        WHERE
            continent = 'Europe'
    )
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
6 rows affected.
```

name
Brazil
Indonesia
India
China
Pakistan
United States

Example: Find all European countries whose population is smaller than at least one city in the US.

```
%%sql
```

Skip to main content

```
    name
FROM
    country
WHERE
    continent = 'Europe'
    AND
    population < ANY (
        SELECT
            population
        FROM
            city
        WHERE
            countrycode = (
                SELECT
                    code
                FROM
                    country
                WHERE
                    name ILIKE '%United%States'
            )
    )
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
26 rows affected.
```

Skip to main content

name
Albania
Andorra
Bosnia and Herzegovina
Faroe Islands
Gibraltar
Svalbard and Jan Mayen
Ireland
Iceland
Croatia
Latvia
Liechtenstein
Lithuania
Luxembourg
Macedonia
Malta
Moldova
Monaco
Norway
San Marino
Slovakia
Slovenia
Finland
Switzerland
Denmark
Holy See (Vatican City State)
Estonia

Skip to main content

Subqueries written with `ALL` or `ANY` and `<`, `<=`, `>`, and `>=` be rewritten using aggregations. For example, we can rewrite the above query as:

```
%%sql

SELECT
    name
FROM
    country
WHERE
    continent = 'Europe'
    AND
    population < (
        SELECT
            MAX(population)
        FROM
            city
        WHERE
            countrycode = (
                SELECT
                    code
                FROM
                    country
                WHERE
                    name ILIKE '%United%States'
            )
    )
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
26 rows affected.
```

Skip to main content

name
Albania
Andorra
Bosnia and Herzegovina
Faroe Islands
Gibraltar
Svalbard and Jan Mayen
Ireland
Iceland
Croatia
Latvia
Liechtenstein
Lithuania
Luxembourg
Macedonia
Malta
Moldova
Monaco
Norway
San Marino
Slovakia
Slovenia
Finland
Switzerland
Denmark
Holy See (Vatican City State)
Estonia

Skip to main content

Here are equivalencies between using `ALL` and `ANY` with `MAX` and `MIN` in subqueries:

Using `ALL` and `ANY`	Using `MIN` and `MAX`
`< ALL (subquery)`	`< MIN(values)`
`> ALL (subquery)`	`> MAX(values)`
`< ANY (subquery)`	`< MAX(values)`
`> ANY (subquery)`	`> MIN(values)`

> **Remember:** The fact that two queries achieve the same result does not mean that they are also the same in terms of performance. For example, using `> MAX()` is usually faster than `> ALL()`.

EXISTS

With subqueries, sometimes we don't care about the rows that are returned, but if a row is returned at all or not. The `EXISTS` and `NOT EXISTS` keyword in SQL provide this type of functionality for us, i.e., the check for **existence** of rows, not their values. Here is the syntax for an `EXISTS` subquery:

```
SELECT
    column_name(s)
FROM
    table_name
WHERE
    [NOT] EXISTS (
        SELECT column_name FROM table_name WHERE condition
```

Skip to main content

```
    )
;
```

If the subquery returns one or more rows, then the `WHERE` condition becomes `TRUE`.

Remember that:

- With `EXISTS`, the columns returned by the subquery do not matter at all, which is why we usually use `SELECT *` in the subquery.
- The subquery following `EXISTS` can be either simple or correlated, but typically they are correlated.

Example: Find all countries that have at least a city with a population greater that 5 million.

```
%%sql

SELECT
    co.name
FROM
    country co
WHERE
    EXISTS (
        SELECT
            *
        FROM
            city ci
        WHERE
            co.code = ci.countrycode
            AND
            ci.population > 5000000
    )
;
```

Skip to main content

```
18 rows affected.
```

name
Brazil
United Kingdom
Egypt
Indonesia
India
Iran
Japan
China
Colombia
Congo, The Democratic Republic of the
South Korea
Mexico
Pakistan
Peru
Thailand
Turkey
Russian Federation
United States

Example: Which countries speak at least one language that is not spoken in any other country in their continent?

It's best to first create a temporary table that stores the names of countries, continents and their languages:

Skip to main content

```
%%sql

DROP TABLE IF EXISTS ccl;

CREATE TEMPORARY TABLE ccl AS (
    SELECT
        co.name, co.continent, cl.language
    FROM
        country co
    JOIN
        countrylanguage cl
    ON
        co.code = cl.countrycode
    WHERE
        co.continent IN ('Asia', 'Europe')
)
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
Done.
441 rows affected.
```

```
[]
```

Now we can use `NOT EXISTS` to detect countries that have languages not spoken elsewhere in their continent. Note that because a subquery runs its query over all rows, we need to exclude the current row of the outer query, otherwise there would always be a country in the same continent speaking all languages of the country in the outer query, i.e. itself!

```
%%sql

SELECT
    t1.*
FROM
    ccl t1
WHERE
    NOT EXISTS (
        SELECT
            *
        FROM
            ccl t2
        WHERE
            t1.name <> t2.name
            AND
```

Skip to main content

```
            t1.language = t2.language
    )
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
129 rows affected.
```

Skip to main content

name	continent	language
Bhutan	Asia	Dzongkha
Philippines	Asia	Pilipino
Faroe Islands	Europe	Faroese
Georgia	Asia	Georgiana
Indonesia	Asia	Javanese
Iceland	Europe	Icelandic
Japan	Asia	Japanese
Kyrgyzstan	Asia	Kirgiz
Cyprus	Asia	Greek
Latvia	Europe	Latvian
Luxembourg	Europe	Luxembourgish
Maldives	Asia	Dhivehi
Malta	Europe	Maltese
Myanmar	Asia	Burmese
Sri Lanka	Asia	Singali
Taiwan	Asia	Min
Afghanistan	Asia	Dari
Netherlands	Europe	Fries
Bangladesh	Asia	Chakma
United Kingdom	Europe	Kymri
Brunei	Asia	Malay-English
Philippines	Asia	Cebuano
Ireland	Europe	Irish
Italy	Europe	Sardinian
Yemen	Asia	Soqutri
Jordan	Asia	Circassian
China	Asia	Zhuang

Skip to main content

name	continent	language
Laos	Asia	Mon-khmer
Monaco	Europe	Monegasque
Myanmar	Asia	Shan

129 rows, truncated to displaylimit of 30

We can also replace `t1.*` in the `SELECT` clause with `COUNT(DISTINCT name)` to find how many countries with such a property exist in the world.

To summarize, we use `EXISTS` when:

- We don't need the data from a related table. With joins, we always have access to the columns of the target tables as well.
- We just need to check existence, which can also be achieved by using outer joins and checking for nulls.

Example: Find city names that happen to be used in more than one country!

```
%%sql

SELECT
    DISTINCT ci1.name
FROM
    city ci1
WHERE
    EXISTS (
        SELECT
            ci2.name
        FROM
            city ci2
        WHERE
            ci2.name = ci1.name
            AND
            ci1.countrycode <> ci2.countrycode
    )
ORDER BY
    ci1.name DESC
```

Skip to main content

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
47 rows affected.
```

Skip to main content

name
York
Worcester
Victoria
Vancouver
Valencia
Tripoli
Toledo
Taiping
Santa Rosa
Santa Maria
Santa Clara
Santa Ana
San Miguel
San Mateo
San Juan
San Jose
San Fernando
San Felipe
Salem
Salamanca
Saint John´s
Richmond
Portsmouth
Plymouth
Newcastle
Mérida
Manzanillo

Skip to main content

name
Manchester
Los Angeles
London

47 rows, truncated to displaylimit of 30

Alternatively, we can use a self-join to find city names used in more than one country:

```
%%sql

SELECT
    DISTINCT ON (ci1.name) ci1.name,
    ci1.countrycode,
    ci2.name,
    ci2.countrycode
FROM
    city ci1
JOIN
    city ci2
ON
    ci2.name = ci1.name
    AND
    ci1.countrycode <> ci2.countrycode
ORDER BY
    ci1.name DESC
;
```

```
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
47 rows affected.
```

Skip to main content

name	countrycode	name_1	countrycode_1
York	CAN	York	GBR
Worcester	USA	Worcester	GBR
Victoria	MEX	Victoria	SYC
Vancouver	CAN	Vancouver	USA
Valencia	ESP	Valencia	VEN
Tripoli	LBY	Tripoli	LBN
Toledo	BRA	Toledo	PHL
Taiping	MYS	Taiping	TWN
Santa Rosa	PHL	Santa Rosa	USA
Santa Maria	BRA	Santa Maria	PHL
Santa Clara	USA	Santa Clara	CUB
Santa Ana	USA	Santa Ana	SLV
San Miguel	ARG	San Miguel	PHL
San Mateo	USA	San Mateo	PHL
San Juan	PRI	San Juan	ARG
San Jose	USA	San Jose	PHL
San Fernando	PHL	San Fernando	ARG
San Felipe	MEX	San Felipe	VEN
Salem	IND	Salem	USA
Salamanca	ESP	Salamanca	MEX
Saint John´s	CAN	Saint John´s	ATG
Richmond	CAN	Richmond	USA
Portsmouth	GBR	Portsmouth	USA
Plymouth	GBR	Plymouth	MSR
Newcastle	ZAF	Newcastle	AUS
Mérida	MEX	Mérida	VEN
Manzanillo	CUB	Manzanillo	MEX

Skip to main content

name	countrycode	name_1	countrycode_1
Manchester	USA	Manchester	GBR
Los Angeles	USA	Los Angeles	CHL
London	GBR	London	CAN

47 rows, truncated to displaylimit of 30

Set operators (OPTIONAL)

The set operations are used to combine different `SELECT` statements. All the three set operators require that the `SELECT` clause lists the same number of columns.

- `UNION` combines all rows from both selects into the same table. It removes duplicates. (use UNION ALL to keep duplicates).
- `INTERSECT` combines only rows that are present in both tables. It removes duplicates. (use INTERSECT ALL to keep duplicates).
- `EXCEPT` only keeps the rows that are in the first table but are not in the second table.