Apache AGE

Kamil Bernacik, Artur Stefańczyk

1. Wybór technologii

W projekcie zdecydowaliśmy się na wykorzystanie Apache AGE (A Graph Extension) jako głównej technologii do zarządzania danymi grafowymi. Apache AGE to rozszerzenie dla bazy danych PostgreSQL, które umożliwia przechowywanie i zapytania na danych w formie grafu. Dzięki integracji z PostgreSQL, zyskaliśmy dostęp do wydajnych narzędzi SQL, jednocześnie poszerzając możliwości o grafową reprezentację danych.

2. Architektura: komponenty i interakcje, schemat

2.1. Komponenty systemu

Głównymi komponentami systemu są:

- Skrypt w Pythonie do importu: odpowiada za przetwarzanie danych wejściowych, ich czyszczenie, przekształcanie oraz eksport do plików kompatybilnych z Apache AGE i PostgreSQL

- Skrypt w Pythonie do obsługi programu dbctl: odpowiada połączenie z kontenerem z bazą danych wysyłanie zapytań i odbieranie rezultatów

- Baza danych PostgreSQL z Apache AGE: PostgreSQL z rozszerzeniem Apache AGE służy jako baza danych grafowa, która przechowuje dane o relacjach między węzłami i umożliwia wydajne wykonywanie zapytań grafowych

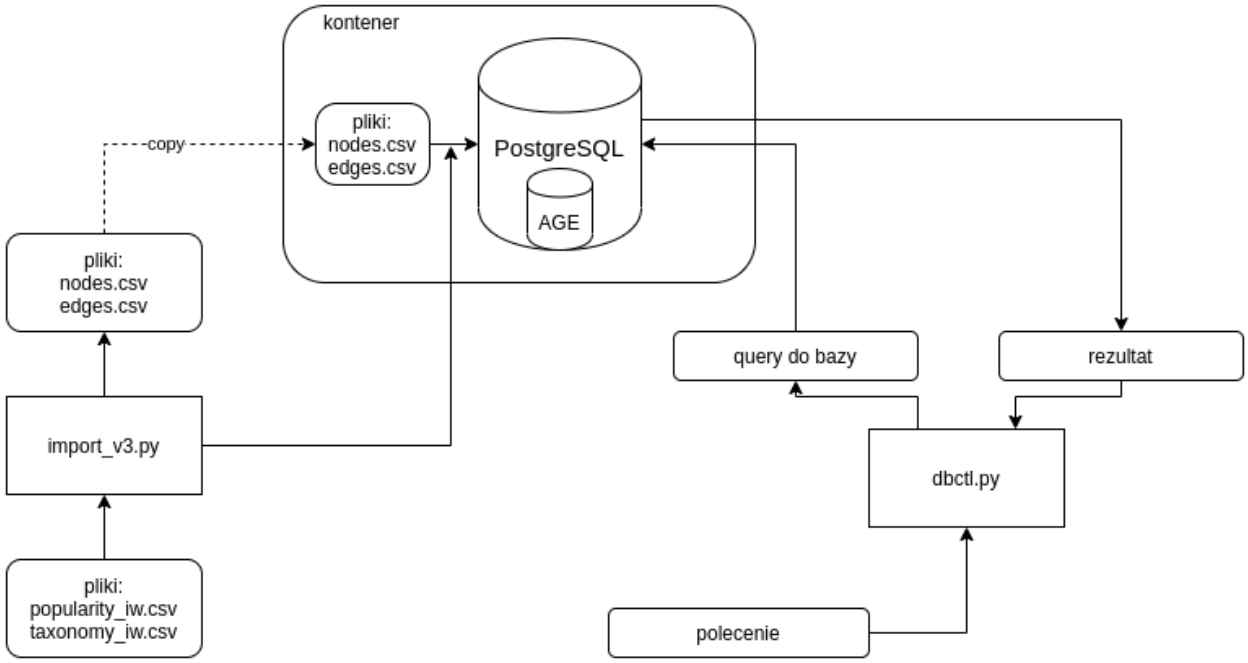
- Kontener Docker: kontener zawierający PostgreSQL z zainstalowanym Apache AGE, do którego dane są importowane oraz w którym przeprowadzane są obliczenia grafowe

2.2 Interakcje między komponentami

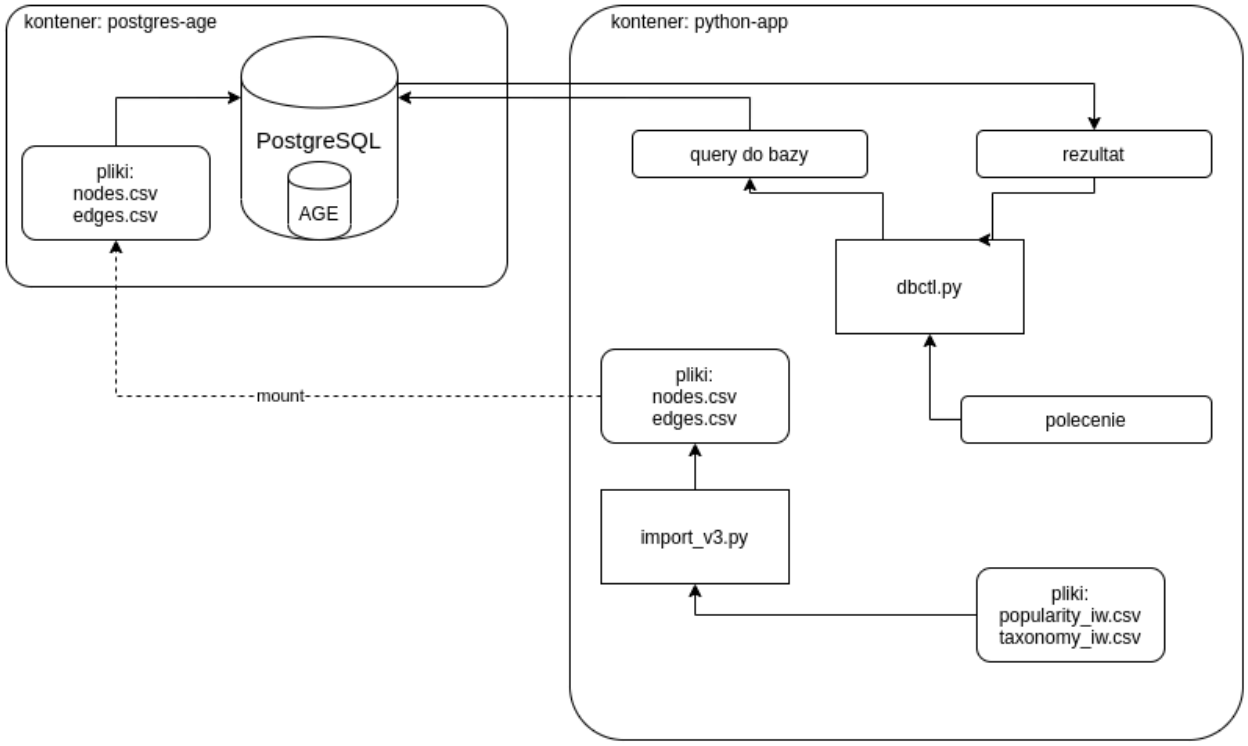
Skrypt w Pythonie przetwarza dane z plików wejściowych (**popularity_iw.csv.gz** oraz **taxonomy_iw.csv.gz**), a następnie zapisuje przetworzone dane w postaci dwóch plików CSV: nodes.csv i edges.csv. Pliki CSV są kopiowane do kontenera Dockera za pomocą poleceń **docker cp**, gdzie są następnie używane do załadowania wierzchołków i krawędzi do grafu w PostgreSQL. Zapytania grafowe są wykonywane w bazie danych PostgreSQL za pomocą rozszerzenia Apache AGE, które wykorzystuje składnię **Cypher**.

2.3. Schemat architektury

Przed docker-compose:



Po docker-compose:



2.4. Technologie użyte w architekturze

Python: do przetwarzania i przygotowania danych wejściowych, z bibliotekami:
- pandas do operacji na danych,
- gzip do obsługi skompresowanych plików,
- psycopg2 do łączenia się z PostgreSQL

Docker: do izolacji środowiska uruchomieniowego PostgreSQL z Apache AGE

PostgreSQL z Apache AGE: do przechowywania i analizy danych w postaci grafu

3. Wymagania i zależności

System powinien mieć:
- dostęp do internetu
- zainstalowanego dockera
- zainstalowanego pythona (jeśli nie korzystamy z docker-compose)
- pliki **popularity_iw.csv.gz**, **taxonomy_iw.csv.gz**

5. Instrukcja obsługi narzędzia oraz instalacji i konfiguracji

5.1 Przed docker-compose:

5.1.1 Pobranie obrazu dockera

Na systemie powinien być zainstalowany docker:

`docker pull apache/age`

`docker run --name age-container -e POSTGRES_PASSWORD=$haslo -p 5432:5432 -d apache/age`

w **$haslo** należy wpisać hasło do bazy psql, ale zalecane jest `root` bo takie jest stosowane w skryptach

5.1.2 Pobranie bibliotek

Na systemie powinien być zainstalowany python, zalecamy również stworzenie wirtualnego środowiska:

`python3 -m venv venv`

`source venv/bin/activate`

Pobranie bibliotek:

`pip install -r requirements.txt`

5.1.3 Import danych

Należy w głównym folderze mieć dwa pliki: **popularity_iw.csv.gz**, **taxonomy_iw.csv.gz**

Uruchomić skrypt:

`python import_v3.py`

5.1.4 Narzędzie

Należy uruchomić:

`python dbctl.py $number_zadania $arg1 $arg2`

5.2 Po docker-compose:

5.2.1 Uruchomienie docker-compose

Na systemie powinien być zainstalowany docker:
Należy w głównym folderze mieć dwa pliki: **popularity_iw/csv.gz**, **taxonomy_iw/csv.gz**

Uruchomić:

`docker compose build`

`docker compose up -d`

– **uwaga** port 5432 powinien być wolny (uruchomienie programu z podrozdziału 5.1 powoduje że port 5432 jest zajęty, należy zatrzymać wtedy kontener: `docker stop age-container`)

5.2.2 Wejście do kontenera z programem dbctl

Uruchomić:

`docker exec -it python_app /bin/bash`

5.2.3 Import danych

Uruchomić:

`python import_v3.py`

5.2.4 Narzędzie

Uruchomić:

`python dbctl.py $number_zadania $arg1 $arg2`

6. Proces projektowania i wdrażania krok po kroku

Rozdział ten opisuje szczegółowo wdrożenie funkcji obsługujących 18 operacji na grafie z wykorzystaniem Apache AGE, rozszerzenia dla PostgreSQL.

Task 1: Znajduje wszystkie dzieci danego węzła

Funkcja `task_1(node_name)` wyszukuje wszystkie dzieci węzła o nazwie `node_name`. W zapytaniu Cypher identyfikowany jest węzeł na podstawie nazwy, a relacje wychodzące są używane do znalezienia jego dzieci. Wynikiem zapytania jest lista nazw dzieci.

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (n {name: 'node_name'})-[e]->(child)
    RETURN child.name
$$) AS result(n agtype);
```

Task 2: Zlicza wszystkie dzieci danego węzła

Funkcja `task_2(node_name)` oblicza liczbę dzieci węzła `node_name`. Podobnie jak w Task 1, zapytanie Cypher analizuje relacje wychodzące, ale zamiast zwracać listę dzieci, zlicza ich liczbę.

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (n {name: 'node_name'})-[e]->(child)
    RETURN COUNT(child) AS child_count
$$) AS result(child_count agtype);
```

Task 3: Znajduje wszystkie wnuki danego węzła

Funkcja `task_3(node_name)` identyfikuje wszystkie wnuki węzła `node_name`. Zapytanie Cypher korzysta z relacji dwustopniowych (od węzła przez dziecko do wnuka) w celu zwrócenia nazw wnuków.

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (n {name: 'node_name'})-[e]->(child)-[e2]->(grandchild)
    RETURN grandchild.name
$$) AS result(name agtype);
```

Task 4: Znajduje wszystkich rodziców danego węzła

Funkcja `task_4(node_name)` wyszukuje wszystkie węzły, które są rodzicami węzła `node_name`. W zapytaniu Cypher analizowane są relacje skierowane do danego węzła, a następnie zwracane są nazwy rodziców.

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (parent)-[e]->(n {name: 'node_name'})
    RETURN parent.name
$$) AS result(name agtype);
```

Task 5: Zlicza wszystkich rodziców danego węzła

Funkcja `task_5(node_name)` zlicza wszystkich rodziców węzła `node_name`. Jest to analogiczne do Task 4, ale zamiast zwracać listę nazw rodziców, zapytanie Cypher zwraca ich liczbę.

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (parent)-[e]->(n {name: 'node_name'})
    RETURN COUNT(parent) AS parent_count
$$) AS result(parent_count int);
```

Task 6: Znajduje wszystkich dziadków danego węzła

Funkcja `task_6(node_name)` wyszukuje wszystkich dziadków węzła `node_name`. W zapytaniu Cypher analizowane są relacje dwustopniowe, tym razem skierowane do danego węzła (od dziadka przez rodzica).

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (grandparent)-[e]->(parent)-[e2]->(n {name: 'node_name'})
```

```
    RETURN grandparent.name
$$) AS result(name agtype);
```

Task 7: Liczy, ile jest węzłów o unikatowych nazwach

Funkcja `task_7()` zlicza wszystkie węzły w grafie, które mają unikatowe nazwy. W zapytaniu Cypher pobierane są wszystkie nazwy węzłów, a następnie zliczane są tylko te, które są unikalne.

Przykład zapytania:

```
SELECT COUNT(*) AS unique_node_count
FROM cypher('iw_graph', $$
    MATCH (n)
    RETURN DISTINCT n.name
$$) AS result(name agtype);
```

Task 8: Znajduje węzły, które nie są podkategorią żadnego innego węzła

Funkcja `task_8()` identyfikuje węzły w grafie, które nie mają żadnych relacji skierowanych do nich od innych węzłów. Zapytanie Cypher przeszukuje wszystkie węzły, a następnie stosuje filtrację w celu znalezienia tych, które mają zerową liczbę krawędzi przychodzących. Jest to pierwsze zapytanie, przy którym pojawiły się problemy związane z wydajnością.

Przykład zapytania:

```
            SELECT no_inbound_id
            FROM cypher('iw_graph', $$
                  MATCH (n)
                  WITH n
                  ORDER BY id(n)
                  SKIP {current_offset}
                  LIMIT {PAGE_SIZE}
                  WITH collect(n) AS page_nodes
                  UNWIND page_nodes AS candidate
                  OPTIONAL MATCH (m)-[r]->(candidate)
                  WITH candidate, COUNT(m) AS inbound_count
                  WHERE inbound_count = 0
                  RETURN id(candidate) AS no_inbound_id
```

Funkcja przetwarza wyniki zapytania w partiach (ponieważ graf jest duży), aby zapewnić wydajność, a następnie zwraca listę węzłów z ich nazwami, które nie są podkategorią żadnego innego węzła.

Task 9: Zlicza węzły, które nie są podkategorią żadnego innego węzła

Funkcja `task_9()` liczy wszystkie węzły, które nie mają żadnych relacji skierowanych do nich. Jest to rozszerzenie funkcji `task_8()`, gdzie zamiast zwracania listy węzłów, zapytanie Cypher zlicza ich liczbę. Podobnie jak w przypadku zadania 8, pojawiły się tu problemy z wydajnością. Rozwiązaniem tego problemu było użycie paginacji.

Przykład zapytania:

```
            SELECT no_inbound_id
            FROM cypher('iw_graph', $$
                  MATCH (n)
                  WITH n
                  ORDER BY id(n)
                  SKIP {current_offset}
                  LIMIT {PAGE_SIZE}
                  WITH collect(n) AS page_nodes
                  UNWIND page_nodes AS candidate
                  OPTIONAL MATCH (m)-[r]->(candidate)
                  WITH candidate, COUNT(m) AS inbound_count
                  WHERE inbound_count = 0
                  RETURN id(candidate) AS no_inbound_id
```

Task 10: Znajduje węzły z największą liczbą dzieci

Funkcja `task_10()` identyfikuje węzły w grafie, które mają największą liczbę dzieci. Zapytanie Cypher oblicza liczbę dzieci dla każdego węzła, sortuje wyniki malejąco według liczby dzieci, a następnie zwraca węzły o najwyższej liczbie dzieci. Jeśli istnieje więcej niż jeden węzeł z tą samą maksymalną liczbą dzieci, wszystkie takie węzły są zwracane.

Przykład zapytania:

```
WITH aggregated_data AS (
    SELECT start_id, COUNT(end_id) AS num_childs
    FROM iw_graph.has
    GROUP BY start_id
    ORDER BY num_childs DESC
    LIMIT 1
)
```

```
SELECT c.properties, a.num_childs
FROM aggregated_data a
LEFT JOIN iw_graph."Category" c ON a.start_id = c.id;
```

Task 11: Znajduje węzły z najmniejszą liczbą dzieci (więcej niż zero)

Funkcja `task_11()` wyszukuje węzły, które mają dokładnie jedno dziecko. Zapytanie Cypher przechodzi przez wszystkie węzły w grafie, analizuje liczbę relacji wychodzących dla każdego węzła, a następnie zwraca te, które mają jedno dziecko. Funkcja dodatkowo przetwarza wyniki w partiach podobnie jak w przypadku zapytań z task 8 i 9, jest to ponownie związane z wydajnością zapytania, a także podobieństwem pomiędzy tymi zadaniami.

Przykład zapytania:

```
            SELECT node_id
            FROM cypher('iw_graph', $$
                  MATCH (n:Category)
                  WITH n
                  ORDER BY id(n)
                  SKIP {current_offset}
                  LIMIT {PAGE_SIZE}
                  MATCH (n)-[:has]->(c)
                  WITH n, COUNT(c) AS num_childs
                  WHERE num_childs = 1
                  RETURN id(n) AS node_id
            $$) AS (node_id agtype);
```

Task 12: Zmienia nazwę danego węzła

Funkcja `task_12(old_name, new_name)` zmienia nazwę węzła w grafie. Zapytanie Cypher wyszukuje węzeł o podanej starej nazwie (`old_name`) i aktualizuje jego nazwę na nową wartość (`new_name`).

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (n {name: 'old_name'})
    SET n.name = 'new_name'
    RETURN n
$$) AS result(n agtype);
```

Task 13: Zmienia popularność danego węzła

Funkcja `task_13(node_name, new_popularity)` aktualizuje wartość popularności węzła w grafie. Zapytanie Cypher wyszukuje węzeł o nazwie `node_name` i ustawia jego pole `popularity` na nową wartość `new_popularity`.

Przykład zapytania:

```
SELECT * FROM cypher('iw_graph', $$
    MATCH (n {name: 'node_name'})
    SET n.popularity = new_popularity
    RETURN n
$$) AS result(n agtype);
```

Task 14: Znajduje wszystkie ścieżki pomiędzy dwoma węzłami

Funkcja `task_14(start_name, end_name, max_path_length=10, incremental_step=1)` znajduje wszystkie ścieżki skierowane między dwoma węzłami: `start_name` i `end_name`. W zapytaniu Cypher wyszukiwane są ścieżki, których długość nie przekracza określonej wartości `max_path_length`. Funkcja iteracyjnie zwiększa długość ścieżki od 1 do `max_path_length`, zbierając wyniki. Konieczność zastosowania zmiennej `max_path_length` była podyktowana problemami z wydajnością związanymi z zapytaniami wykorzystującymi składnię -[*]-.

Przykład zapytania dla ścieżek o długości `n`:

```
SELECT p
FROM cypher('iw_graph', $$
    MATCH p = (startNode)-[*n]->(endNode)
    WHERE startNode.name = 'start_name'
      AND endNode.name = 'end_name'
    RETURN p
$$) AS (p agtype);
```

Task 15: Zlicza wszystkie ścieżki pomiędzy dwoma węzłami

Funkcja `task_15(start_name, end_name, max_path_length=10, incremental_step=1)` zlicza liczbę wszystkich skierowanych ścieżek pomiędzy dwoma węzłami: `start_name` i `end_name`. Działa podobnie do funkcji `task_14`, jednak zamiast zbierać wszystkie ścieżki, skupia się na ich zliczeniu. Ze względu na podobieństwo z zadaniem 14 pojawiły się także problemy z wydajnością.

Przykład zapytania dla ścieżek o długości `n`:

```
SELECT p
FROM cypher('iw_graph', $$
    MATCH p = (startNode)-[*n]->(endNode)
    WHERE startNode.name = 'start_name'
      AND endNode.name = 'end_name'
    RETURN p
$$) AS (p agtype);
```

Task 16: Liczy popularność w sąsiedztwie węzła o zadanym promieniu

Funkcja `task_16(node_name, r)` oblicza sumę popularności węzła `node_name` oraz wszystkich węzłów znajdujących się w jego sąsiedztwie w promieniu `r`. Zapytanie Cypher iteracyjnie przeszukuje węzły na odległość od 0 do `r` i sumuje wartość ich pola `popularity`. Ze względu na problemy z wydajnością, zapytanie do tego zadania generowane jest dynamicznie. Proste zapytanie wykorzystujące składnie -[*..{r}]-> nie zadziałało. Zapytanie jest także wrażliwe na wielkość zadanego promienia(przy większych promieniach pojaiwa się problem z wydajnością).

Task 17: Liczy popularność na najkrótszej ścieżce między dwoma węzłami

Funkcja `task_17(node_name1, node_name2)` oblicza sumę popularności węzłów znajdujących się na najkrótszej ścieżce skierowanej między węzłami `node_name1` i `node_name2`. Zapytanie Cypher wyszukuje najkrótszą ścieżkę, iteruje po węzłach na tej ścieżce i sumuje ich popularność.

Przykład zapytania:

```
WITH paths_cte AS (
    SELECT * FROM cypher('iw_graph', $$
        MATCH path = (V {name: 'node_name1'})-[*]->(V2 {name: 'node_name2'})
        UNWIND nodes(path) AS nodes_on_path
        RETURN nodes_on_path.popularity, length(path)
    $$) AS result(popularity_on_path float, path_len int)
)
SELECT popularity_on_path
FROM paths_cte
ORDER BY path_len ASC
LIMIT 1;
```

Task 18: Znajduje ścieżkę o największej popularności pomiędzy dwoma węzłami

Funkcja `task_18(node_name1, node_name2)` wyszukuje skierowaną ścieżkę między węzłami `node_name1` i `node_name2`, która ma najwyższą sumę popularności spośród wszystkich

możliwych ścieżek. Zapytanie Cypher przeszukuje graf, iteruje po wszystkich ścieżkach, oblicza sumę popularności i zwraca tę z największą wartością.

Przykład zapytania:

```
WITH paths_cte AS (
    SELECT * FROM cypher('iw_graph', $$
        MATCH path = (V {name: 'node_name1'})-[*]->(V2 {name: 'node_name2'})
        UNWIND nodes(path) AS nodes_on_path
        RETURN nodes_on_path.popularity, path
    $$) AS result(popularity_on_path float, path agtype)
)
SELECT path
FROM paths_cte
ORDER BY popularity_on_path DESC
LIMIT 1;
```

7. Role wszystkich osób w projekcie i opis tego, kto co zrobił

Kamil Bernacik:

- Importer
- task 1
- task 2
- task 3
- task 4
- task 5
- task 6
- task 7
- task 10
- task 12
- task 13
- task 16
- task 17
- task 18

Artur Stefańczyk:

- task 8
- task 9
- task 11
- task 14
- task 15
- docker-compose

8. Wyniki

W poniższym rozdziale przedstawiono uzyskane wyniki dla przykładowych zapytań:

Task 1:

docker exec -it python_app python dbcli.py 1 1889
output:

```
SELECT * FROM cypher('iw_graph', $$
MATCH (n {name: '1889'})-[e]->(child)
RETURN child.name
$$) AS result(n agtype);
```

[('"1889_works"',), ('"1889_crimes"',), ('"1889_in_law"',) …

Task 2:

docker exec -it python_app python dbcli.py 2 1889
output:

```
SELECT * FROM cypher('iw_graph', $$
MATCH (n {name: '1889'})-[e]->(child)
RETURN COUNT(child) AS child_count
$$) AS result(child_count agtype);
```

[('30',)]

Task 3:

docker exec -it python_app python dbcli.py 3 1889
output:

```
SELECT * FROM cypher('iw_graph', $$
MATCH (n {name: '1889'})-[e]->(child)-[e2]->(grandchild)
RETURN grandchild.name
$$) AS result(name agtype);
```

[('"1889_musicals"',), ('"1889_plays"',), ('"1889_books"',) …

Task 4:

docker exec -it python_app python dbcli.py 4 1889
output:

```
SELECT * FROM cypher('iw_graph', $$
MATCH (parent)-[e]->(n {name: '1889'})
RETURN parent.name
$$) AS result(name agtype);
```

[('"1880s"',), ('"Years"',)]

Task 5:

docker exec -it python_app python dbcli.py 5 1889
output:

```
SELECT * FROM cypher('iw_graph', $$
MATCH (parent)-[e]->(n {name: '1889'})
RETURN COUNT(parent) AS parent_count
$$) AS result(parent_count int);
```

[(2,)]

Task 6:

docker exec -it python_app python dbcli.py 6 1889
output:

```
SELECT * FROM cypher('iw_graph', $$
MATCH (grandparent)-[e]->(parent)-[e2]->(n {name: '1889'})
RETURN grandparent.name
$$) AS result(name agtype);
```

[('"Decades"',), ('"19th_century"',), ('"Chronology"',), ('"Units_of_time"',), ('"Calendars"',)]

Task 7:

docker exec -it python_app python dbcli.py 7 1889
output:
➜ PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 7 1889

```
SELECT COUNT(*) AS unique_node_count
FROM cypher('iw_graph', $$
MATCH (n)
```

```
        RETURN DISTINCT n.name
        $$) AS result(name agtype);
```

[(2031337,)]

Task 8:

docker exec -it python_app python dbcli.py 8
output:
➜ PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 8

- Node ID=844424932141477, name=Wikipedia_editnotices
- Node ID=844424932141893, name=Wikipedia_style_templates
- Node ID=844424932143007, name=Copyright_examinations
- Node ID=844424932148245, name=Peter_Cooper
- Node ID=844424932148475, name=Geometry_articles_needing_expert_attention
- Node ID=844424932153544, name=Wikipedia_categories_named_after_Czechoslovak_politicians
- Node ID=844424932154717, name=Wallis_Simpson
…

Task 9:

docker exec -it python_app python dbcli.py 9
output:
➜ PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 9

Total 2141 such nodes.

Task 10:

docker exec -it python_app python dbcli.py 10
output:
➜ PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 10

```
        WITH aggregated_data AS (
        SELECT start_id, COUNT(end_id) AS num_childs
        FROM iw_graph.has
        GROUP BY start_id
        ORDER BY num_childs DESC
        LIMIT 1
        )
        SELECT c.properties, a.num_childs
```

```
FROM aggregated_data a LEFT JOIN iw_graph."Category" c ON a.start_id = c.id;
```

```
[('{"id": "43354", "name": "Albums_by_artist", "__id__": 43354, "popularity": "937.0"}', 23435)]
```

Task 11:

docker exec -it python_app python dbcli.py 11
output:

```
➜  PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 11

 - Node ID=844424932163094, name=Congolais_Wikipedians
 - Node ID=844424932163095, name=Congolese_Wikipedians
 - Node ID=844424932163213,
name=NCAA_Division_I_Women-s_Basketball_Tournament_Final_Four_seasons
 - Node ID=844424932163230, name=Mill_architecture
 - Node ID=844424932163263, name=Independent_comics_images
 - Node ID=844424932163294, name=2nd-millennium_establishments_in_Eswatini
 - Node ID=844424932163298, name=2nd-millennium_disestablishments_in_Eswatini
…
```

Task 12:

output:

```
➜  PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 12 1889 1890
```

```
SELECT * FROM cypher('iw_graph', $$
MATCH (n {name: '1889'})
SET n.name = '1890'
RETURN n
$$) AS result(n agtype);
```

```
[('{"id": 844424930136184, "label": "Category", "properties": {"id": "4216", "name": "1890",
"__id__": 4216, "popularity": "21553.0"}}::vertex',)]
```

Task 13:

output:

```
➜  PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 13 1889 100
```

```
SELECT * FROM cypher('iw_graph', $$
MATCH (n {name: '1889'})
SET n.popularity = 100
RETURN n
$$) AS result(n agtype);
```

[('{"id": 844424930136319, "label": "Category", "properties": {"id": "4351", "name": "1889", "__id__": 4351, "popularity": 100}}::vertex',), ('{"id": 844424930136184, "label": "Category", "properties": {"id": "4216", "name": "1889", "__id__": 4216, "popularity": 100}}::vertex',)]

Task 14:

output:
➜ PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 14 1889 1889_in_law
Path 1: [{"id": 844424930136184, "label": "Category", "properties": {"id": "4216", "name": "1889", "__id__": 4216, "popularity": 100}}::vertex, {"id": 1125899906917029, "label": "has", "end_id": 844424930136185, "start_id": 844424930136184, "properties": {}}::edge, {"id": 844424930136185, "label": "Category", "properties": {"id": "4217", "name": "1889_in_law", "__id__": 4217, "popularity": "29300.0"}}::vertex]::path

Task 15:

➜ PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 15 1889 1889_in_law

Total paths found from '1889' to '1889_in_law': 1

Task 16:

➜ PSQL-AGE-project git:(main) docker exec -it python_app python dbcli.py 16 1889 1

```
WITH nodes0 AS (
SELECT * FROM cypher('iw_graph', $$
        MATCH (n {name: '1889'})
        RETURN n.name, n.popularity
$$) AS result(node_name agtype, popularity float)
)
,
nodes1 AS (
SELECT * FROM cypher('iw_graph', $$
        MATCH (n {name: '1889'})-[e1]->(neighbor)
        RETURN neighbor.name, neighbor.popularity
$$) AS result(node_name agtype, popularity float)
)
,
combined_nodes AS (
SELECT * FROM nodes0
UNION ALL
```

```
    SELECT * FROM nodes1
    )
    SELECT SUM(popularity) AS total_popularity
    FROM (SELECT DISTINCT node_name, popularity FROM combined_nodes) AS
unique_nodes;
```

[(838654.0,)]

9. Instrukcja krok po kroku jak odtworzyć wyniki

Aby odtworzyć wyniki należy postawić obraz dockera stosując komendy:

`docker compose build`

`docker compose up -d`

a następnie użyć wszystkich z wymienionych powyżej komend w podanej kolejności:

- *docker exec -it python_app python dbcli.py 1 1889*
- *docker exec -it python_app python dbcli.py 2 1889*
- *docker exec -it python_app python dbcli.py 3 1889*
- *docker exec -it python_app python dbcli.py 4 1889*
- *docker exec -it python_app python dbcli.py 5 1889*
- *docker exec -it python_app python dbcli.py 6 1889*
- *docker exec -it python_app python dbcli.py 7 1889*
- *docker exec -it python_app python dbcli.py 8*
- *docker exec -it python_app python dbcli.py 9*
- *docker exec -it python_app python dbcli.py 10*
- *docker exec -it python_app python dbcli.py 11*
- docker exec -it python_app python dbcli.py 12 1889 1890
- docker exec -it python_app python dbcli.py 12 1890 1889
- docker exec -it python_app python dbcli.py 13 1889 100
- docker exec -it python_app python dbcli.py 14 1889
- docker exec -it python_app python dbcli.py 15 1889
- docker exec -it python_app python dbcli.py 16 1889 1

10. Samoocena

5.0 za walkę z AGEm.

11. Strategie przyszłego łagodzenia zidentyfikowanych niedociągnięć

Szczególnie problematyczne okazały się zadania 8, 9, 11, 14, 15, 16, 17 oraz 18, które wymagały zaawansowanych operacji na dużych zbiorach danych.

Zadania 8, 9 oraz 11 udało się rozwiązać przy pomocy paginacji, co choć wolne, pozwala na obsługę dużych grafów bez przeciążenia pamięci. Paginacja umożliwia przetwarzanie wyników w partiach, ale jest procesem czasochłonnym, zwłaszcza w przypadku grafów o milionach węzłów. Można rozważyć równoległe wykonywanie zapytań w ramach paginacji, co pozwoliłoby na efektywniejsze wykorzystanie zasobów.

Zadania 14 i 15 zostały tylko częściowo rozwiązane z uwagi na ograniczenie związane z parametrem `max_length`, co oznacza, że ścieżki dłuższe niż ten parametr nie są uwzględniane.

Zapytanie 16 zostało rozwiązane przy pomocy dynamicznego generowania zapytań, jednak metoda ta nie radzi sobie efektywnie z dużym promieniem, ponieważ liczba węzłów do przetworzenia rośnie wykładniczo wraz z promieniem.

Zadania 17 i 18 zostały zrealizowane, ale ich czas wykonania jest bardzo długi. Głównym problemem jest czasochłonność obliczania popularności na ścieżkach i przeszukiwania grafu. Problem AGE jest taki, że nie ma funkcji znajdowania najkrótszej ścieżk, przez co przeszukuje wszystkie, a następnie wszystkie takie można posortować po długości.

Dodatkowo, dla wszystkich wymienionych problemów, warto rozważyć równoległe wykonywanie zapytań oraz lepsze wykorzystanie zasobów sprzętowych, takich jak GPU, które mogą znacząco przyspieszyć przetwarzanie dużych grafów. Poprawa indeksowania węzłów i relacji oraz wprowadzenie wydajniejszych strategii przeszukiwania z pewnością zwiększyłoby efektywność systemu.

Rozwiązaniem wszystkich tych problemów jest zmiana narzędzia pracy. AGE może być dobrym narzędziem gdy przykładowo już korzystamy z PostgreSQL a potrzebujemy rozwiązania z bazą grafową. Nie musimy wtedy stawiać nowej bazy, co jest korzystne. Lecz ze względu na wydajność, problem z wstawianiem danych trzeba mieć na uwadze, że taka baza powinna mieć nie więcej niż 100 000 węzłów.