Dokumentacja Projektu: Porównanie wydajności operacji CRUD na dużym zbiorze danych w PostgreSQL, MongoDB, Redis - Jakub Chrzanowski

1. Wprowadzenie

Celem projektu jest analiza i porównanie wydajności operacji CRUD (Create, Read, Update, Delete) na dużym zbiorze danych w trzech popularnych systemach bazodanowych, reprezentujących różne paradygmaty:

- **PostgreSQL**: Relacyjna baza danych (SQL).
- **MongoDB**: Dokumentowa baza danych (NoSQL).
- **Redis**: Baza danych klucz-wartość, działająca w pamięci (In-Memory NoSQL).

Jako zbiór danych wykorzystano metadane publikacji z serwisu arXiv, udostępnione na platformie Kaggle. Środowisko testowe zostało w pełni skonteneryzowane przy użyciu technologii Docker, a skrypty benchmarkowe napisano w języku Python.

2. Architektura i Modelowanie Danych

Kluczowym aspektem projektu jest proces adaptacji (modelowania) danych źródłowych w formacie JSON do specyfiki każdej z baz.

2.1. Przegląd Baz Danych

- **PostgreSQL (Relacyjna)**: Wymusza ścisłą, zdefiniowaną strukturę danych w postaci tabel, kolumn i relacji między nimi. Gwarantuje spójność danych (ACID). Idealna do danych transakcyjnych i analitycznych, gdzie integralność jest kluczowa.
- **MongoDB (Dokumentowa)**: Przechowuje dane w elastycznych, zagnieżdżonych dokumentach (podobnych do JSON). Nie wymaga sztywnego schematu, co ułatwia pracę z danymi o złożonej lub zmieniającej się strukturze.
- **Redis (Klucz-Wartość)**: Najprostszy model, przechowujący dane jako pary unikalnych kluczy i odpowiadających im wartości. Działa w pamięci RAM, co zapewnia ekstremalną szybkość dla prostych operacji odczytu i zapisu.

2.2. Dane Źródłowe (JSON)

Każdy rekord w zbiorze danych to obiekt JSON opisujący jedną publikację naukową. Jego uproszczona struktura wygląda następująco:

{
"id": "0704.0001",
"title": "Computation of a chaotic invariant set",
"authors_parsed": [["Dettmann", "C. P.", ""], ["Cohen", "E. G. D.", ""]],
"categories": "nlin.CD math.DS",
"abstract": "..."
}

Największym wyzwaniem modelowania są pola `authors_parsed` i `categories`, które reprezentują relacje jeden-do-wielu.

2.3. Adaptacja Danych dla PostgreSQL

```
def load_data_to_postgres(conn):
    print("\n--- Starting data load to PostgreSQL ---")
    start_time = time.time()
    with conn.cursor() as cur:
        print("  Cleaning PostgreSQL tables...")
        cur.execute("TRUNCATE TABLE papers, authors, categories RESTART
IDENTITY CASCADE;")
        conn.commit()
        with open(config.DATA_FILE_PATH, 'r') as f:
            for i, line in enumerate(f):
                if config.RECORD_LIMIT and i >= config.RECORD_LIMIT: break
                record = json.loads(line)
                cur.execute("INSERT INTO papers (id, title, abstract, doi,
submitter, update_date) VALUES (%s, %s, %s, %s, %s, %s) ON CONFLICT (id)
DO NOTHING",
                            (record.get('id'), record.get('title'),
record.get('abstract'), record.get('doi'), record.get('submitter'),
record.get('update_date')))
                authors = record.get('authors_parsed', [])
                for author_parts in authors:
                    author_name = " ".join(filter(None, [author_parts[1],
author_parts[0]]))
                    if author_name:
                        author_id = get_or_create_id(cur, 'authors',
'author_id', 'author_name', author_name)
```

```
                        cur.execute("INSERT INTO paper_authors (paper_id,
author_id) VALUES (%s, %s) ON CONFLICT DO NOTHING", (record['id'],
author_id))
                categories = record.get('categories', '').split()
                for category_name in categories:
                    category_id = get_or_create_id(cur, 'categories',
'category_id', 'category_name', category_name)
                    cur.execute("INSERT INTO paper_categories (paper_id,
category_id) VALUES (%s, %s) ON CONFLICT DO NOTHING", (record['id'],
category_id))
                if (i + 1) % 5000 == 0: print(f"  Processed {i+1}
records...")
    conn.commit()
    print(f"✅ Finished loading data to PostgreSQL. Time: {time.time() -
start_time:.2f} s")
```

Wymagany był proces normalizacji, czyli rozbicia płaskiego obiektu JSON na wiele powiązanych tabel w celu uniknięcia redundancji i zapewnienia integralności. Stworzono następujący schemat:
1. Tabela `papers`: Główna tabela z informacjami o artykule (`id`, `title`, `abstract` itp.). Kolumna `id` jest **kluczem głównym** (`PRIMARY KEY`), co automatycznie tworzy na niej indeks.
2. Tabela `authors`: Przechowuje unikalnych autorów (`author_id`, `author_name`).
3. Tabela `categories`: Przechowuje unikalne kategorie (`category_id`, `category_name`).
4. Tabele łączące (`paper_authors` i `paper_categories`): Tabele pośredniczące, które modelują relacje wiele-do-wielu, łącząc `paper_id` z odpowiednimi `author_id` i `category_id`.

Logika kodu: dla każdego rekordu JSON skrypt wykonuje serię operacji `INSERT` do wielu tabel, sprawdzając, czy dany autor lub kategoria już istnieją, aby uniknąć duplikatów.

2.4. Adaptacja Danych dla MongoDB

```
def load_data_to_mongo(db):
    print("\n--- Starting data load to MongoDB ---")
    start_time = time.time()
    print("  Cleaning 'papers' collection in MongoDB...")
    db['papers'].drop()
```

```
    papers_collection = db['papers']
    batch = []
    with open(config.DATA_FILE_PATH, 'r') as f:
        for i, line in enumerate(f):
            if config.RECORD_LIMIT and i >= config.RECORD_LIMIT: break
            record = json.loads(line)
            record['_id'] = record['id']
            try: record['update_date'] =
datetime.strptime(record['update_date'], '%Y-%m-%d')
            except (ValueError, TypeError): record['update_date'] = None
            batch.append(record)
            if len(batch) >= 1000:
                papers_collection.insert_many(batch)
                batch = []
                if (i + 1) % 5000 == 0: print(f"  Processed {i+1}
records...")
        if batch: papers_collection.insert_many(batch)
    print(f"✅ Finished loading data to MongoDB. Time: {time.time() -
start_time:.2f} s")
```

Proces był trywialny i polegał na bezpośrednim mapowaniu struktury JSON na dokument BSON w MongoDB.

* Kolekcja `papers`: Każdy rekord JSON staje się jednym dokumentem w kolekcji.
* Zagnieżdżone dane: Pola takie jak `authors_parsed` są przechowywane jako zagnieżdżone tablice wewnątrz dokumentu, co jest naturalne dla tego typu bazy.
* Klucz główny: Pole `id` z JSON zostało użyte jako unikalny identyfikator `_id` dokumentu, na którym MongoDB automatycznie tworzy indeks.

Skrypt wczytuje rekordy JSON, grupuje je w paczki (np. po 1000) i wstawia do bazy za pomocą jednej, wydajnej operacji `insert_many`.

2.5. Adaptacja Danych dla Redis

```
def load_data_to_redis(r):
    print("\n--- Starting data load to Redis ---")
    start_time = time.time()
    print("  Cleaning Redis database...")
    r.flushall()
    pipe = r.pipeline()
    with open(config.DATA_FILE_PATH, 'r') as f:
```

```
        for i, line in enumerate(f):
            if config.RECORD_LIMIT and i >= config.RECORD_LIMIT: break
            record = json.loads(line)
            paper_id = record['id']
            paper_data = {'title': record.get('title', ''), 'abstract':
record.get('abstract', ''), 'update_date': record.get('update_date', '')}
            pipe.hset(f"paper:{paper_id}", mapping=paper_data)
            categories = record.get('categories', '').split()
            for cat in categories: pipe.sadd(f"category:{cat}", paper_id)
            authors = record.get('authors_parsed', [])
            for author_parts in authors:
                author_name = "_".join(filter(None, [author_parts[1],
author_parts[0]])).replace(" ", "_")
                if author_name: pipe.sadd(f"author:{author_name}",
paper_id)
            if (i + 1) % 1000 == 0:
                pipe.execute()
                if (i + 1) % 5000 == 0: print(f"  Processed {i+1}
records...")
        pipe.execute()
    print(f"✅ Finished loading data to Redis. Time: {time.time() -
start_time:.2f} s")
```

Redis wymagał najbardziej kreatywnego podejścia. Zamiast polegać na wbudowanych mechanizmach, sami stworzyliśmy odpowiednie struktury danych.

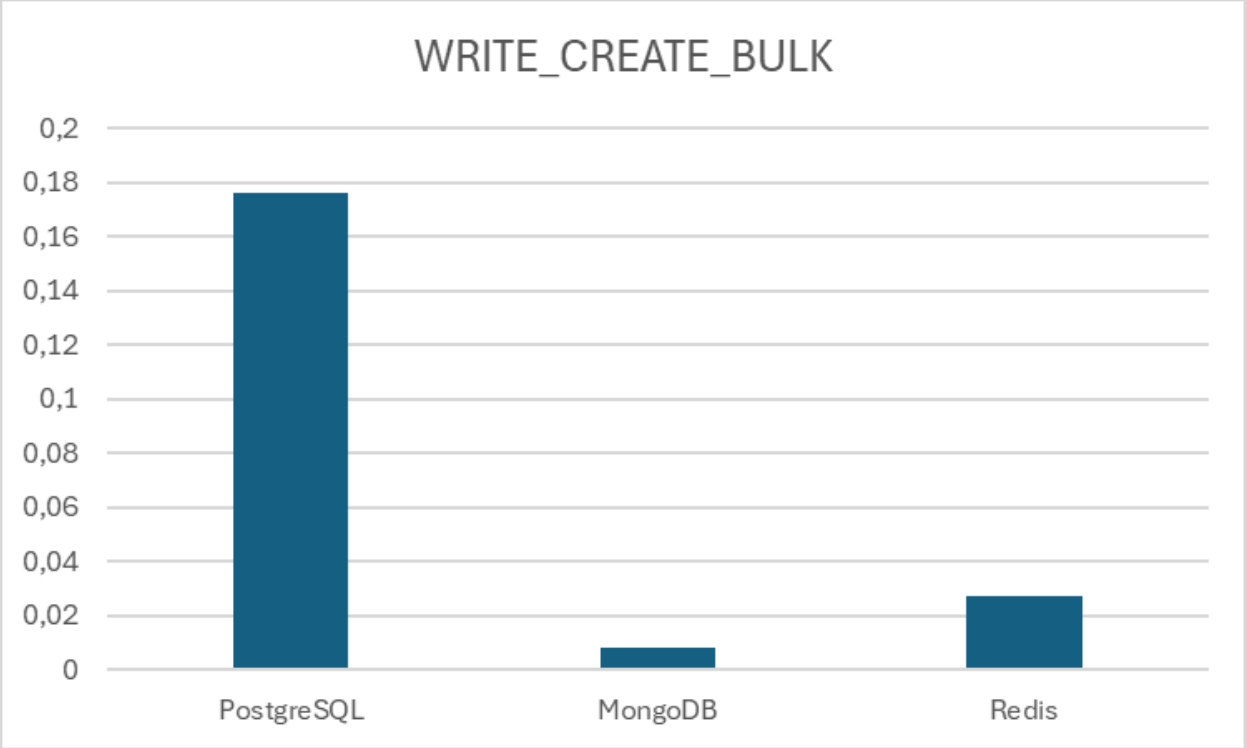
* Dane główne (`HASH`): Informacje o każdym artykule (`title`, `abstract`) są przechowywane w strukturze `HASH` pod unikalnym kluczem, np. `paper:0704.0001`.
* Indeksy wtórne (`SET`): Aby umożliwić wyszukiwanie po autorze lub kategorii, stworzyliśmy "ręczne" indeksy. Dla każdej kategorii (np. `hep-th`) tworzony jest zbiór (`SET`) o kluczu `category:hep-th`, który przechowuje unikalne ID wszystkich artykułów z tej kategorii. `SET` jest idealny, bo automatycznie dba o unikalność.

Dla każdego rekordu JSON skrypt dodaje do potoku (pipeline) serię komend: jedną `HSET` do zapisu danych głównych oraz wiele `SADD` do aktualizacji wszystkich powiązanych indeksów (kategorii i autorów). Użycie potoku drastycznie przyspiesza proces zapisu.

3. Analiza Wyników Benchmarków

Testy zostały przeprowadzone na zbiorze 1 000 000 rekordów. Poniższe wykresy i analizy bazują na wynikach z jednego uruchomienia.

Operacja 1: Zapis Masowy (WRITE_CREATE_BULK)
Test ten mierzy czas potrzebny na wstawienie 500 nowych, prostych rekordów.

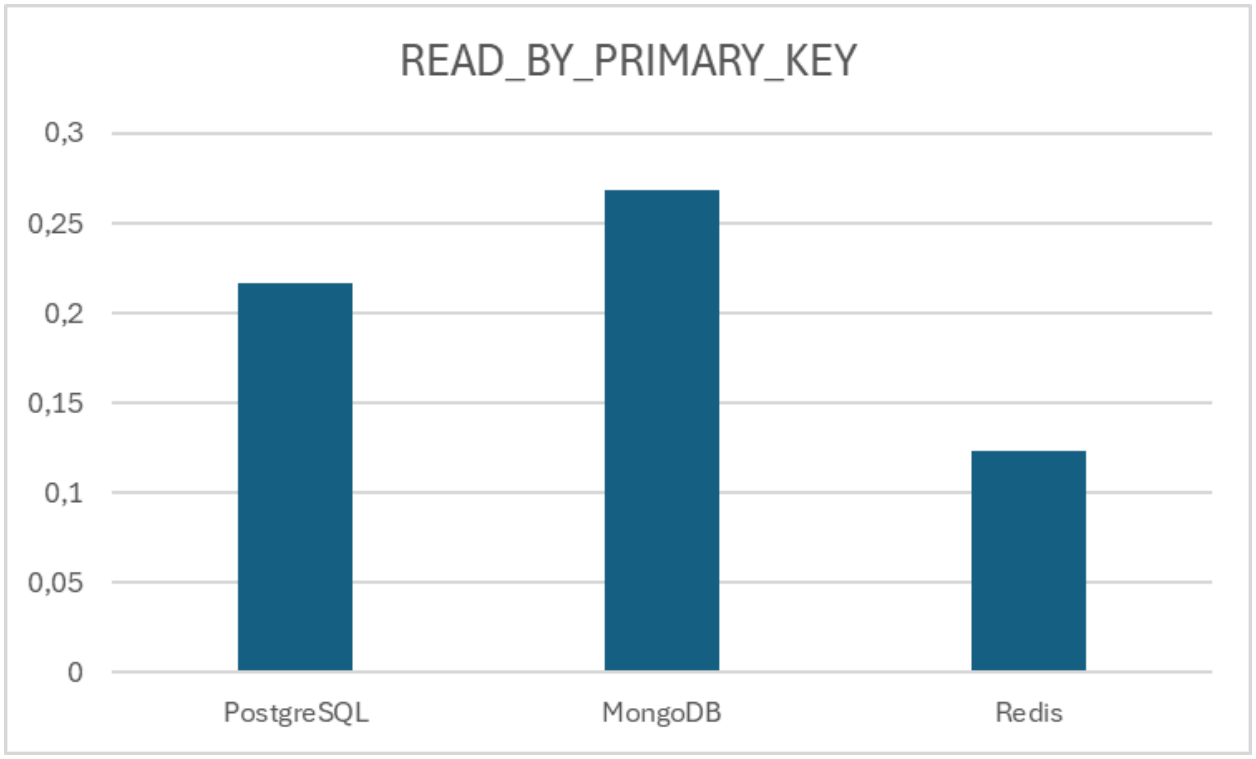


MongoDB i Redis są bezkonkurencyjne. Ich schemat zapisu jest prostszy i nie wymaga tak wielu sprawdzeń integralności jak PostgreSQL, który musi upewnić się, że klucz główny jest unikalny. MongoDB dodatkowo zyskuje dzięki operacji `insert_many`, a Redis dzięki potokowaniu komend.

Operacja 2: Odczyt po Kluczu Głównym (READ_BY_PRIMARY_KEY)
Test ten mierzy czas wykonania 500 zapytań o jeden, konkretny rekord po jego unikalnym ID.

```
start = time.time()
    for _ in range(config.BENCHMARK_N):
        r.hgetall(f"paper:{config.SAMPLE_PAPER_ID}")
    results.append({'database': 'Redis', 'operation':
'READ_BY_PRIMARY_KEY', 'time_seconds': time.time() - start,
'records_processed': config.BENCHMARK_N})
```

`HGETALL` w Redis to polecenie, które pobiera wszystkie pola i wartości z `HASHa` przechowywanego pod danym kluczem.

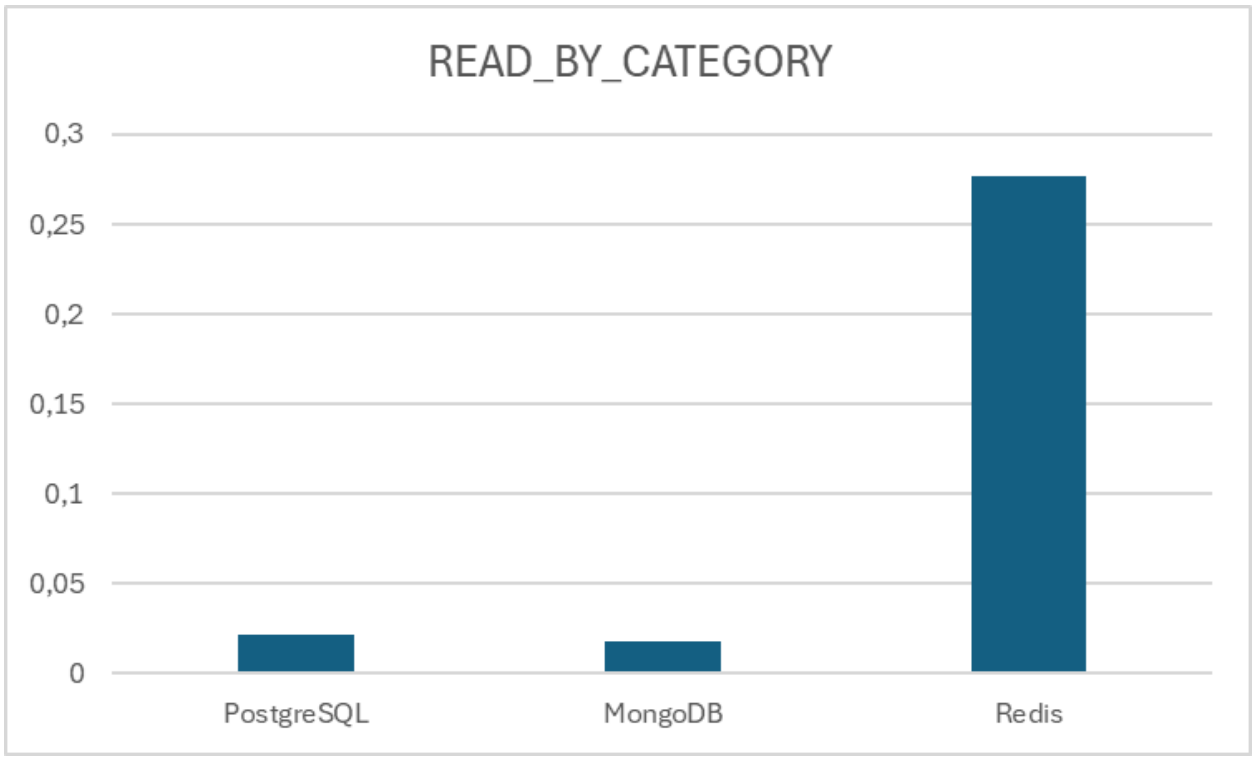


Redis, jako baza działająca w pamięci, jest tutaj najszybszy. Wszystkie bazy korzystają z wbudowanych, zoptymalizowanych indeksów na kluczu głównym, co sprawia, że czasy są stosunkowo niskie i porównywalne.

Operacja 3: Odczyt po Polu Indeksowanym (READ_BY_CATEGORY)
Test ten mierzy czas odnalezienia 100 rekordów na podstawie przynależności do jednej kategorii.

```
paper_ids_from_index =
list(r.smembers(f"category:{config.SAMPLE_CATEGORY}"))[:100]
    pipe = r.pipeline()
    for paper_id in paper_ids_from_index:
        pipe.hgetall(f"paper:{paper_id}")
    pipe.execute()
```

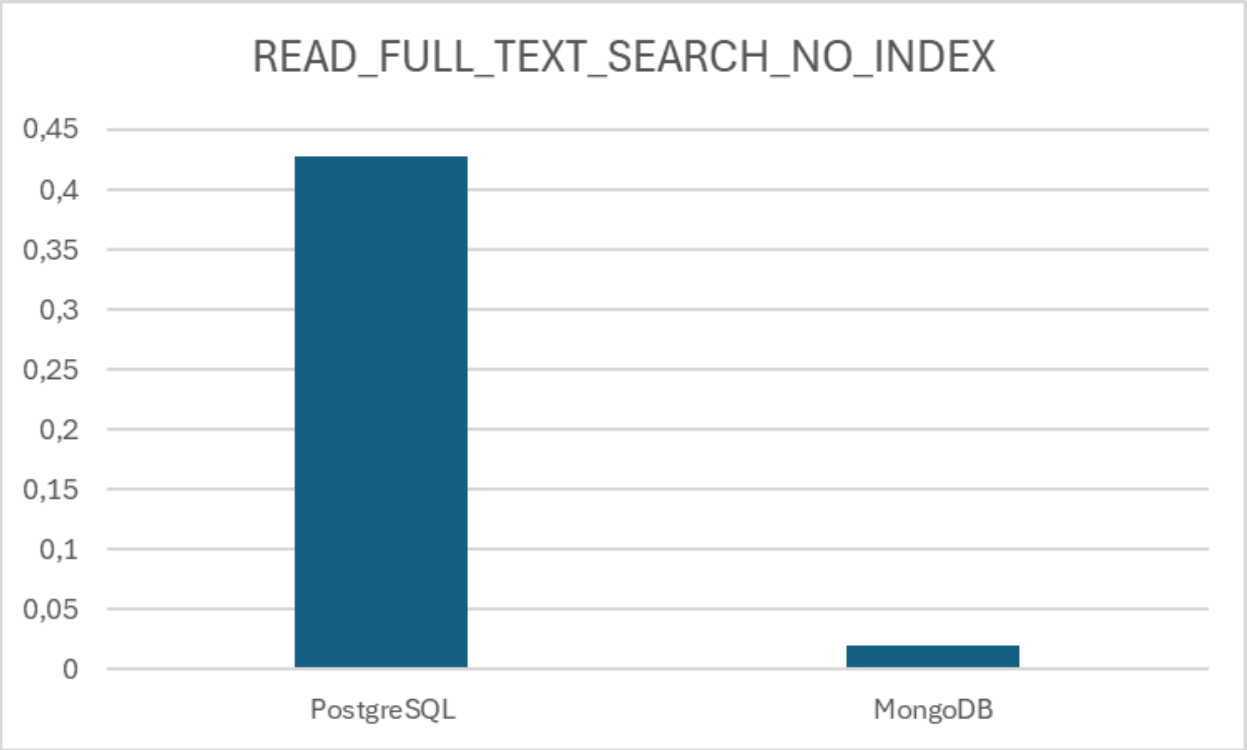
W Redis to proces dwuetapowy: smembers(f"category:{...}"): Najpierw pobieramy zawartość naszego indeksu – czyli SET zawierający wszystkie ID artykułów z danej kategorii. To jest bardzo szybkie. Następnie, w potoku, wykonujemy serię poleceń HGETALL, aby pobrać dane dla każdego z odnalezionych ID. To ten drugi krok generuje większość kosztów.



PostgreSQL i MongoDB radzą sobie dobrze. W realnej aplikacji zapytanie w Redis można by zoptymalizować.

Operacja 4: Wyszukiwanie Pełnotekstowe bez Indeksu (READ_FULL_TEXT_SEARCH_NO_INDEX)

Test ten mierzy czas odnalezienia 100 rekordów zawierających rzadkie słowo w abstrakcie, bez użycia dedykowanego indeksu.



MongoDB jest tu znacznie szybsze, co może wynikać z optymalizacji wewnętrznego silnika dla operacji na pojedynczych dokumentach. PostgreSQL musi wykonać kosztowny `TABLE SCAN`, który dla operatora `ILIKE` jest wolniejszy. Redis nie posiada takiej funkcjonalności.

Operacja 5: Wyszukiwanie Pełnotekstowe z Indeksem (READ_FULL_TEXT_SEARCH_WITH_INDEX)

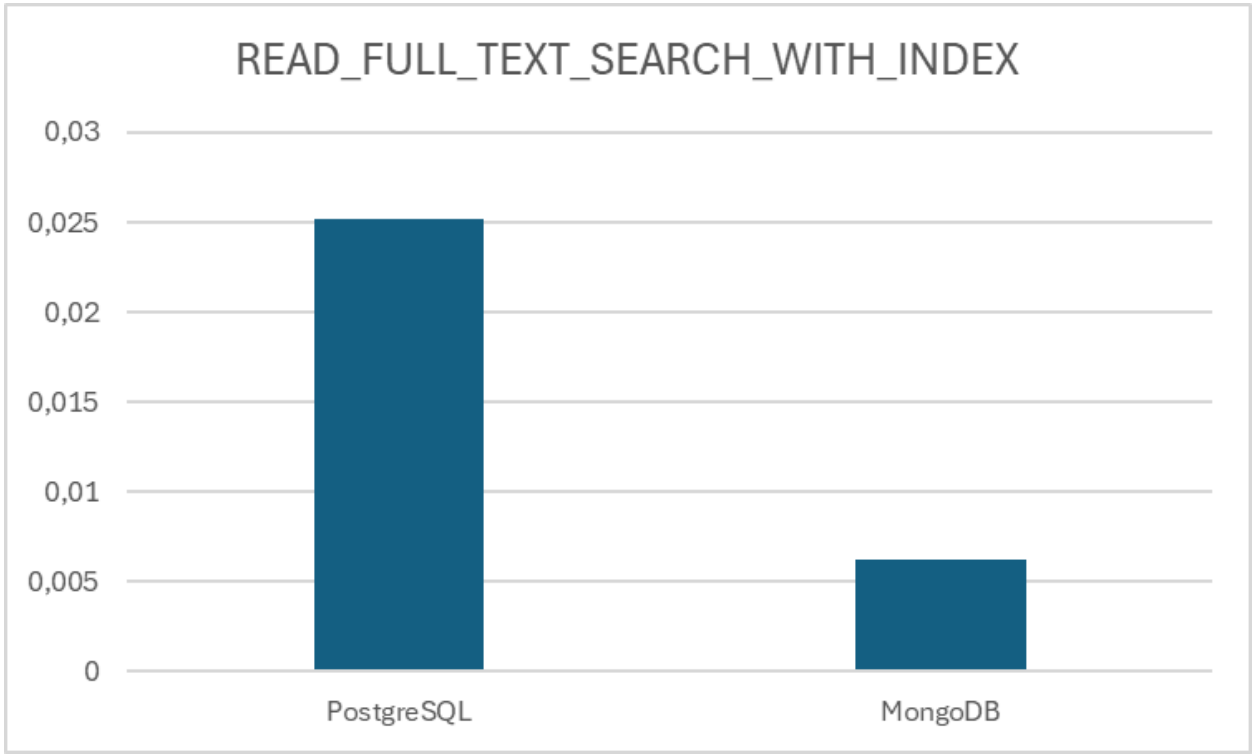
Ten sam test co powyżej, ale po wcześniejszym utworzeniu dedykowanego indeksu pełnotekstowego.

```
cur.execute("CREATE INDEX idx_papers_abstract_gin ON papers USING
gin(to_tsvector('english', abstract));")
        conn.commit()
        start = time.time()
        cur.execute("SELECT id FROM papers WHERE to_tsvector('english',
abstract) @@ to_tsquery('english', %s) LIMIT 100;",
(config.RARE_WORD_SEARCH_TERM,))
        cur.fetchall()
        results.append({'database': 'PostgreSQL', 'operation':
'READ_FULL_TEXT_SEARCH_WITH_INDEX', 'time_seconds': time.time() - start,
'records_processed': 100})
        cur.execute("DROP INDEX idx_papers_abstract_gin;")
        conn.commit()
```

Został użyty zaawansowany, wbudowany silnik wyszukiwania pełnotekstowego w PostgreSQL.
CREATE INDEX ... USING gin(to_tsvector(...)): To nie jest zwykły indeks. to_tsvector('english', abstract) to funkcja, która konwertuje tekst abstraktu na specjalny format: dzieli go na słowa, usuwa popularne słowa (tzw. "stop words" jak 'the', 'a', 'is'), a słowa bazowe sprowadza do ich rdzenia (np. 'running' i 'ran' stają się 'run'). Indeks GIN jest zoptymalizowany do bardzo szybkiego przeszukiwania tej struktury.
@@ to_tsquery(...): To nie jest zwykłe porównanie tekstu. to_tsquery('english', ...) konwertuje szukane słowo na ten sam specjalny format. Operator @@ oznacza "pasuje do" i używa stworzonego wcześniej indeksu GIN do błyskawicznego znalezienia pasujących dokumentów.

```
papers.create_index([('abstract', 'text')])
    start = time.time()
    list(papers.find({'$text': {'$search':
config.RARE_WORD_SEARCH_TERM}}).limit(100))
    results.append({'database': 'MongoDB', 'operation':
'READ_FULL_TEXT_SEARCH_WITH_INDEX', 'time_seconds': time.time() - start,
'records_processed': 100})
    papers.drop_index('abstract_text')
```

create_index([('abstract', 'text')]): Tworzy specjalny indeks tekstowy na polu abstract. {'$text': {'$search': ...}}: To specjalny operator, który może być użyty tylko wtedy, gdy istnieje indeks tekstowy. Mongodb używa tego indeksu do bardzo szybkiego znalezienia dokumentów zawierających dane słowo.



Obie bazy danych pokazują duży wzrost wydajności w porównaniu do wersji bez indeksu. To kluczowy dowód na niezbędność stosowania indeksów przy tego typu zapytaniach. Redis nie posiada takiej funkcjonalności.

Operacja 6: Agregacja Danych (READ_AGGREGATE_COUNT)

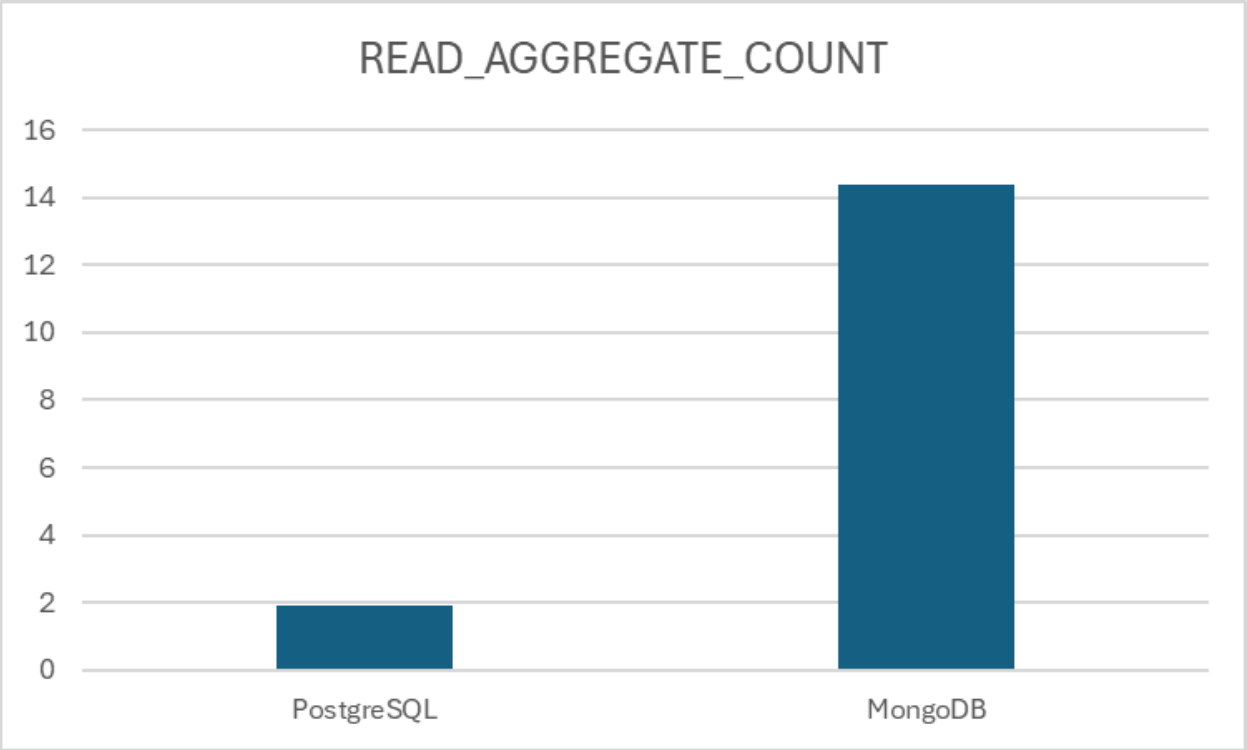
Test ten mierzy czas potrzebny na policzenie wszystkich unikalnych autorów w bazie.

```
list(papers.aggregate([
        {'$unwind': '$authors_parsed'},
        {'$group': {'_id': '$authors_parsed'}},
        {'$count': 'unique_authors'}
    ]))
```

{'$unwind': '$authors_parsed'}: To kluczowa i kosztowna operacja. Jeśli jeden dokument ma tablicę authors_parsed z 3 autorami, $unwind tworzy w pamięci trzy osobne kopie tego dokumentu, każda z jednym autorem. Z miliona dokumentów tworzy w ten sposób tymczasową, wielomilionową strukturę.

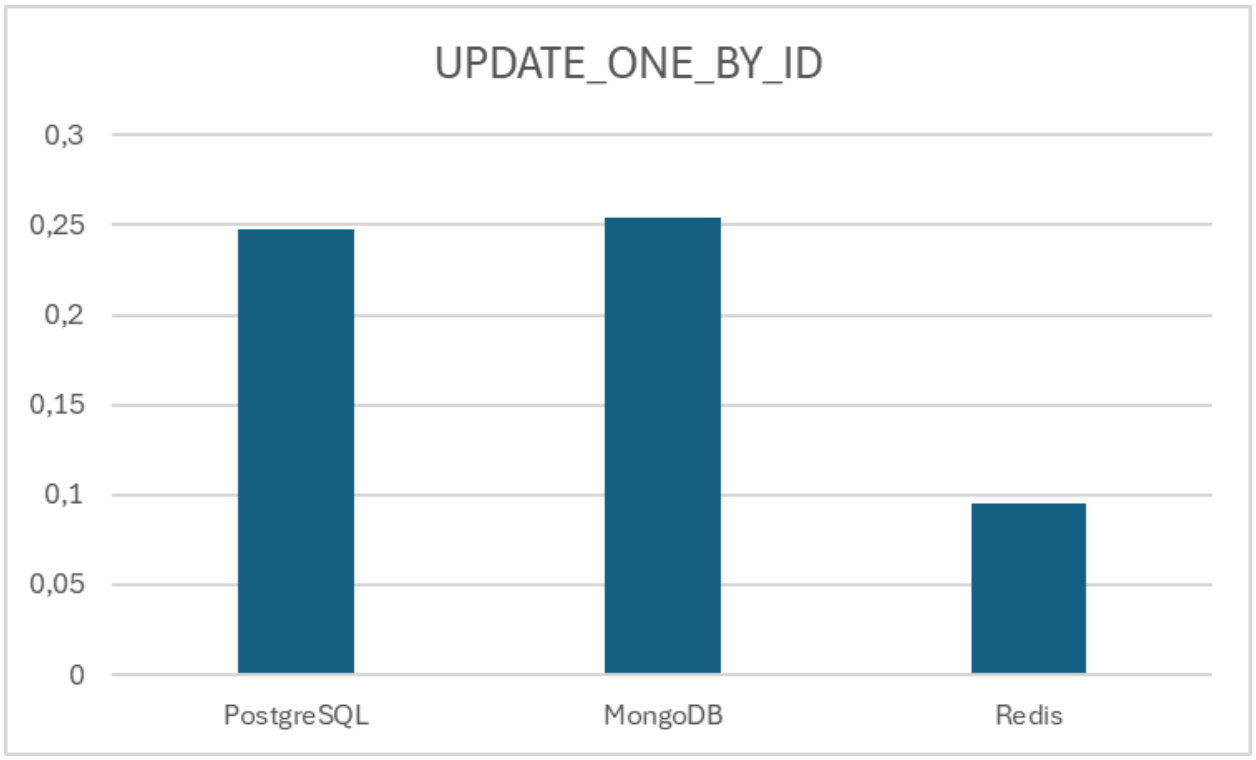
{'$group': {'_id': '$authors_parsed'}}: Grupuje te "rozpakowane" dokumenty po autorze. W efekcie dla każdego unikalnego autora powstaje jedna grupa.

{'$count': 'unique_authors'}: Zlicza, ile unikalnych grup (czyli autorów) powstało w poprzednim kroku.



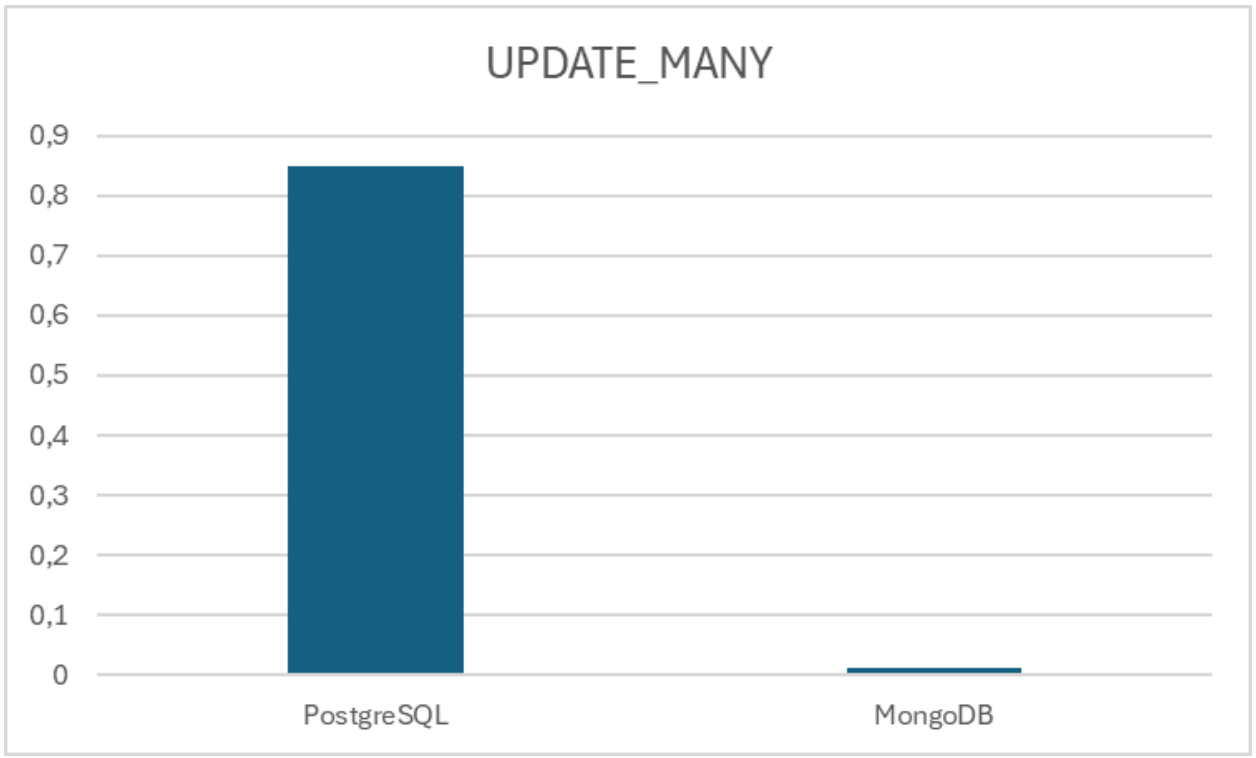
PostgreSQL jest tutaj absolutnym zwycięzcą. Jego silnik jest zoptymalizowany pod kątem operacji analitycznych na ustrukturyzowanych danych. Agregacja w MongoDB wymaga kosztownej operacji `$unwind` na zagnieżdżonej tablicy autorów, co okazało się bardzo wolne na tym zbiorze danych. Redis nie posiada takiej funkcjonalności.

Operacja 7: Aktualizacja Jednego Rekordu (UPDATE_ONE_BY_ID)
Test ten mierzy czas wykonania 500 operacji aktualizacji jednego, tego samego rekordu.



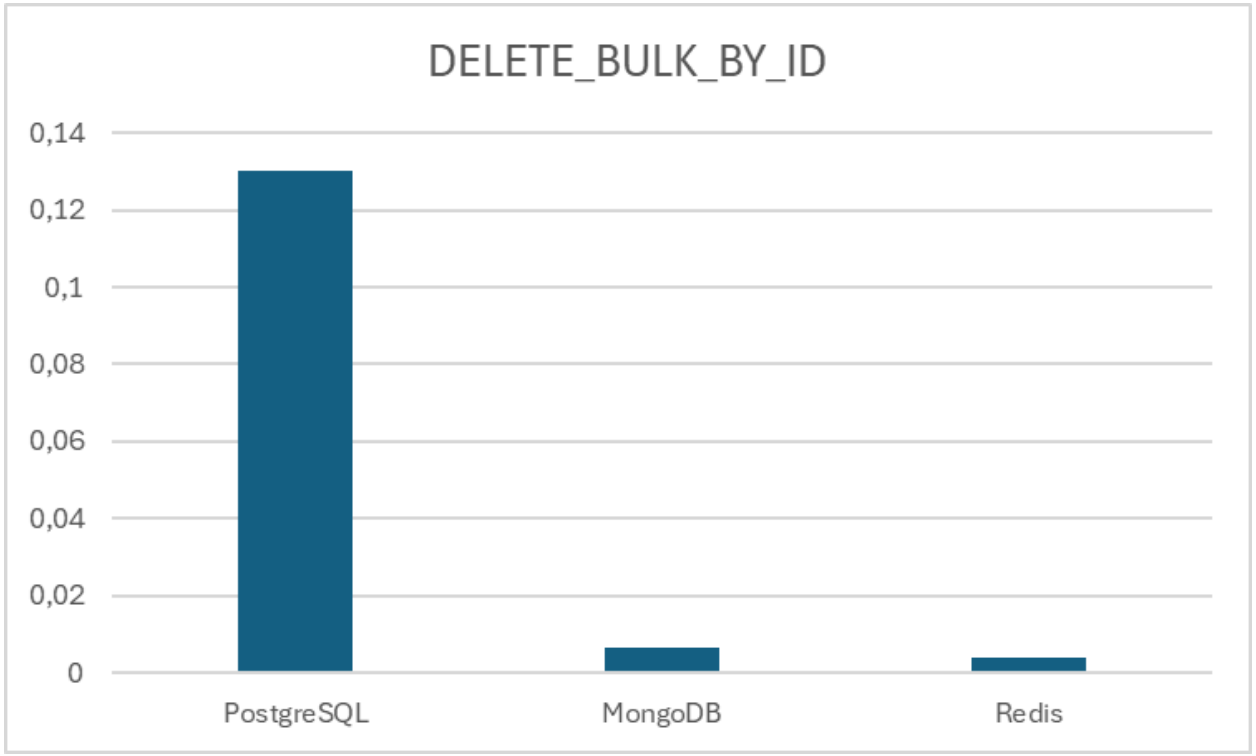
Ponownie Redis pokazuje swoją siłę w prostych operacjach zapisu w pamięci. PostgreSQL i MongoDB mają porównywalne, dobre wyniki.

Operacja 8: Aktualizacja Wielu Rekordów (UPDATE_MANY)
Test ten mierzy czas potrzebny na zaktualizowanie 500 rekordów za pomocą jednego polecenia.



MongoDB jest tu dużo szybsze. Operacja `update_many` na dokumentach, które nie mają złożonych powiązań, jest bardzo wydajna. W PostgreSQL aktualizacja wielu wierszy wiąże się z większym narzutem transakcyjnym i mechanizmami spójności. Redis nie posiada takiej funkcjonalności.

Operacja 9: Usuwanie Masowe (DELETE_BULK_BY_ID)
Test ten mierzy czas potrzebny na usunięcie 500 rekordów, które zostały stworzone na początku testów.



Wyniki są podobne do zapisu masowego. Redis i MongoDB są szybkie w usuwaniu prostych danych. PostgreSQL musi zarządzać integralnością relacyjną, co wiąże się z większym narzutem.

4. Ograniczenia Funkcjonalne Redis

W czterech z przeprowadzonych testów Redis nie mógł zostać uwzględniony. Wynika to z jego fundamentalnej filozofii - Redis nie jest "bogatą w funkcje" bazą danych, ale szybkim zestawem podstawowych "klocków".

* Wyszukiwanie Pełnotekstowe i Agregacje: Standardowy Redis nie oferuje takich mechanizmów. W realnych projektach takie funkcjonalności osiąga się poprzez:
    1. Odpowiednie modelowanie danych (np. utrzymywanie osobnych SET`ów do zliczania unikalnych elementów).
    2. Użycie Modułów Redis, takich jak **RediSearch**, które rozszerzają bazę o zaawansowane możliwości wyszukiwania.
* JOIN-y i UPDATE_MANY: Te operacje są sprzeczne z prostym modelem klucz-wartość. Logikę łączenia danych lub masowej aktualizacji po stronie klienta (w aplikacji) lub za pomocą skryptów Lua wykonywanych po stronie serwera.

5. Podsumowanie i Wnioski

Projekt jednoznacznie pokazał, że nie istnieje jedna "najlepsza" baza danych. Wybór zależy ściśle od wymagań konkretnego przypadku użycia.

* PostgreSQL okazał się królem spójności i zaawansowanych zapytań analitycznych (agregacje, JOIN-y). Jego wydajność w prostych operacjach zapisu jest niższa z powodu narzutu na utrzymanie integralności danych, co jest jego największą zaletą w systemach transakcyjnych.
* MongoDB stanowi doskonały, zbalansowany wybór dla danych semi-ustrukturyzowanych. Błyszczy w operacjach zapisu i masowej aktualizacji, a jego elastyczność drastycznie upraszcza proces deweloperski. Okazał się jednak wolniejszy w teście agregacji.
* Redis jest bezkonkurencyjny pod względem szybkości dla prostych operacji odczytu i zapisu po kluczu. Jego model danych wymaga jednak od programisty więcej pracy przy implementacji złożonej logiki, takiej jak indeksy wtórne czy agregacje.