Bazy danych - powtórka

Zadania sqlbolt →

1. Find the `title` of each film
2. Find the `director` of each film
3. Find the `title` and `director` of each film
4. Find the `title` and `year` of each film
5. Find `all` the information about each film

```
SELECT title FROM movies;

SELECT director FROM movies;

SELECT title,director FROM movies;

SELECT title,year FROM movies;

SELECT * FROM movies;
```

1. Find the movie with a row `id` of 6
2. Find the movies released in the `year`s between 2000 and 2010
3. Find the movies **not** released in the `year`s between 2000 and 2010
4. Find the first 5 Pixar movies and their release `year`

```
SELECT * FROM movies
WHERE id = 6;

SELECT * FROM movies
```

```
WHERE year BETWEEN 2000 AND 2010;


SELECT * FROM movies
WHERE year NOT BETWEEN 2000 AND 2010;

SELECT * FROM movies
ORDER BY year
LIMIT 5;
```

1. Find all the Toy Story movies ✓
2. Find all the movies directed by John Lasseter
3. Find all the movies (and director) not directed by John Lasseter
4. Find all the WALL-* movies

```
SELECT * FROM movies
WHERE title LIKE 'toy stroy%';

SELECT * FROM movies
WHERE director IN ('John Lasseter'); // może być LIKE, jest cas

SELECT * FROM movies
WHERE director NOT IN ('John Lasseter');

SELECT * FROM movies
WHERE title like 'WALL-%';
```

1. List all directors of Pixar movies (alphabetically), without duplicates ✓
2. List the last four Pixar movies released (ordered from most recent to least)
3. List the **first** five Pixar movies sorted alphabetically
4. List the **next** five Pixar movies sorted alphabetically

```
SELECT DISTINCT director FROM movies
ORDER BY director ;

SELECT * FROM movies
ORDER BY year DESC
LIMIT 4;

SELECT * FROM movies
ORDER BY title
LIMIT 5;

SELECT * FROM movies
ORDER BY title ASC
LIMIT 5, OFFSET 5;
```

1. List all the Canadian cities and their populations
2. Order all the cities in the United States by their latitude from north to south
3. List all the cities west of Chicago, ordered from west to east
4. List the two largest cities in Mexico (by population)
5. List the third and fourth largest cities (by population) in the United States and their population

```
SELECT city,population FROM north_american_cities
WHERE COUNTRY LIKE 'Canada'

SELECT * FROM north_american_cities
WHERE COUNTRY LIKE 'United States'
ORDER BY Latitude DESC;

SELECT city,longitude FROM north_american_cities
WHERE longitude < -87.629798
ORDER BY longitude ASC;
```

```
SELECT * FROM north_american_cities
WHERE country LIKE 'Mexico'
ORDER BY population DESC
LIMIT 2;

SELECT * FROM north_american_cities
WHERE country LIKE 'United States'
ORDER BY population DESC
LIMIT 2 OFFSET 2;
```

1. Find the domestic and international sales for each movie ✓
2. Show the sales numbers for each movie that did better internationally rather than domestically
3. List all the movies by their ratings in descending ord

```
SELECT title,domestic_sales,international_sales FROM movies
INNER JOIN boxoffice
    ON movies.id = boxoffice.movie_id;

SELECT title,domestic_sales,international_sales FROM movies
INNER JOIN boxoffice
    ON movies.id = boxoffice.movie_id
WHERE international_sales > domestic_sales;

SELECT title,rating FROM movies
INNER JOIN boxoffice
    ON movies.id = boxoffice.movie_id
ORDER BY rating DESC;
```

1. Find the list of all buildings that have employees ✓
2. Find the list of all buildings and their capacity
3. List all buildings and the distinct employee roles in each building (including empty buildings)

```
SELECT DISTINCT building_name FROM buildings
LEFT JOIN employees
    ON employees.building = buildings.building_name
WHERE name NOT NULL;

//LUB

SELECT DISTINCT building FROM employees;

SELECT building_name,capacity FROM buildings;


SELECT DISTINCT building_name, role
FROM buildings
  LEFT JOIN employees
    ON building_name = building;
```

1. Find the name and role of all employees who have not been assigned to a building ✓
2. Find the names of the buildings that hold no employees

```
SELECT * FROM employees
WHERE BUILDING IS NULL;

SELECT * from buildings
    LEFT JOIN employees
    ON building_name=building
WHERE name IS NULL;
```

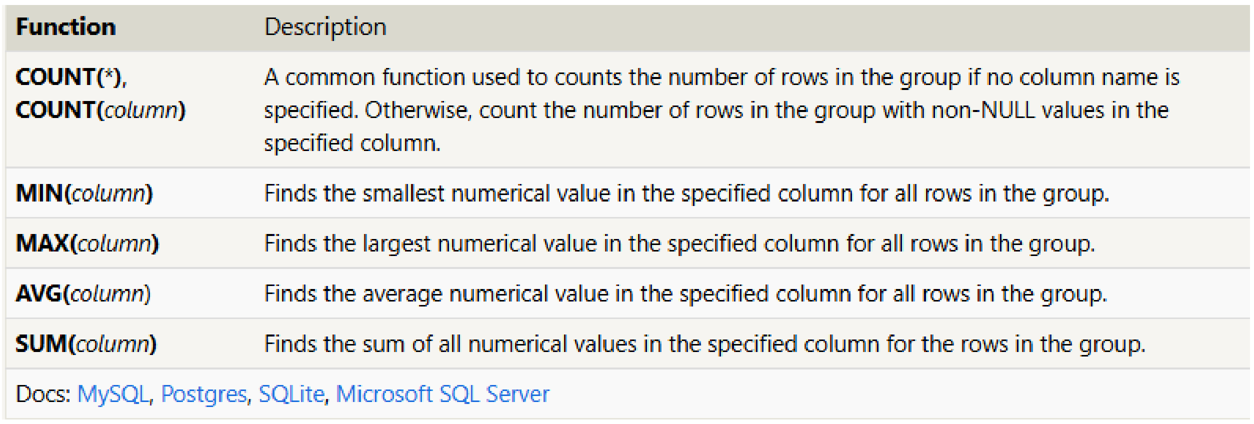
Queries with expressions: + aliasy (możemy aliasować kolumny)

1. List all movies and their combined sales in **millions** of dollars
2. List all movies and their ratings **in percent**

3. List all movies that were released on even number years

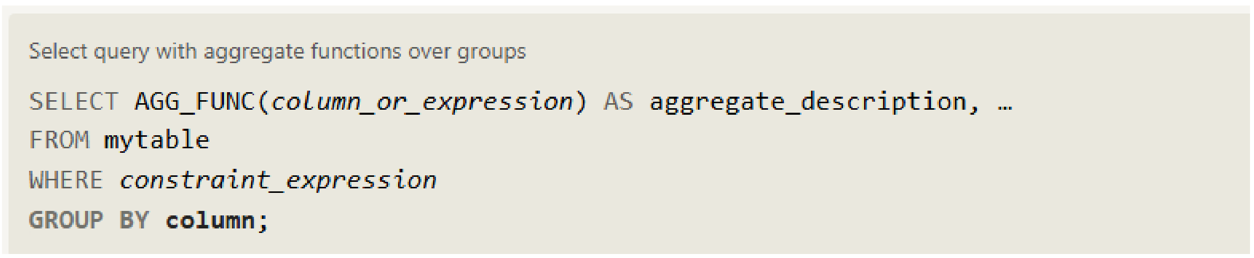
```
SELECT title,(domestic_sales+international_sales)/1000000 AS al
    INNER JOIN boxoffice
    ON id=movie_id;

SELECT title, (rating*10) AS Percent FROM movies
    JOIN boxoffice
    ON id=movie_id

SELECT * FROM movies
    join boxoffice
    on id=movie_id
WHERE year%2=0
```

Queries with aggregates - grupują, count, sum itp.



Function	Description
COUNT(*****), **COUNT(***column***)**	A common function used to counts the number of rows in the group if no column name is specified. Otherwise, count the number of rows in the group with non-NULL values in the specified column.
MIN(*column***)**	Finds the smallest numerical value in the specified column for all rows in the group.
MAX(*column***)**	Finds the largest numerical value in the specified column for all rows in the group.
AVG(*column***)**	Finds the average numerical value in the specified column for all rows in the group.
SUM(*column***)**	Finds the sum of all numerical values in the specified column for the rows in the group.
Docs: MySQL, Postgres, SQLite, Microsoft SQL Server	

Najczęściej używa się z **GROUP BY → tak o:**



Select query with aggregate functions over groups

```
SELECT AGG_FUNC(column_or_expression) AS aggregate_description, …
FROM mytable
WHERE constraint_expression
GROUP BY column;
```

Czyli:

1. Wyniki grupujemy
2. Dla każdej grupy agregujemy dane

np. Grupujemy piłkarzy na buckety, na podstawie pierwszej litery nazwiska (grupowanie) i następnie sumujemy (agregujemy). Jak mamy ochotę, możemy

Tylko kolejność jest odwrotna bo najpierw podajemy co agregujemy (ma to być suma, średnia, ilość itp.)

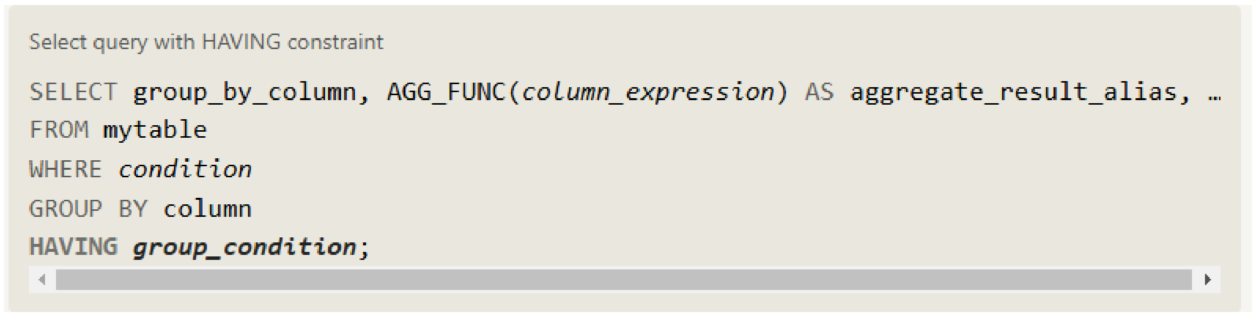
1. Find the longest time that an employee has been at the studio ✓
2. For each role, find the average number of years employed by employees in that role
3. Find the total number of employee years worked in each building

```
SELECT name, years_employed FROM Employees
ORDER BY years_employed desc
Limit 1;
//LUB lepiej, funkcja agregująca MAX()
SELECT name, MAX(years_employed) FROM employees;

    //-- 1.Grupujemy dla roli -> Group By
//-- 2.Agregujemy -> AVG()

SELECT role, AVG(years_employed) as avg
FROM employees
GROUP BY role;

//Znajdz sumę godzin spędzonych w każdym budynku przez pracowni
SELECT building, SUM(years_employed)
FROM Employees
GROUP BY building;
```

Queries with aggregates - part 2

Jak WHEN jest przed Group by , to jak filtrujemy elementy grupy ??? → `HAVING` !

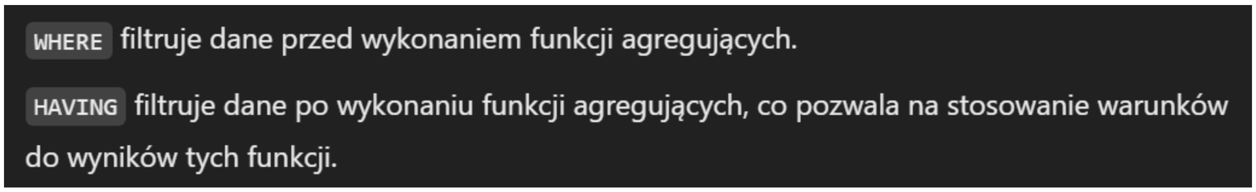
Składnia:



Select query with HAVING constraint

```
SELECT group_by_column, AGG_FUNC(column_expression) AS aggregate_result_alias, …
FROM mytable
WHERE condition
GROUP BY column
HAVING group_condition;
```

Można stosować having bez group by, ale musi być funkcja agregująca !



`WHERE` filtruje dane przed wykonaniem funkcji agregujących.

`HAVING` filtruje dane po wykonaniu funkcji agregujących, co pozwala na stosowanie warunków do wyników tych funkcji.

1. Find the number of Artists in the studio (without a **HAVING** clause)
2. Find the number of Employees of each role in the studio
3. Find the total number of years employed by all Engineers

```
SELECT SUM(ROLE LIKE 'artist') FROM employees;
// lub zcase sensitive
SELECT SUM(ROLE in ('Artist')) FROM employees;
```

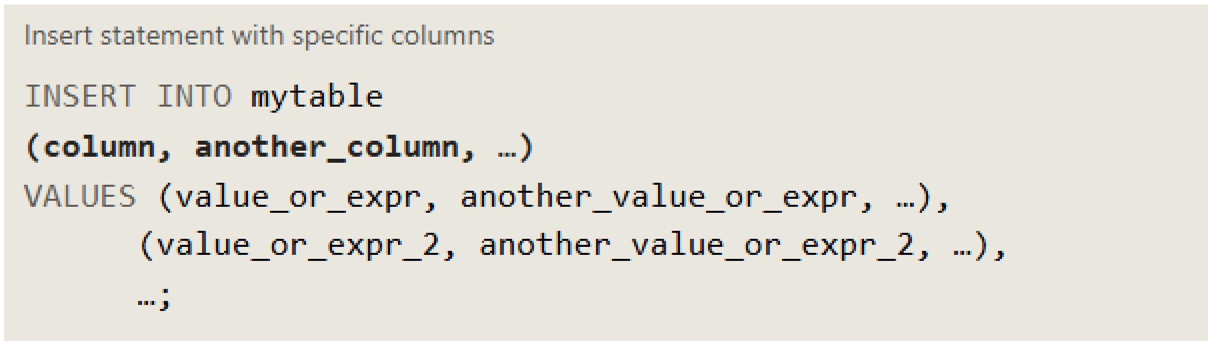
1. Find the number of movies each director has directed ✓
2. Find the total domestic and international sales that can be attributed to each director

```
SELECT director, count(director)  FROM movies
group by director;
```

```
SELECT director, sum(domestic_sales + international_sales) FROM
INNER JOIN boxoffice
    ON id=boxoffice.movie_id
group by director;
```

Inserting syntax:

1. Add the studio's new production, **Toy Story 4** to the list of movies (you can use any director) ✓
2. Toy Story 4 has been released to critical acclaim! It had a rating of **8.7**, and made **340 million domestically** and **270 million internationally**. Add the record to the `BoxOffice` table. ✓


Insert statement with specific columns

```
INSERT INTO mytable
(column, another_column, …)
VALUES (value_or_expr, another_value_or_expr, …),
       (value_or_expr_2, another_value_or_expr_2, …),
       …;
```

```
insert into movies (title,director,year,length_minutes) values
("Toy Story 4","John Lasseter",1995,64);

insert into boxoffice  values
(15,8.7,340000000,270000000);
```

1. The director for A Bug's Life is incorrect, it was actually directed by **John Lasseter** ✓
2. The year that Toy Story 2 was released is incorrect, it was actually released in **1999**
3. Both the title and director for Toy Story 8 is incorrect! The title should be "Toy Story 3" and it was directed by **Lee Unkrich**

```
UPDATE movies
SET director = "John Lasseter"
WHERE id=2
```

```
UPDATE movies
SET year = 1999
WHERE id= 3;

UPDATE movies
SET title="Toy Story 3", director="Lee Unkrich"
WHERE id = 11;
```

1. This database is getting too big, lets remove all movies that were released **before** 2005.
2. Andrew Stanton has also left the studio, so please remove all movies directed by him.

```
DELETE FROM movies
WHERE year<2005;

DELETE FROM MOVIES
WHERE director LIKE 'Andrew Stanton';
```

1. Create a new table named `Database` with the following columns:This table has no constraints. ✓

   – `Name` A string (text) describing the name of the database

   – `Version` A number (floating point) of the latest version of this database

   – `Download_count` An integer count of the number of times this database was downloaded

```
CREATE TABLE database(
name TEXT,
version FLOAT,
download_count INTEGER
)
```

1. Add a column named **Aspect_ratio** with a **FLOAT** data type to store the aspect-ratio each movie was released in.
2. Add another column named **Language** with a **TEXT** data type to store the language that the movie was released in. Ensure that the default for this language is **English**.

```
ALTER TABLE movies
ADD aspect_ratio FLOAT
DEFAULT 0;

ALTER TABLE movies
ADD language TEXT
DEFAULT english;
```

```
DROP TABLE MOVIES ; //POWINN BYĆ Z IF EXIST
DROP TABLE BOXOFFICE;
```

SQL For Beginners Tutorial | Learn SQL in 4.2 Hours | 2021

1. On conflict do nothing
2. on conflict do update

Mają rekord w bazie danych, możemy dodać rekord, ale z dopiskiem on clonflict. W przypadku wystąpienia konfliktu ona podanej kolumnie rekordu, zostanie wykonana akcja.

np. id = 10, name = Rose, age=20 - istnieje w DB

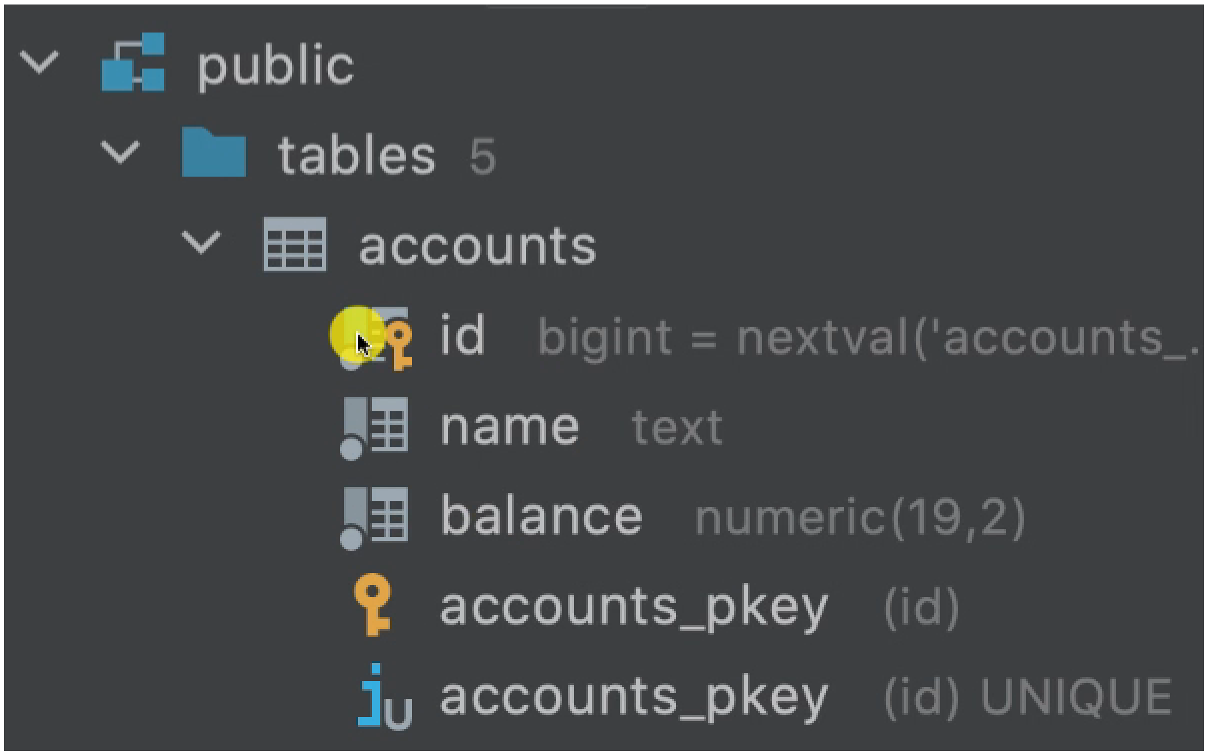
```
INSERT INTO PERSON (ID,NAME,SURNAME) VALUES(10,'MICHAL',25) ON (
// W PRZYPADKU WYSTĄPIENIA KONFLIKTU NA POLU ID, ZMIANY NIE ZOST
// ZAMIAST ID, MOŻĘ BYĆ INNE POLE KTÓRE MA UNIQE CONSTRAINT
INSERT INTO PERSON (ID,NAME,SURNAME) VALUES(10,'MICHAL',25) ON (
DO UPDATE SET name = EXCLUTED.name
To powoduje, że pole 'name' zostanie zastąpione mimo konfliktu
```

Indeksy

W przypadku jeżeli w bazie danych często wyszukujemy encji na podstawie innej kolumny niż id (np. dla użytkonika może to być unikalny adres e-mail)

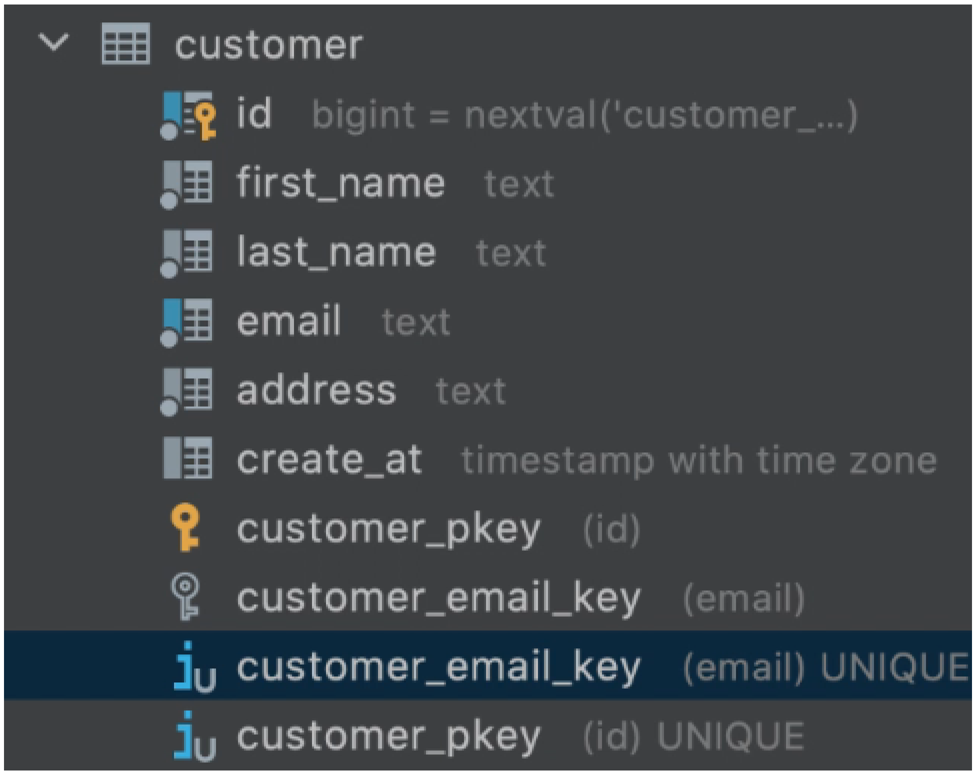
W DataGrip widać kolumny, oznaczenie co jest primary_key oraz indeksy (oznaczone literką i)

To przyśpiesza wyszukiwanie w bazie danych

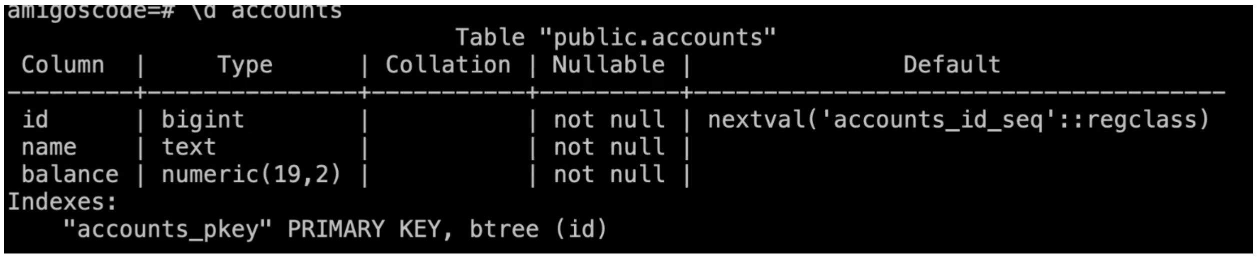


```
public
  tables 5
    accounts
      id bigint = nextval('accounts_.
      name text
      balance numeric(19,2)
      accounts_pkey (id)
      accounts_pkey (id) UNIQUE
```

Dodanie constrainta UNIQE, powoduje dodanie indeksu na kolumne


```
customer
    id  bigint = nextval('customer_...)
    first_name  text
    last_name  text
    email  text
    address  text
    create_at  timestamp with time zone
    customer_pkey  (id)
    customer_email_key  (email)
    customer_email_key  (email) UNIQUE
    customer_pkey  (id) UNIQUE
```

Widok iindeksów z terminalu komendą: \d tableName

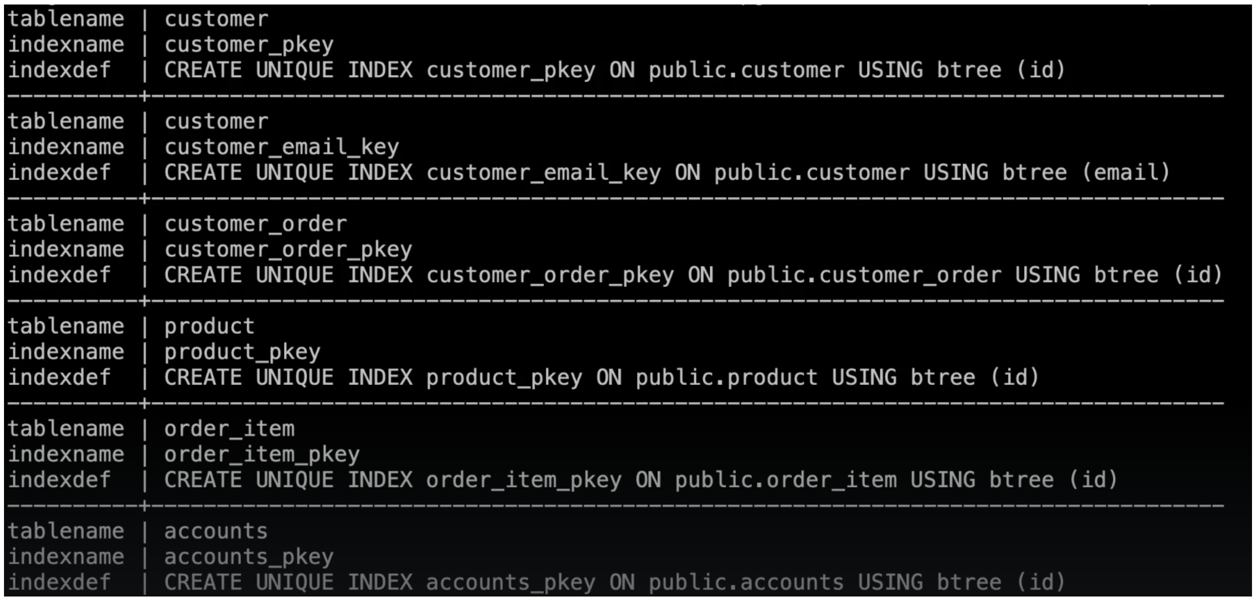

```
amigoscode=# \d accounts
                              Table "public.accounts"
 Column  |     Type      | Collation | Nullable |               Default
---------+---------------+-----------+----------+--------------------------------------
 id      | bigint        |           | not null | nextval('accounts_id_seq'::regclass)
 name    | text          |           | not null |
 balance | numeric(19,2) |           | not null |
Indexes:
    "accounts_pkey" PRIMARY KEY, btree (id)
```

btree to jedna z implementacji indeksów.

Komenda listująca wszystkie dostępne indeksy w bazie danych:

```
SELECT tablename, indexname, indexdef FROM pg_indexes WHERE sche
```

można przed dodać \x - opcja exapnd view


```
tablename | customer
indexname | customer_pkey
indexdef  | CREATE UNIQUE INDEX customer_pkey ON public.customer USING btree (id)
----------+---------------------------------------------------------------------------
tablename | customer
indexname | customer_email_key
indexdef  | CREATE UNIQUE INDEX customer_email_key ON public.customer USING btree (email)
----------+---------------------------------------------------------------------------
tablename | customer_order
indexname | customer_order_pkey
indexdef  | CREATE UNIQUE INDEX customer_order_pkey ON public.customer_order USING btree (id)
----------+---------------------------------------------------------------------------
tablename | product
indexname | product_pkey
indexdef  | CREATE UNIQUE INDEX product_pkey ON public.product USING btree (id)
----------+---------------------------------------------------------------------------
tablename | order_item
indexname | order_item_pkey
indexdef  | CREATE UNIQUE INDEX order_item_pkey ON public.order_item USING btree (id)
----------+---------------------------------------------------------------------------
tablename | accounts
indexname | accounts_pkey
indexdef  | CREATE UNIQUE INDEX accounts_pkey ON public.accounts USING btree (id)
```

Tworzenie indeksów

```
    CREATE INDEX tableName_indexingColumn_idx ON tableName(inde
    ex.
    CREATE INDEX customer_last_name_idx ON customer(last_name);
```

Typem domyślnym indeksu jest b_treee

Usunięcie indeksu:

```
DROP INDEX indexName;
```

Dla 1000

example.sql

Rozróżniamy single-column index, oraz multi-column indexes

Single było poprzenio, teraz natomiast możemy ustawić multi-column → SCHEMAT:

```
CREATE INDEX firstColumnName_secondColumnName_idx
ON table (firstColumnName,secondColumnName);

Specyfika:
.... WHERE a='' AND b='' -> POPRAWNIE
.... WHERE a='' -> POPRAWNIE
.... WHERE b='' -> ŹLE, Tylko porównujemy oba, albo tylko pierws
```

Jeżeli kolumna ma constraint UNIQE, to bez sciągnięcie constraintu, nie idze usunąć indeksu

Update można dropować, jeżeli został stowrzony poprzez create unique index

Parial indexes

Polega to na tym, że tworzymy indeksy tylko określonych wartości w kolumnie (określony subset)

Osiągamy to poprzez

```
CREATE INDEX tableName_columnName_partial_idx
ON tableName(columnName)
WHERE conditions
```

Funkcje Postgres

Szkielet funkcji:

```
CREATE OR REPLACE FUNCTION functionName(param1 type,param2 type,
RETURNS RETURNING_TYPE
LANGUAGE SELECTED_LANGUAGE
AS
    $$
    DECLARE variableName variableType

    BEGIN
    LOGIC...
    END;
    $$
```

Przykładowa funkcja:

```

//Co powinna robić funkcja -> Zwraca ilośc wystąpień podanego in
SELECT COUNT(*) FROM people WHERE first_name like 'Michal';
// Funkcja
CREATE OR REPLACE FUNCTION count_by_fiest_name(p_first_name TEXT
    RETURNS INT
    LANGUAGE plpgsql
AS
$$
DECLARE
        totalNamesCount int;
BEGIN
    SELECT COUNT(*)
    INTO totalNamesCount
    FROM people
    WHERE first_name
    LIKE p_first_name ;

    RETURN totalNamesCount;
END;
$$

// Użycie funkcji
SELECT count_by_fiest_name('Michal');
```

Sprawdzenie dostępnych funkcji → konsola \df

Administracja bazami danych - Role

W przypadku administrowania mamy następujące role:

https://www.postgresql.org/docs/current/sql-createrole.html

```
REATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | SYSID uid
```

Można tworzyć grupy ról, w których

Żeby sprawdzić dostępnych użytkowników należ w cmd podać \du - listowanie użytkoników z rolami

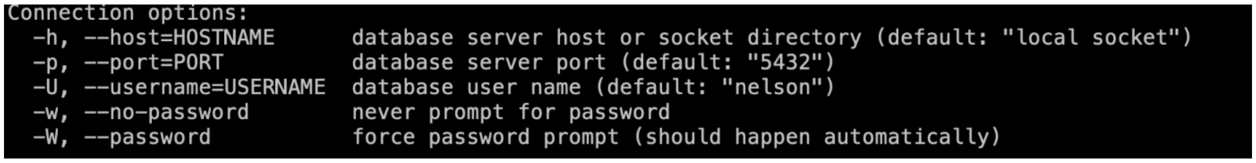
Wpisujac zapytanie:

```
SELECT rolname FROM pg_roles;
```

Uzyskamy wszystkie role, wraz z systemowymi

Zmiana użytkownika np. na użytkownika postgres

```
psql -U postgres
```



```
Connection options:
  -h, --host=HOSTNAME      database server host or socket directory (default: "local socket")
  -p, --port=PORT          database server port (default: "5432")
  -U, --username=USERNAME  database user name (default: "nelson")
  -w, --no-password        never prompt for password
  -W, --password           force password prompt (should happen automatically)
```

Wyjście \q (od quitt)

Tworzenie roli

```
CREATE ROLE michal; <- empty role (nie możne nawet się zalogowa
CREATE ROLE michal WITH LOGIN PASSWORD 'passwordToUserLogin';
CREATE USER michal; -> działa tak samo jak wyżej (bez password)
CREATE ROLE michal SUPERUSER LOGIN PASSWORD 'password'
etc....
```

Uprawnienia (Privileges) → Grant

Uprawnienia definiują kto, co, gdzie jakie operacje może wykonywać.

https://www.postgresql.org/docs/current/sql-grant.html

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERE
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]
```

```
GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name [ ( [
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA s
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { { SET | ALTER SYSTEM } [, ... ] | ALL [ PRIVILEGES ] }
    ON PARAMETER configuration_parameter [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_specification ]

GRANT role_name [, ...] TO role_specification [, ...]
    [ WITH { ADMIN | INHERIT | SET } { OPTION | TRUE | FALSE } 
    [ GRANTED BY role_specification ]

where role_specification can be:

    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Przykład:

1. Utworzyliśmy użytkownika o nazwie michal

```
//Jako użytkownik który może tworzyć role (create role)
GRANT SELECT ON grantingTableName TO grantingUserName;
//ex.2
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO michal;
```

2. Revoking (Usuwanie dostępów)

```
REVOKE functions ON range IN SCHEMA PUBLIC FROM michal;
ex.
REVOKE ALL ON ALL TABLES IN SCHEMA PUBLIC FROM maria;
```

3. Memeber Roles

```
CREATE ROLE engineers;
//Dodawanie użytkoników
GRANT engineers TO michal; -> michal member of engineers
//Revoking
REVOKE engineers FROM michal;
```

Schemas (namespaces for database objects)

Schematy pozwalają na grupowanie obiektów DB w grupy

Postgres defaultowo tworzy schemat: **public.**

Żeby sprawdzić wszystkie dostępne tabele w schematcie public wykonujemy SELECT →

```
SELECT * FROM public.
```

schemat public jest ustawiony jako defaultowy searchpath (nie tzreba wpisywać)

Można sprawdzić aktualny schemat poprzez:

```
SELECT current_schema();
```

Dzięki schematom możemy mieć kilka tak samo nazywających się tabel, ale w różnych schematach np.

```
SELECT * FROM sales.product;
SELECT * FROM reporting.product;
```

Tworzenie schematu

```
CREATE SCHEMA schemaName;
ex.
CREATE SCHEMA sales;
CREATE SCHEMA engineering;
CREATE SCHEMA marketing;
```

\dn → wyświetelenie aktualnych schematów

\d → wyświetlenie relacji w schema public

Tworzenie nowych tabel w schemacie

```
CREATE TABLE sales.product();
```

Backup database + restroing

Komenda pg_dump —help - Tu są wysztkie komendy

Speeding up Postgres

https://devcenter.heroku.com/articles/postgresql-indexes

https://database.guide/prepared-statements-in-postgresql-a-complete-guide/

https://www.youtube.com/watch?v=YON9PliOYFk

1. **Prepared statements**

   Raz tworzymy funkcje, która przyjmuje wartości argumentów - Przykład:

   ```
   PREPARE insert_task (VARCHAR(32),DATE,BOOLEAN) AS
   INSERT INTO task (name, date, isDone)
   VALUES ($1, $2, $3);

   //Wywyołanie
   EXECUTE insert_task ('make diner','2024-11-11',false);
   ```

   Dlaczego jest optymalne ?

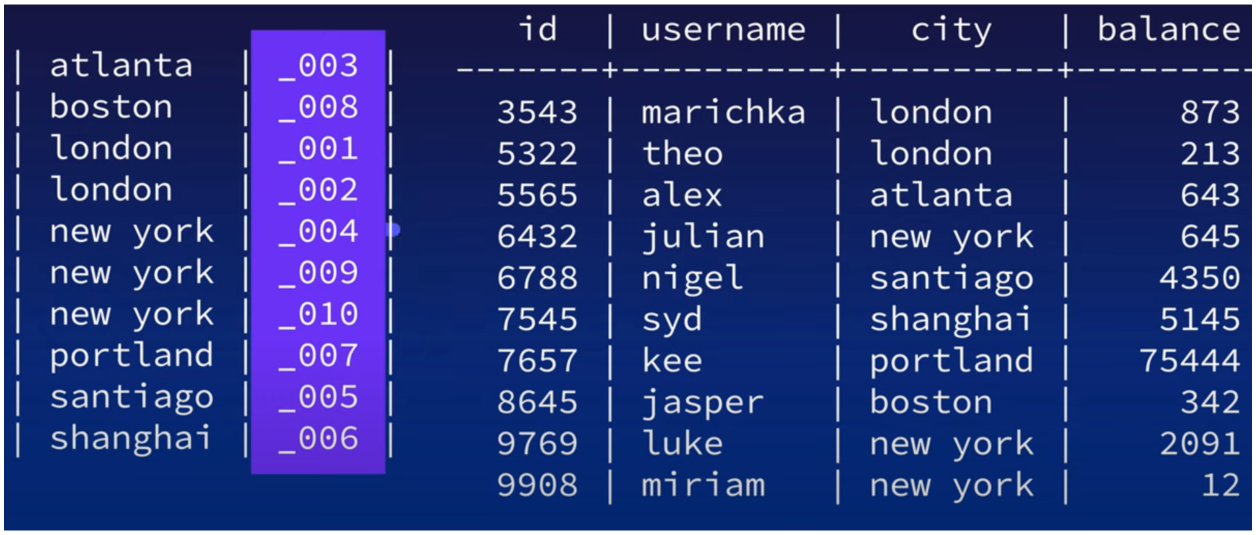
   1. Jest raz kompilowane i aprsowane
   2. Zabezpieczenie przed SQL injection
   3. Możliwość wykonywania burowania na poziomie DB

   Obsługiwane zapytania:

   - `SELECT`
   - `INSERT`
   - `UPDATE`
   - `DELETE`
   - `MERGE`
   - `VALUES`

Indexing

Defaulowo postgres wykonuje sequential scan (seq scan) .
 Indeks tworzy tabelę, która zawiera wartość indeksowaną jako key, oraz referencje do wiersza w którym dana wartość się znajduje :



```
                                 id  | username |    city    | balance
| atlanta   | _003 |          -------+----------+------------+---------
| boston    | _008 |           3543 | marichka | london     |     873
| london    | _001 |           5322 | theo     | london     |     213
| london    | _002 |           5565 | alex     | atlanta    |     643
| new york  | _004 |           6432 | julian   | new york   |     645
| new york  | _009 |           6788 | nigel    | santiago   |    4350
| new york  | _010 |           7545 | syd      | shanghai   |    5145
| portland  | _007 |           7657 | kee      | portland   |   75444
| santiago  | _005 |           8645 | jasper   | boston     |     342
| shanghai  | _006 |           9769 | luke     | new york   |    2091
                                9908 | miriam   | new york   |      12
```

Komenda: EXPLAIN ANALYZE+ QUERY, wyświetla analizę naszego SQL query

Zalecenia:

1. Twórz indeksty tylko tam, gdzie to potrzebne

PARTITIONING, czyli dzielenie na partycje

Możemy partycjonować, bazując na dowlonej kolumnie np. na timestamp

Jeżeli tworzymy tabelę:

EX.

```
CREATE TABLE events(
    event_id UUID,
    name VARCHAR(32),
    event_timestamp
)
PARTITION BY RANGE (event_timestamp);
```

Następnie możemy partycjonować manualnie:

```
CREATE TABLE events_2024_01
PARTITION OF events
FOR VALUES
FROM ('2024-01-01')
TO ('2024-01-31')
```

Zazwyczaj sie to robi automatycznie przy pomocy jakiś pipeline np. z użyciem cron (pg_cron)

Separation of concerns → read replicas

Koncepcja polega na posiadaniu 1 DB która jest READ-WRITE, oraz wielu replikach read-only. Skanowanie horyzontalne/