Dokumentacja Końcowa

Przestrzenne Bazy Danych

Porównanie możliwości realizacji zapytań o dane przestrzenne — Python (GeoPandas) vs PostGIS

Dawid Budzyński, 319020

Filip Budzyński, 319021

Repozytorium — https://github.com/FilipBudzynski/spdb

Opis środowiska testowego

Testy zostały przeprowadzone na komputerze:

- **MacBook Air M1** 2021 (Apple Silicon)
- **RAM**: 16 GB
- **Dysk**: 256 GB SSD
- **System**: macOS

Środowisko uruchomieniowe:

- **PostgreSQL:** 16.4
- **PostGIS:** 3.4 (uruchomiony w kontenerze Docker)
- **Python:** 3.11
- **GeoPandas:** 1.0.1
- **SQLAlchemy:** 2.0.4
- **Poetry:** do zarządzania środowiskiem i zależnościami
- **NoteBooki** `.ipynb`**:** do przeprowadzenia eksperymentów

Zbiory danych i ich opis

W eksperymentach wykorzystano dane z http://download.geofabrik.de/europe/poland/mazowieckie-latest.osm.pbf

Ze względu na problemy z załadowaniem danych do pamięci przy eksperymentach dla **GeoPandas** oraz na wydajność kontenera Docker, zdecydowano się na zmniejszenie obszaru danych do fragmentu Warszawy.

Dane zostały pobrane w formacie `.pbf` a następnie zaimportowane do bazy danych.

Dane zostały *przycięte* za pomocą narzędzia `osmosis` :

```
osmosis --read-pbf mazowieckie-latest.osm.pbf \
    --bounding-box top=52.4 left=20.9 bottom=52.2 right=21.2 \
    --write-pbf warszawa.osm.pbf
```

Opis Danych

Poniższy podpunkt opisuje dane, które zostały wykorzystane w eksperymentach oraz ich reprezentację w bazie danych.

Wykorzystane tabele:

- `planet_osm_point` — tabela opisująca indywidualne punkty z przypisanymi tagami np. szkoły, przystanki, sklepy.

  Dane z tej tabeli zostały wykorzystane przy zapytaniach o np. szkoły czy przystanki autobusowe.

Kolumny	Opis
`osm_id`	OSM node ID
`amenity` , `shop` , `tourism` , `building` , `name` , etc.	kolumny z Tagami
`way`	Typ: Geometry(Point, 3857)

  Kolumna `way` oznacza typ geometri. Dla punktów jest to `Point` z **SRID** (Spatial Reference System Identifier) 3857 powszechnie stosowany w mapach internetowych (np. Google Maps, OpenStreetMap tiles).

- `planet_osm_line` — tabela zawierająca obiekty liniowe (nie zamknięte linie) opisujące np. drogi, tory, rzeki.

  Dane z tej tabeli zostały wykorzystane przy zapytaniach o np. rzeki, drogi, tory kolejowe.

Kolumny	Opis
`osm_id`	OSM node ID
`route` , `power` , `waterway` etc.	kolumny z Tagami
`way`	Typ: Geometry(LineString, 3857)

`LineString` — linia składająca się z co najmniej dwóch punktów, może być zamknięta lub otwarta. Jest *'nie prosta'* gdy przecina samą siebie.

- `planet_osm_polygon` — to tabela, która zawiera dane przestrzenne jako poligony np. budynki, jeziora, parki.

  Dane z tej tabeli zostały wykorzystane przy zapytaniach o np. parki, budynki, granice administracyjne.

Kolumny	Opis
`osm_id`	OSM node ID
`tourism`, `leisure`, `building` etc.	kolumny z Tagami
`way`	Typ: Geometry(Geometry, 3857)

  `Geometry` — oznacza, że jest to dowolna geometria: najczęściej **Polygon** lub **MultiPolygon.** Jest reprezentacją obszaru. Zewnętrzna granica wielokąta jest reprezentowana przez pierścień. Ten pierścień jest typu `LineString`, który jest zarówno zamknięty, jak i prosty.

  `MultiPolygon` jest kolekcją poligonów.

Sposób wczytania danych

Wczytywanie danych do systemu bazodanowego

Aby móc wykonywać zapytania na danych pobranych w formacie `.pbf` wykorzystano narzędzie `osm2pgsql`.

Pobrane i przycięte dane zostały wczytane do systemu bazodanowego przy użyciu komendy:

```
osm2pgsql -d gis_db -U postgres -H localhost -P 5432 \
  --create --slim --hstore \
  --style /usr/share/osm2pgsql/default.style \
```

Flagi:

- slim — użyte ze względu na ograniczone środowisko kontenera Docker
- hstore — potrzebne ze względu na użycie **osm2pgsql**, by dane zostały poprawnie zaimportowane. Tworzy dodatkową kolumnę typu `hstore` w tabelach, która **przechowuje wszystkie tagi OSM** w formacie klucz-wartość, zapewniając spójność z pobranymi danymi z openstreetmaps.

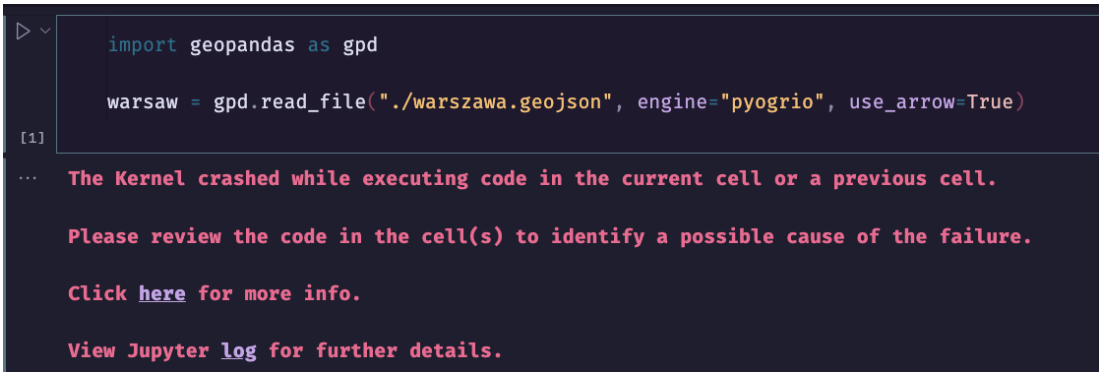
Wczytywanie danych dla biblioteki GeoPandas

GeoPandas jest biblioteką python operującą w pamięci RAM.

1. Jednym z przetestowanych sposobów wczytania danych była konwersja na format `.geojson` oraz wczytanie danych z poziomu notatnika python.

   ```
   osmium export warszawa.osm.pbf -o warszawa.geojson -f geojson
   ```

   Niestety pomimo wykorzystania dodatkowych bibliotek przyspieszających wczytywanie danych tj. **engine=pyogrio** oraz **pyarrow** nie udało się załadować danych do pamięci operacyjnej. Efektem tych prób była awaria jądra używanego dla notatnika python.



2. Próba podzielenia danych i importu. Pierwszy sposób wczytania danych okazał się zawodny, w związku z czym podzielono dane np. na parki oraz budynki i wczytano jest do pamięci operacyjnej.

   Dane podzielono w następujący sposób:

   ```
   osmium tags-filter warszawa.osm.pbf nwr/leisure=park -o parks.osm.pbf
   osmium export parks.osm.pbf -o parks.geojson -f geojson
   ```

   Zrezygnowano z powyższego sposobu wczytywania danych i realizacji zapytań gdyż podzielenie danych i operowanie na mniejszej ilości, mogło wpłynąć na rzetelność przeprowadzanych eksperymentów. Wykonując zapytania z PostGIS wykonujemy je na całym zbiorze — oczekujemy tego samego zachowania dla biblioteki GeoPandas.

3. Import danych z systemu bazodanowego do pamięci operacyjnej dla GeoPandas.

   Zastosowano podejście spełniające możliwość wykonania zapytań na całych zbiorach danych poprzez wczytanie danych za pomocą funkcji `geopandas.read_postgis`.

   Wczytano po kolei:

   - **gp_point** — tabela **planet_osm_point** w systemie bazodanowym
   - **gp_lines** — tabela **planet_osm_lines** w systemie bazodanowym
   - **gp_polygons** — table **planet_osm_polygons** w systemie bazodanowym

   Następnie wykonując zapytania, filtrowano odpowiednie wartości np. aby uzyskać dane o parkach:

   ```
   parks = gp_polygons[gp_polygons['leisure'] == 'park']
   ```

   Co jest ekwiwalentem do zapytania SQL:

   ```
   SELECT p.way as geom
   FROM planet_osm_polygon p
   WHERE p.leisure = 'park';
   ```

   Zdecydowano się na ten sposób wczytania danych ze względu na:

   - Brak awarii jądra jak w przypadku wczytywania danych w formacie `.geojson`
   - Zachowanie takiej samej ilości wykorzystanych danych jak w systemie bazodanowym, czego nie oferowało podejście drugie.

Wnioski

- **PostGIS** — Sposób wczytywania danych do systemu bazodanowego z PostGIS wymaga dodatkowych narzędzi (**osm2pgsql**) aby wczytać dane w formacie `.osm.pbf` do systemu. Niemniej proces ten przebiegł bez większych trudności i zastrzeżeń a dane są gotowe do wykonywania o nie zapytań. Z związku z powyższym, PostGIS dobrze nadaję się do wykonywania operacji na dużych zbiorach danych.
- **GeoPandas** — Napotkano wiele problemów podczas prób wczytywania danych w formacie `.geojson` . Najkrótszy czas ładowania danych wyniósł 48 minut, niestety dane nie zostały załadowane nawet po tak długim czasie ponieważ nastąpiła awaria jądra, która uniemożliwiła dalszą pracę. GeoPandas nie jest przystosowane do wczytywania bardziej obszernych danych.

Wyniki przeprowadzonych testów

Testy obejmowały porównanie możliwości wykonywania zapytań przestrzennych na podstawie

- relacji topologicznych
- relacji odległościowych
- złączeń przestrzennych
- funkcji agregujących

Wykonano również eksperyment pokazujący możliwości wykonywania zapytań przy ograniczonych zasobach pamięci operacyjnej.

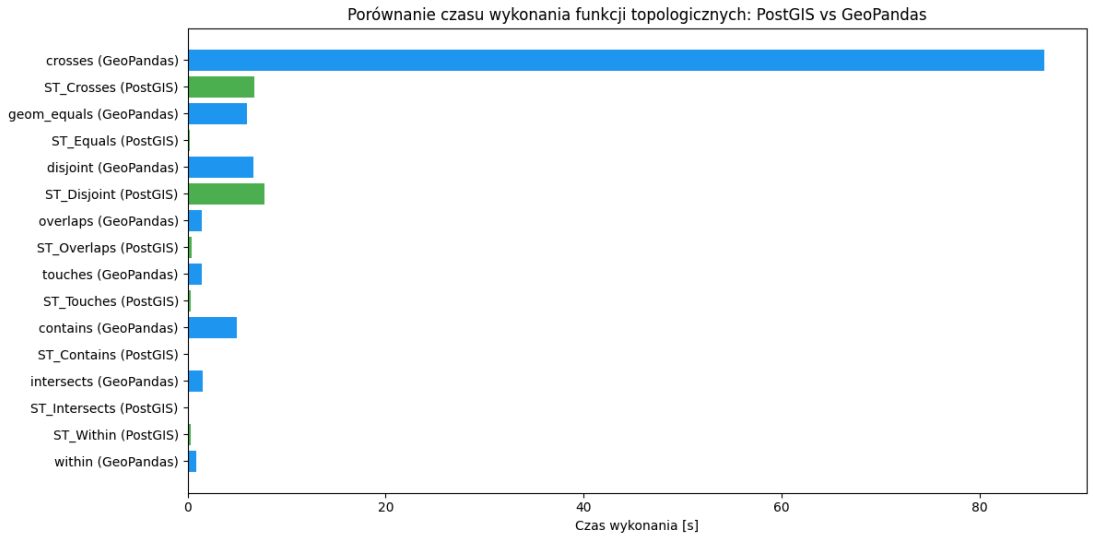
> W każdym z testów zaprezentowano i opisano:
>
> - jak wykonać przykładowe zapytanie, oraz na jakiej zasadzie działają metody wykonujące to zapytanie (*wraz z odnośnikiem do dokumentacji*)
> - wykazano różnicę między sposobem realizacji zapytania

- zmierzono czasy wywoływania zapytań
- wykazano braki w bibliotece **GeoPandas** w stosunku do możliwości udostępnianych przez **PostGIS**

Możliwości zapytań na podstawie relacji topologicznych:

Funkcja PostGIS	Funkcja w GeoPandas	Różnice
ST_Within(A,B)	A.within(B)	W GeoPandas — brak możliwości bezpośredniego wykonania zapytania dla każdej geometrii ze zbioru A z którąkolwiek geometrią ze zbioru B. Aby to zrobić należy wykorzystać funkcję agregującą `union_all()` .
ST_Intersects(A, B)	A.intersects(B)	W GeoPandas — nie są zwracane geometrie, lecz seria wartości typu `bool` , którą następnie trzeba nałożyć na zbiór podstawowy.
ST_Contains(A, B)	A.contains(B)	-
ST_Touches(A, B)	A.touches(B)	-
ST_Overlaps(A, B)	A.overlaps(B)	-
ST_Disjoint(A, B)	A.disjoint(B)	-
ST_Equals(A, B)	A.geom_equals(B)	W PostGIS — porównanie jednej geometrii wobec drugiej. Aby sprawdzić serie geometrii należy użyć złączenia.
-	A.geom_equals_exact(B, tolerance=...)	w GeoPandas — Porównanie punkt po punkcie z podaną tolerancją. Brak w PostGIS.
-	A.geom_almost_equals(B)	W GeoPandas — Przybliżona równość jest sprawdzana w każdym punkcie z dokładnością do określonego miejsca po przecinku. Brak w PostGIS
ST_Crosses(A, B)	A.crosses(B)	-

Porównanie czasów wykonania:



Wnioski

Poprzez przeprowadzone testy udało się pokazać możliwości realizacji zapytań na podstawie relacji topologicznych. Wszystkie funkcję dostępne w PostGIS mają swój odpowiednik w bibliotece GeoPandas. Są to funkcję podstawowe jeżeli chodzi o relację topologiczne, w związku z czym nie ma większych różnic między sposobem ich działania.

Widać natomiast różnicę w sposobie wykonywania zapytań. Większość zapytań w GeoPandas operuję na pojedynczych geometriach, w związku z czym aby możliwe było wykonanie zapytania **każdy z każdym** potrzebne jest wykonanie dodatkowych operacji np. scalenie serii geometrii **B** za pomocą agregacji `union_all()` lub wykorzystanie funkcji lambda w pythonie a następnie wykorzystanie maski aby uzyskać oczekiwany wynik:

```
disjoint_mask = buildings.geometry.apply(lambda b: np.all(parks.geometry.disjoint(b)))
buildings_disjoint_parks = buildings[disjoint_mask]
```

Oznacza to, że choć GeoPandas oferuję podobny zakres możliwości, to sposób ich realizacji różni się istotnie pod względem wydajności i przede wszystkim wygody operowania na dużych zbiorach danych. W PostGIS zapytania typu "każdy z każdym" są

realizowane wydajnie wewnątrz bazy danych za pomocą SQL, natomiast w GeoPandas wymagają ręcznego programowania logiki iteracyjnej, np. przez `apply()` i funkcje lambda.

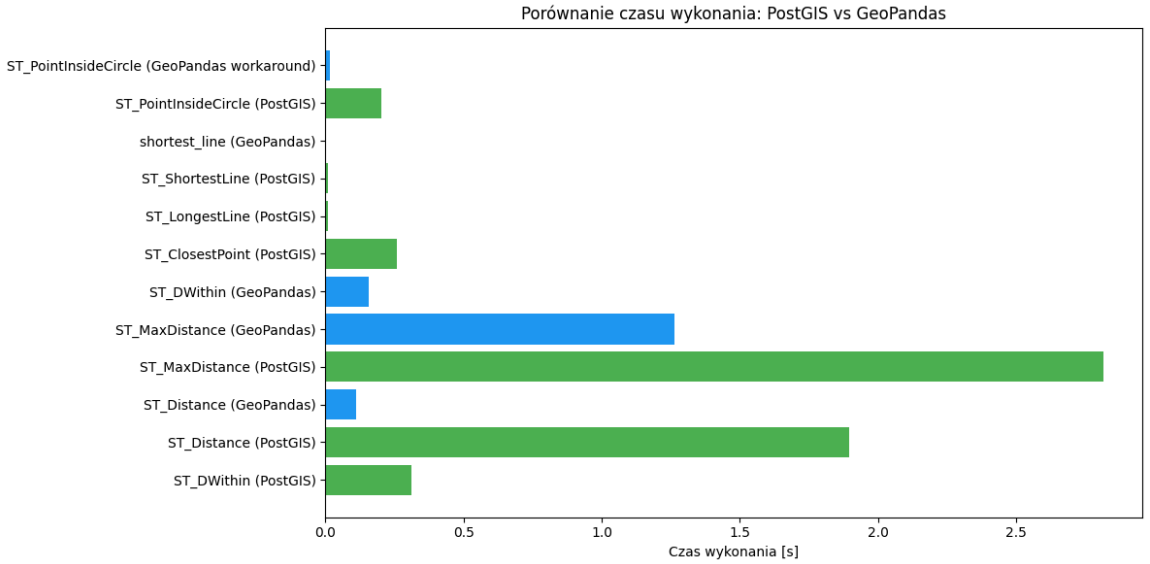
Wydajność — Jak widać na wykresie, **funkcje PostGIS są szybsze** od odpowiedników w GeoPandas. Wynika to z faktu, że PostGIS działa bezpośrednio na bazie danych, korzystając z indeksów przestrzennych i zoptymalizowanych algorytmów. GeoPandas natomiast wczytuje dane do pamięci RAM i operuje na nich w Pythonie, co przy większych zbiorach danych prowadzi do znacznie dłuższych czasów wywołania.

Wniosek — W przypadku zapytań o relacje topologiczne, system bazodanowy z PostGIS uruchamiany w środowisku pythonowym za pomocą SQLAlchemy jest bardzo wydajną alternatywą dla radzących sobie wolniej i bardziej skomplikowanych pod względem wprowadzania logiki zapytań GeoPandas. Warto jednak zaznaczyć, że wszystkie możliwości dostępne w PostGIS są dostępne dla użytkownika GeoPandas, lecz wymagają dodatkowej pracy przy danych.

Możliwości zapytań na podstawie relacji odległościowych:

Funkcja PostGIS	Funkcja w GeoPandas/Shapely	Różnice
ST_DWithin(A, B, d)	A.dwithin(B, d)	-
ST_Distance(A, B)	A.distance(B)	W GeoPandas — działa tylko w trybie 1:1, aby wykonać każdy z każdym należy użyć funkcji lambda do manipulacji danych.
ST_MaxDistance(A, B)	A.hausdorff_distance(B)	Hausdorff distance nie zawsze daje identyczny wynik jak ST_MaxDistance ze względu na to iż nie jest to największa odległość między dwiema geometriami a największa odległość między punktem geometrii A a najbliższym punktem geometrii B. A więc nie jest to bezpośredni odpowiednik funkcji ST_MaxDistance.
ST_ClosestPoint(A, B)	-	Brak odpowiednika w GeoPandas.
ST_LongestLine(A, B)	-	Brak odpowiednika w GeoPandas.
ST_ShortestLine(A, B)	A.shortest_line(B)	-
ST_PointInsideCircle(P, x, y, r)	P.within(Point(x, y).buffer(r))	Brak bezpośredniego odpowiednika. W GeoPandas trzeba ręcznie utworzyć bufor (koło) i sprawdzić relację within.

Porównanie czasów wykonania



Wnioski

Przeprowadzone testy pokazały, że **podstawowe zapytania o relacje odległościowye** można zrealizować zarówno w PostGIS, jak i w GeoPandas.

Jednak:

- **PostGIS** oferuje znacznie szerszy zakres funkcji (np. ST_ClosestPoint, ST_LongestLine), które nie mają bezpośredniego odpowiednika w bibliotece GeoPandas i których nie da się łatwo odtworzyć w GeoPandas.
- **GeoPandas** wymaga często ręcznego programowania logiki (np. przez lambdy, apply, tworzenie bufora,), szczególnie dla zapytań typu "każdy z każdym" lub gdy nie ma natywnego odpowiednika funkcji PostGIS.

Wydajność:

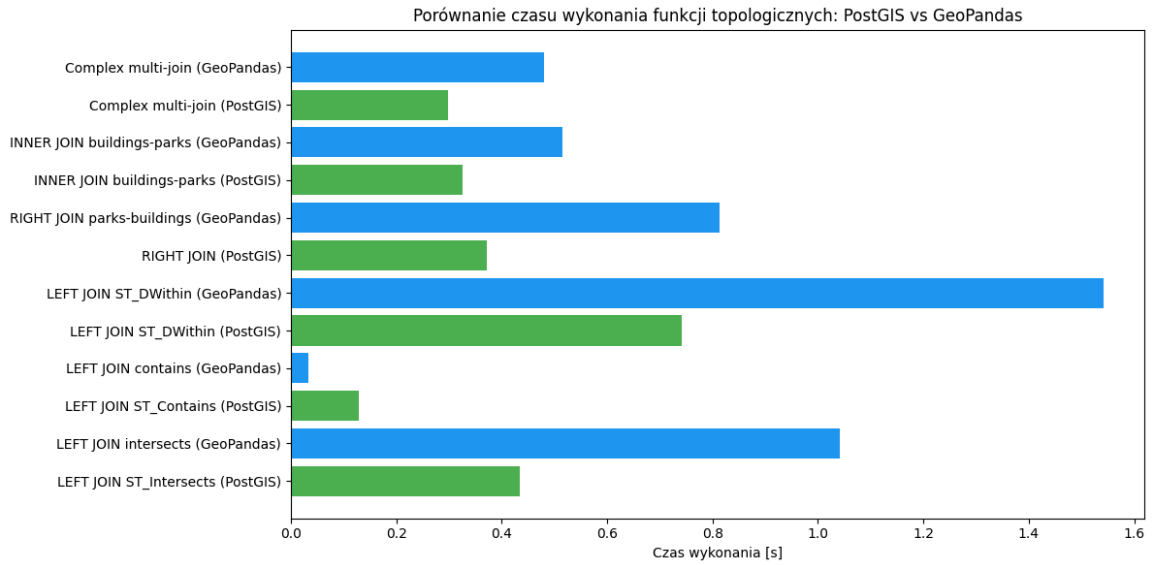
- GeoPandas radzi sobie szybciej z zapytaniami na podstawie relacji odległościowych. Przede wszystkim jest to spowodowane przetwarzaniem zbioru danych w pamięci operacyjnej. Dane są już wczytane i nie ma potrzeby ich bezpośredniego wyciągania.

Wniosek — W przypadku zapytań o relacje odległościowe PostGIS jest znacznie bardziej elastyczny. Jego możliwości tworzenia takich zapytań są szersze (chociażby funkcja. ST_LongestLine, ST_ClosestPoint czy ST_PointInsideCircle, których nie znajdziemy w GeoPandas). W praktyce do przeprowadzania bardziej zaawansowanych analiz przestrzennych lepiej sprawdzi się PostGIS. Jeżeli zależy nam jednak na szybkości wyliczenia prostych relacji, które udostępnia GeoPandas i nie potrzebne są nam bardziej zaawansowane metody analizy, warto skorzystać z biblioteki GeoPandas.

Możliwości złączeń przestrzennych (spatial joins):

Typ złączenia / Funkcja PostGIS	Funkcja w GeoPandas	Różnice
INNER JOIN + ST_Intersects	`sjoin(..., how="inner", predicate="intersects")`	Wynik analogiczny. W PostGIS można łączyć wiele warunków i typów geometrii, w GeoPandas predykaty ograniczone do funkcji z Shapely (intersects, contains, within, touches, crosses, overlaps)
LEFT JOIN + ST_Intersects	`sjoin(..., how="left", predicate="intersects")`	-
RIGHT JOIN + ST_Intersects	`sjoin(..., how="right", predicate="intersects")`	Wynik analogiczny. W GeoPandas RIGHT JOIN jest dostępny, ale mniej wydajny dla dużych zbiorów.
FULL OUTER JOIN	-	Brak wsparcia w GeoPandas. Można próbować łączyć wyniki kilku joinów, ale jest to nieefektywne.
CROSS JOIN + ST_Intersects	-	Brak wsparcia w GeoPandas. W PostGIS można łatwo wykonać zapytania typu "każdy z każdym".
LATERAL JOIN (np. najbliższy sąsiad)	-	Brak wsparcia w GeoPandas. W PostGIS można dynamicznie wybierać np. najbliższy punkt dla każdego obiektu.
ST_Contains, ST_Within, ST_DWithin	`sjoin(..., predicate="contains/within")`, `sjoin_nearest`	W GeoPandas predykaty odpowiadają wyżej wspomnianym funkcjom Shapely. Dla ST_DWithin można użyć `sjoin_nearest` z `max_distance`.
Złożone multi-joiny	sekwencja `sjoin`	W PostGIS można wykonać wielokrotne złączenia w jednym zapytaniu SQL. W GeoPandas trzeba wykonywać kolejne złączenia krok po kroku na DataFrame'ach.

Porównanie czasów wykonania:



Wnioski

Przeprowadzone testy pokazują, że zarówno **PostGIS,** jak i **GeoPandas** umożliwiają realizację złączeń przestrzennych (INNER, LEFT, RIGHT JOIN) z wykorzystaniem relacji topologicznych takich jak ST_Intersects, ST_Contains, ST_Within czy ST_DWithin. Jednak:

PostGIS oferuje pełen zakres złączeń SQL (w tym FULL OUTER JOIN, CROSS JOIN, LATERAL JOIN) oraz możliwość budowania złożonych, wieloetapowych zapytań w jednym kroku. Pozwala to na bardzo wydajne i elastyczne analizy przestrzenne wykonując jedno zapytanie, bez konieczności posługiwania się pośrednimi odpowiedziami.

GeoPandas wspiera tylko trzy podstawowe typy (do nie dawna były to tylko dwa, bez right join) złączeń (`left` , `right` , `inner`) i nie oferuje natywnie FULL OUTER JOIN ani zaawansowanych złączeń typu LATERAL czy CROSS JOIN. Złożone analizy wymagają sekwencyjnego wykonywania wielu joinów na DataFrame'ach, co jest mniej wydajne i bardziej skomplikowane. Konieczne jest posługiwanie się pośrednimi odpowiedziami w celu manipulacji danych, tak by przekształcić je do oczekiwanego efektu.

Wydajność:

- **PostGIS** ponownie, jak w przypadku relacji topologicznych, jest znacznie szybszy (poza ST_Contains). Wykorzystanie indeksów przestrzennych i wykonywanie operacje bezpośrednio w systemie bazodanowym powoduje szybsze uzyskanie odpowiedzi na zapytanie.
- **GeoPandas** działa w pamięci RAM i przy większych danych staje się wyraźnie wolniejszy. Widać to w szczególność dla nieco bardziej złożonego zapytania z predykatem DWithin.

Elastyczność:

- PostGIS pozwala na dynamiczne podzapytania (np. LATERAL JOIN – najbliższy sąsiad), których nie da się łatwo odtworzyć w GeoPandas. Pozwala również na wielokrotne zagnieżdżanie złączeń bez konieczności posługiwania się pośrednimi wynikami.
- GeoPandas jest wygodny do szybkiej eksploracji i wizualizacji, ale ograniczony w zakresie wykonywania złączeń.

Wnioski — PostGIS radzi sobie zdecydowanie lepiej i oferuje znacznie szerszy wachlarz możliwości w przypadku złączeń przestrzennych. Użytkownik uzyskuje wyniki szybciej oraz ma większą elastyczność przy tworzeniu zapytań. W przypadku złączeń, nie zauważyliśmy benefitów korzystania z biblioteki GeoPandas.

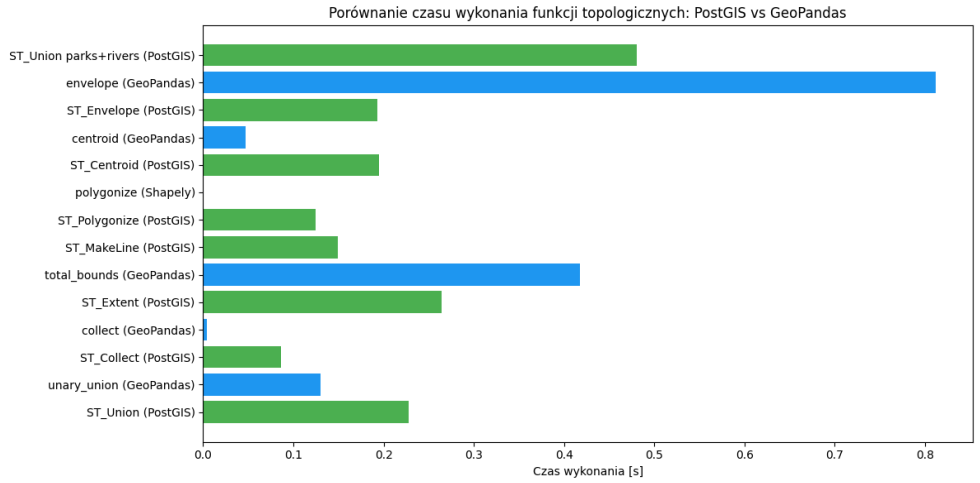
Możliwości funkcji agregujących: PostGIS vs GeoPandas

Zakres funkcji agregujących dostępnych w PostGIS jest znacznie szerszy niż w GeoPandas. Przetestowane zostały wybrane funkcje, które posiadają swoje odpowiedniki w GeoPandas (lub Shapely).

Wszystkie funkcje agregujące można znaleźć w dokumentacji.

Funkcja PostGIS	Funkcja w GeoPandas/Shapely	Różnice i uwagi
ST_Union(geom)	A.union(B)	Wynik geometrycznie równoważny. W GeoPandas `union_all` na serii GeoSeries.
ST_UnaryUnion(geom)	A.union_all()	-
ST_Collect(geom)	geopandas.tools.collect()	Brak odpowiednika bezpośrednio w GeoPandas. Funkcja znajduje się w module tools, nie jako metoda DataFrame.
ST_Extent(geom)	gdf.total_bounds	ST_Extent zwraca BOX (tekst), total_bounds tablicę liczb. Tablicę liczb łatwiej zamienić na POLYGON w celu dalszej analizy lub wizualizacji wyników.
ST_MakeLine(geom)	-	Brak bezpośredniego odpowiednika w GeoPandas.
ST_Polygonize(geom)	shapely.ops.polygonize()	W GeoPandas przez Shapely, nie bezpośrednio. Wynik geometrycznie zgodny.
ST_Centroid(geom)	gdf.centroid, unary_union.centroid	-
ST_Envelope(geom)	gdf.envelope	Wynik geometrycznie zgodny, różnice w obsłudze CRS i precyzji.

Porównanie czasów wykonania



Wnioski

Wyniki testów wykazują, że główne funkcje agregujące dostępne w PostGIS mają swoje odpowiedniki w GeoPandas (lub Shapely). Wyniki są w większości geometryczne są zgodne, choć format oraz precyzja zwracanych danych bywają różne (np. BOX vs tablica

liczb).

Różnice funkcjonalne:

- PostGIS oferuje znacznie szerszy zakres funkcji agregujących, w tym zaawansowane operacje 3D, klastrowanie czy agregacje na rasterach, które nie mają odpowiedników w GeoPandas.
- W GeoPandas niektóre funkcje są dostępne tylko przez Shapely lub wymagają dodatkowego obejścia (np. polygonize, makeline).
- W GeoPandas część funkcji agregujących nie jest dostępna bezpośrednio na obiektach GeoDataFrame, lecz w osobnych modułach lub jako metody przynależące do innych klas. Powoduje to brak spójności.

Wydajność:

Wydajność każdej z funkcji agregujących jest różna, w zależności od funkcji, jest ona szybsza w PostGIS lub GeoPandas. Nie ma jednoznacznego faworyta pod względem wydajności.

Wygoda użycia:

PostGIS pozwala na bardzo proste i czytelne zapytania SQL, które agregują dane w jednym kroku. W GeoPandas często wymagane jest dodatkowe przetwarzanie lub korzystanie z funkcji z innych bibliotek.

Wnioski — Pod względem możliwości funkcji agregujących lepiej wypada PostGIS który oferuję znacznie większą ilość możliwości. Dodatkowym atutem jest czytelna i klarowna dokumentacja, z którą praca (naszym subiektywnym zdaniem) była wygodniejsza niż w przypadku GeoPandas.

GeoPandas sprawdzi się dla użytkowników nie potrzebujących bardziej zaawansowanych możliwości agregacji oraz pracujących na mniejszych zbiorach danych. Konieczność szukania funkcjonalności po różnych bibliotekach związanych z GeoPandas oraz częste wykonywanie obejść w postaci funkcji lambda czy wyników pośrednich sugeruje jednak na skorzystanie z PostGIS do analizy danych przestrzennych.

Eksperymenty z ograniczoną pamięcią operacyjną

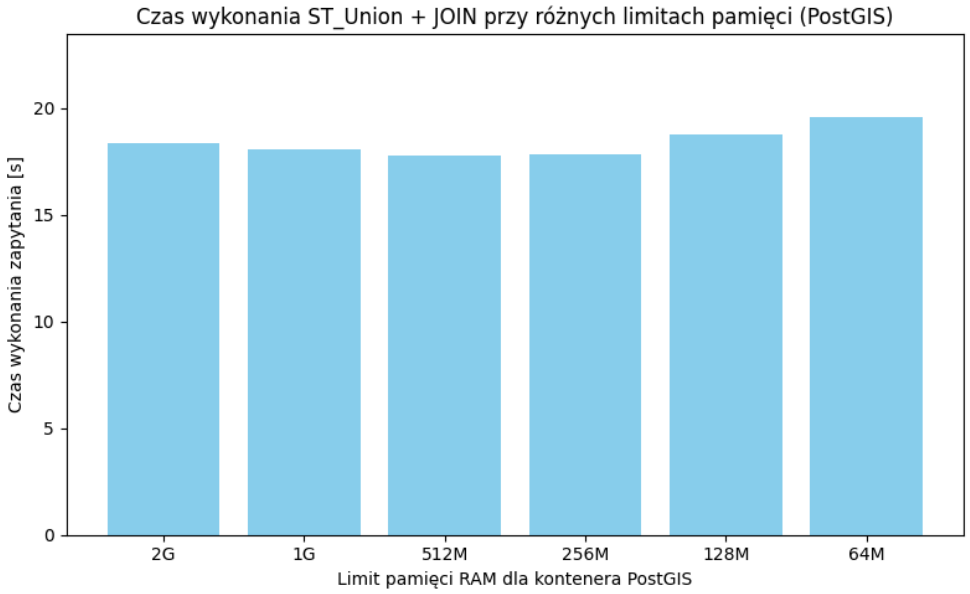
Przygotowano skrypty uruchomieniowe dla eksperymentów ograniczających pamięć operacyjną.

Eksperymenty polegały na zmniejszeniu pamięci operacyjnej dla kontenerów, w których wykonywane były operacje przestrzenne.

Pamięć stała (tzw. swap memmory) została na niezmienionym poziomie 16GB. Natomiast pamięć operacyjną zmieniano w następujący sposób:

- 2GB
- 1GB
- 512MB
- 256MB
- 128MB
- 64MB

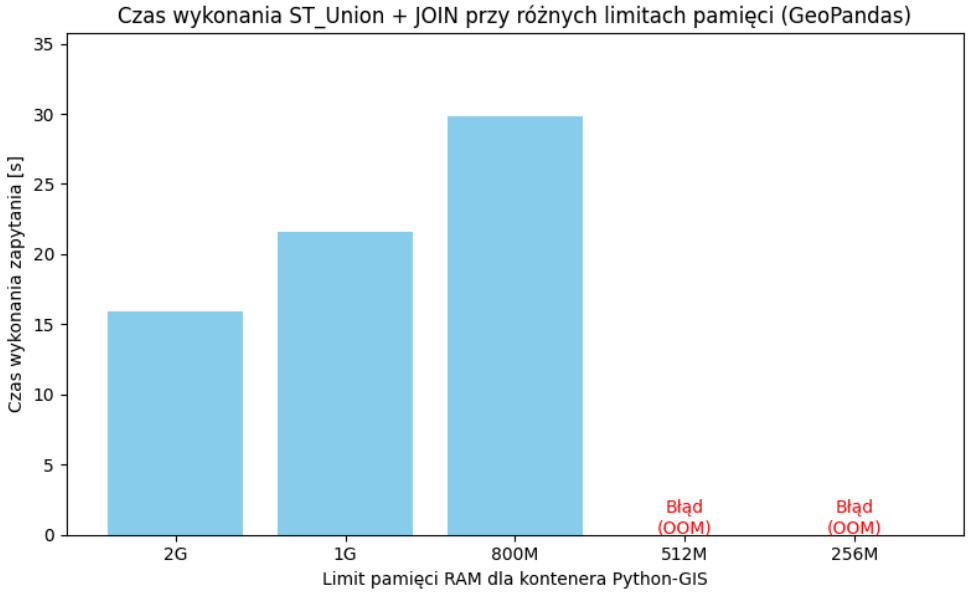
Wyniki i interpretacja



Podczas zmniejszania pamięci aż do 256MB następuje zmniejszenie czasu wykonania zapytania. Czasy wykonania zapytań przy niższych limitach pamięci (np. 512M, 256M) są krótsze niż przy większych limitach, ponieważ dane zostają zbuforowane przez bazę danych (cache dysku, cache zapytań).
Przy powtarzanych zapytaniach PostGIS korzysta z już wczytanych do pamięci stron danych, co przyspiesza wykonanie, dopóki nie zabraknie pamięci na nowe operacje.

Zmniejszając limity dalej (128MB oraz 64MB) widoczny jest już spadek wydajności. Mniejsza pamięć operacyjna sprawia, że czas wykonania zapytania jest dłuższy pomimo zbuforowania części zapytania.

Nie nastąpił brak pamięci operacyjnej, który spowodowałby błąd, nawet dla tak małej ilości pamięci jak 64MB.



Przy ograniczeniach pamięci operacyjnej (RAM) na poziomie 1GB i 800MB, czasy wykonania zapytań poprzez GeoPandas rosną. Jest to spodziewany efekt, ponieważ nie następuje tutaj buforowanie danych w pamięci operacyjnej, a GeoPandas musi każdorazowo, z coraz to mniejszą ilością pamięci przetrzymywać tą samą ilośc danych oraz wykonywać zapytanie / operacje. Juz przy 512M następuje "Out of memory" error.

Wniosek — PostGIS radzi sobie bardzo dobrze, nawet przy ograniczonych zasobach pamięci operacyjnej. GeoPandas jako biblioteka python'owa operuje wyłącznie w pamięci RAM, tak więc zmniejszając jej ilość, rośnie czas wykonywania zapytania.

Ponowne wykonanie eksperymentów

Aby ponownie uruchomić eksperymenty należy przejść do katalogu głównego repozytorium oraz wykonać komendę:

```
docker-compose up -d
```

która uruchomi kontenery oraz przygotuje je do pracy z danymi.

Uruchomienie środowiska dla notebook'ów

Znajdując się w głównym katalogu należy przejść do podkatalogu `python` oraz uruchomić środowisko wirtualne python poprzez:

```
$ cd python
$ poetry install
```

Następnie stworzyć kernel dla notatników:

```
$ poetry run python -m ipykernel install --user --name=poetry-spdb --display-name "Poetry (spdb)"
```

Należy uruchomić edytor kodu np. VSCode oraz wejść w jeden z przygotowanych notatników:

- `topological_relations.ipynb`
- `distance_relations_tests.ipynb`
- `spatial_joins.ipynb`
- `aggregate_functions.ipynb`

a następnie wybrać dla niego kernel o nazwie `Poetry (spdb)`.

Uruchomienie testów z ograniczoną pamięcią

Aby ponownie przeprowadzić testy z ograniczoną pamięcią należy upewnić się, że oba kontenery (postgis oraz python) są uruchomione.

Będąc w głównym katalogu (root) repozytorium uruchomić skrypt:

```
./run_stress_tests.sh
```

Wyniki zostaną zapisane w plikach:

- stress_test_results_postgis.csv
- stress_test_results_geopandas.csv

Aby ponownie utworzyć wykresy należy uruchomić notatnik `stress_tests.ipynb`

Podsumowanie

Realizacja projektu umożliwiła porównanie możliwości realizacji zapytań o dane przestrzenne za pomocą GeoPandas(Python) z systemem bazodanowym z rozszerzeniem PostGIS.

Rezultaty przeprowadzonych eksperymentów sugerują, że baza danych PostgreSQL + PostGIS udostępniają więcej możliwości tworzenia zapytań w kontekście:

- relacji topologicznych
- relacji odległościowych
- możliwości złączeń przestrzennych
- funkcji agregujących

oraz możliwości działania z ograniczoną liczbą pamięci operacyjnej RAM.

PostGIS + SQLAlchemy w środowisku python'owym sprawuje się dobrze, w znaczącej większości udostępnianych możliwości lepiej niż biblioteka GeoPandas. To bardzo dobrze świadczy o tym narzędziu to przeprowadzania zapytań i analiz danych przestrzennych oraz jest dobrym zastąpieniem bibliotek, które na aktualny moment nie mają tak szerokich możliwości jak PostGIS.

Czas wykonywania zapytań oraz elastyczność prezentowanych funkcji, ich dostępność, łatwy sposób użycia oraz możliwość tworzenia złożonych zapytań, a także bardzo dobrze przygotowana dokumentacja czynią z PostGIS narzędzie bardziej efektywne i skalowalne, szczególnie w przypadku pracy z dużymi zbiorami danych przestrzennych.

Z kolei GeoPandas, mimo swoich ograniczeń, wciąż pozostaje wartościowym narzędziem w przypadku mniejszych projektów, analiz lokalnych lub prototypowania. Jego prostota i integracja z popularnym ekosystemem narzędzi naukowych Pythona (np. Pandas, Matplotlib) czyni go przyjaznym rozwiązaniem dla analityków i badaczy nieposiadających doświadczenia z przestrzennymi bazami danych lecz posiadających wiedzę i praktykę w korzystaniu z języka Python oraz bibliotek jak np. Pandas.

Podsumowując, wybór narzędzia powinien być uzależniony od specyfiki projektu, wielkości danych oraz oczekiwanej wydajności i funkcjonalności. Dla projektów wymagających zaawansowanych analiz przestrzennych oraz dobrej wydajności – PostgreSQL z rozszerzeniem PostGIS okazuje się rozwiązaniem bardziej kompleksowym i profesjonalnym. Dodatkowym atutem jest możliwość integracji tego rozwiązania w środowisko python przy użyciu SQLAlchemy co pokazały przeprowadzone eksperymenty.

źródła do dokumentacji

- https://postgis.net/documentation/
- https://geopandas.org/en/stable/docs.html