Zaawansowane bazy danych

PROJEKT

Autorzy:
Damian Gnieciak, 259065
Szymon Leja, 259047

Prowadzący:
Dr inż. Maciej Nikodem

Grupa:
Nr. 4

Termin zajęć:
Poniedziałek NP $17^{05} - 18^{45}$

Wrocław, 24 kwietnia 2024

Politechnika Wrocławska,
Wydział Informatyki i Telekomunikacji

Spis treści

1 Temat projektu

Celem projektu jest porównanie zapytań SQL pisanych ręcznie, z tymi generowanymi przy użyciu popularnych narzędzie ORM.

1.1 Sposób porównania

Aby dokonać obiektywne wyniki, zarówno zapytania napisane ręczie oraz zapytania wygenerowane przez ORM zostały odpalane bezpośrednio na silniku bazy danych, a następnie porównywane zostały następujące kryteria:

- Czasy wykonania [ms]
- Ilość odniesień do pamięci cache
- Ilosc odczytań z dysku
- Składnie zapytania

1.2 Scenariusze testowe

Wyżej wspomniane scenariusze testowe, które umożliwią wyszukiwanie w bazie danych na podstawie różnych kryteriów:

- **GetFlatsByQuery** Wyszukiwanie wolnych pokoi w danym terminie z wybranymi parametrami pokoju, takimi jak termin dostępności, cena, udogodnienia, ilość miejsc.
- **GetUserReservations** Sprawdzanie rezerwacji danego użytkownika.
- **GetUsersWithFlatNumberInCity** Wyszukiwanie użytkowników, którzy mają przypisane budynki z daną ilością pokoi w wybranym mieście.
- **GetFlatsNearLocation** Wyszukiwanie mieszkań znajdujących się w danej odległości od podanej lokalizacji.
- **GetFlatsByCapacityAndRevenue** Wygenerowanie statystyk o najchętniej wynajmowanych mieszkaniach z podziałem na miasta i udogodnienia.
- **GetUserFlatWithHighestIncome** Sprawdzenie które mieszkanie danego właściciela wygenerowało największe zyski w danym przedziale czasu.
- **GetMostPopularFlatStats** Sprawdzenie ilu pokojowe mieszkania są najczęściej wynajmowane oraz jakie przynoszą zyski.

2 Implementacja

Początkowo projekt zakładał porównanie dwóch różnych narzędzie ORM, jednego używanego w języku C# o nazwie *Entity Framework Core*, a drugiego w języku Java o nazwie Spring Boot. Jednak finalnie zrezygnowaliśmy z drugiego narzędzia, ponieważ bardziej skomplikowane zapytania w Java Spring Boot musiały być pisane w języku *JPQL*, który jest bardzo mało abstrakcyjny i bliźniaczo podobny do języka SQL, a w wygenerowanych zapytaniach różnice były kosmetyczne. Po wstępnych testach uznaliśmy że porównywanie zapytań wygenerowanych w taki właśnie sposób jest bezcelowe.

```dotnet
var query = context.Reservations
    .Where(r => r.Flat.Building.Address.City == city)
    .SelectMany(r => r.Flat.Facilities)
    .GroupBy(f => f.Name)
    .Select(g => new GetMostPopularFlatStatDto
    {
        Amenity = g.Key,
        RentalCount = g.Count()
    })
    .OrderByDescending(f => f.RentalCount);
```

(a) Zapytanie w .NET EF Core

```java
@Query("SELECT a.city, fac.name as amenity, COUNT(r) as rentalCount " +
        "FROM Reservation r " +
        "JOIN r.flat f " +
        "JOIN f.building b " +
        "JOIN b.address a " +
        "JOIN f.flatFacilities ff " +
        "JOIN ff.facility fac " +
        "GROUP BY a.city, fac.name " +
        "ORDER BY COUNT(r) DESC")
List<Object[]> findPopularFlatsStatistics();
```

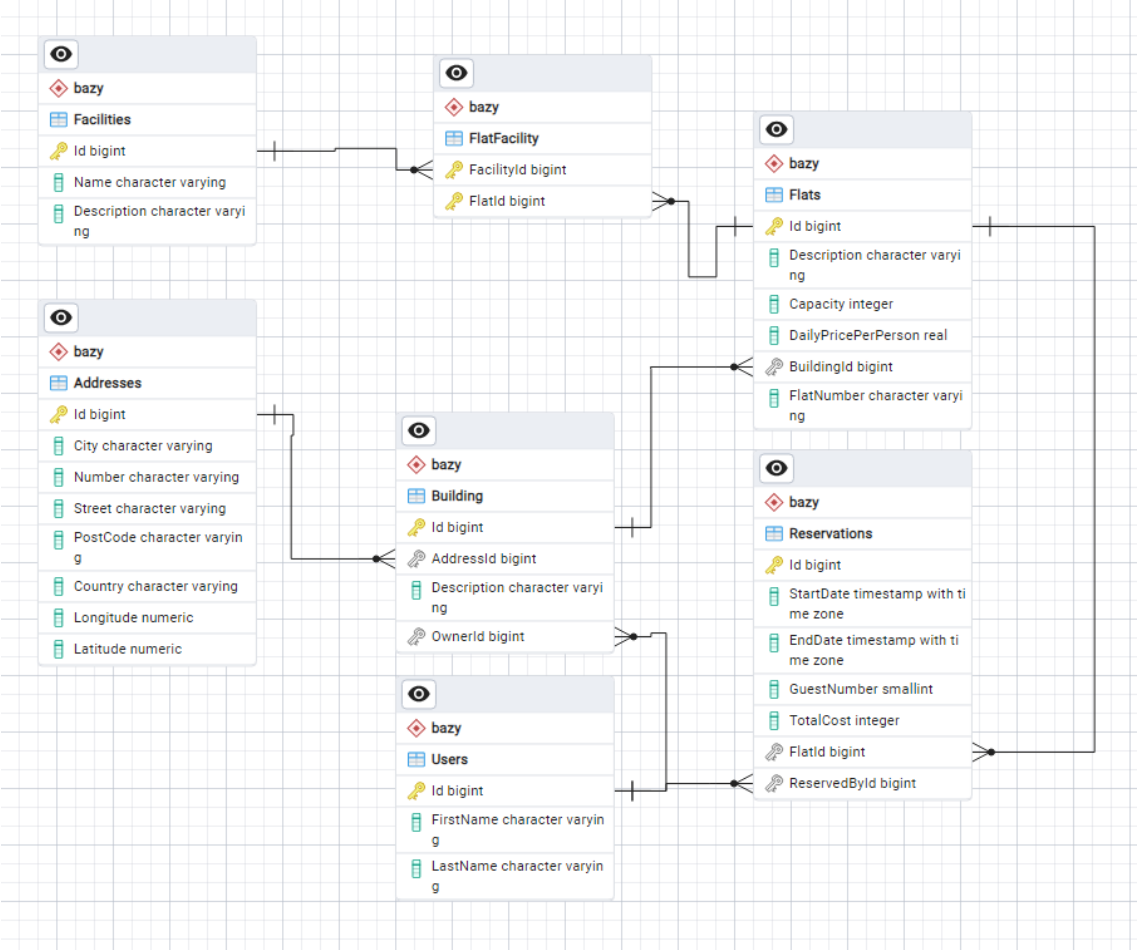
(b) Zapytanie w Java Spring Boot

2.1 Technologie

Testowanym narzędziam był Entity Framework Core w wersji 8.0.2, działający w .NET 8 oraz PostgreSQL w wersji 15.

2.2 Schemat bazy

Analizowanym schematem ma być baza danych systemu służącego do rezerwowania oraz wynajmowania mieszkań.



Rysunek 2: Schemat bazy danych

3 Testowane zapytania

Poniżej znajdują się zapytania dla wszystkich scenariuszy testowych, zarówno wersje pisane ręcznie oraz wygenerowane automatycznie.

3.1 GetUsersWithFlatNumberInCity

```sql
SELECT u.id AS Id, u.first_name AS FirstName, u.last_name AS LastName
FROM rental.users u
JOIN rental.building b ON u.id = b.owner_id
JOIN rental.flats f ON b.id = f.building_id
JOIN rental.addresses a ON b.address_id = a.id
WHERE a.city = @city
GROUP BY u.id
HAVING COUNT(f.id) = @flatNumber;
```

Listing 1: GetUsersWithFlatNumberInCity_raw

```sql
-- @__city_0='Langworthhaven'
-- @__flatNumber_1='5'
SELECT u.id AS "Id", u.first_name AS "FirstName", u.last_name AS "LastName"
FROM rental.users AS u
WHERE (
    SELECT count(*)::int
    FROM rental.building AS b
    INNER JOIN rental.addresses AS a ON b.address_id = a.id
    INNER JOIN rental.flats AS f ON b.id = f.building_id
    WHERE u.id = b.owner_id AND a.city = @__city_0) = @__flatNumber_1;
```

Listing 2: GetUsersWithFlatNumberInCity_generated

3.2 GetUserReservations

```sql
SELECT r.id AS Id, r.start_date AS StartDate, r.end_date AS EndDate, r.guest_number
     AS GuestNumber, r.total_cost AS TotalCost
            FROM rental.reservations r
            WHERE r.reserved_by_id = @userId;
```

Listing 3: GetUserReservations_raw

```sql
-- @__userId_0='1'
SELECT r.id AS "Id", r.start_date AS "StartDate", r.end_date AS "EndDate", r.
    guest_number AS "GuestNumber", r.total_cost AS "TotalCost"
FROM rental.reservations AS r
WHERE r.reserved_by_id = @__userId_0;
```

Listing 4: GetUserReservations_generated

3.3 GetUserFlatWithHighestIncome

```sql
SELECT f.id AS Id, SUM(r.total_cost) AS TotalRevenue
FROM rental.flats f
JOIN rental.reservations r ON f.id = r.flat_id
WHERE r.start_date >= @startDate AND r.end_date <= @endDate
AND f.building_id IN (SELECT id FROM rental.building WHERE owner_id = @userId)
GROUP BY f.id
ORDER BY TotalRevenue DESC
LIMIT 1;
```

Listing 5: GetUserFlatWithHighestIncome_raw

```
-- @__ToUniversalTime_1='2023-12-31T23:00:00.0000000Z' (DbType = DateTime)
-- @__ToUniversalTime_2='2024-05-30T22:00:00.0000000Z' (DbType = DateTime)
-- @__userId_0='4'
SELECT f.id AS "Id", (
    SELECT COALESCE(sum(r0.total_cost), 0)
    FROM rental.reservations AS r0
    WHERE f.id = r0.flat_id AND r0.start_date >= @__ToUniversalTime_1 AND r0.
    end_date <= @__ToUniversalTime_2) AS "TotalRevenue"
FROM rental.flats AS f
INNER JOIN rental.building AS b ON f.building_id = b.id
WHERE b.owner_id = @__userId_0
ORDER BY (
    SELECT COALESCE(sum(r.total_cost), 0)
    FROM rental.reservations AS r
    WHERE f.id = r.flat_id AND r.start_date >= @__ToUniversalTime_1 AND r.end_date
    <= @__ToUniversalTime_2) DESC;
```

Listing 6: GetUserFlatWithHighestIncome_generated

3.4 GetMostPopularFlatStats

```
SELECT fac.name AS Amenity, COUNT(*) AS RentalCount
FROM rental.reservations r
JOIN rental.flats f ON r.flat_id = f.id
JOIN rental.building b ON f.building_id = b.id
JOIN rental.addresses a ON b.address_id = a.id
JOIN rental.flat_facility ff ON f.id = ff.flat_id
JOIN rental.facilities fac ON ff.facility_id = fac.id
WHERE a.city = @city
GROUP BY fac.name
ORDER BY RentalCount DESC;
```

Listing 7: GetMostPopularFlatStats_raw

```
-- @__city_0='Langworthhaven'
SELECT t.name AS "Amenity", count(*)::int AS "RentalCount"
FROM rental.reservations AS r
INNER JOIN rental.flats AS f ON r.flat_id = f.id
INNER JOIN rental.building AS b ON f.building_id = b.id
INNER JOIN rental.addresses AS a ON b.address_id = a.id
INNER JOIN (
    SELECT f1.name, f0.flat_id
    FROM rental.flat_facility AS f0
    INNER JOIN rental.facilities AS f1 ON f0.facility_id = f1.id
) AS t ON f.id = t.flat_id
WHERE a.city = @__city_0
GROUP BY t.name
ORDER BY count(*)::int DESC;
```

Listing 8: GetMostPopularFlatStats_generated

3.5 GetFlatsNearLocation

```
SELECT f.id AS Id, f.description AS Description, f.daily_price_per_person AS
    DailyPricePerPerson, f.capacity AS Capacity, f.building_id AS BuildingId, f.
    flat_number AS FlatNumber
FROM rental.flats f
JOIN rental.building b ON f.building_id = b.id
JOIN rental.addresses a ON b.address_id = a.id
WHERE (
    6371 * acos (
        cos(radians(@latitude)) * cos(radians(a.latitude)) * cos(radians(a.
    longitude - @longitude)) +
        sin(radians(@latitude)) * sin(radians(a.latitude))
    )
) < @radius;
```

Listing 9: GetFlatsNearLocation_raw

```
-- @__Cos_0='0.6156614753256583'
-- @__p_1='0.3839724354387525'
-- @__Sin_2='0.788010753606722'
-- @__radius_3='0.1'
SELECT f.id AS "Id", f.description AS "Description", f.daily_price_per_person AS "
    DailyPricePerPerson", f.capacity AS "Capacity", f.building_id AS "BuildingId",
    f.flat_number AS "FlatNumber"
FROM rental.flats AS f
INNER JOIN rental.building AS b ON f.building_id = b.id
INNER JOIN rental.addresses AS a ON b.address_id = a.id
WHERE 6371.0 * acos((@__Cos_0 * cos((3.1415926535897931 * a.latitude::double
    precision) / 180.0)) * cos((3.1415926535897931 * a.longitude::double precision)
     / 180.0 - @__p_1) + @__Sin_2 * sin((3.1415926535897931 * a.latitude::double
    precision) / 180.0)) < @__radius_3;
```

Listing 10: GetFlatsNearLocation_generated

3.6 GetFlatsByQuery

```
SELECT f.id AS Id, f.description AS Description, f.daily_price_per_person AS
    DailyPricePerPerson, f.capacity AS Capacity, b.id AS BuildingId, f.flat_number
    AS FlatNumber
FROM rental.flats f
JOIN rental.building b ON f.building_id = b.id
WHERE f.daily_price_per_person <= @maxPrice
AND f.capacity >= @minCapacity
AND NOT EXISTS (
    SELECT 1 FROM rental.reservations r
    WHERE r.flat_id = f.id
    AND (r.start_date, r.end_date) OVERLAPS (@startDate, @endDate)
)
ORDER BY f.id;
```

Listing 11: GetFlatsByQuery_raw

```
-- @__maxPrice_0='10'
-- @__minCapacity_1='9' (DbType = Int16)
-- @__startDateUtc_2='2024-01-01T00:00:00.0000000Z' (DbType = DateTime)
-- @__endDateUtc_3='2024-01-31T00:00:00.0000000Z' (DbType = DateTime)
SELECT f.id AS "Id", f.description AS "Description", f.daily_price_per_person AS "
    DailyPricePerPerson", f.capacity AS "Capacity", f.building_id AS "BuildingId",
    f.flat_number AS "FlatNumber"
FROM rental.flats AS f
WHERE f.daily_price_per_person <= @__maxPrice_0 AND f.capacity >= @__minCapacity_1
    AND NOT EXISTS (
    SELECT 1
    FROM rental.reservations AS r
    WHERE f.id = r.flat_id AND ((r.start_date <= @__startDateUtc_2 AND r.end_date
    >= @__endDateUtc_3) OR (r.start_date >= @__startDateUtc_2 AND r.start_date <
    @__endDateUtc_3) OR (r.end_date > @__startDateUtc_2 AND r.end_date <=
    @__endDateUtc_3)))
ORDER BY f.id;
```

Listing 12: GetFlatsByQuery_generated

3.7 GetFlatsByCapacityAndRevenue

```
SELECT f.capacity AS Capacity, COUNT(r.id) AS NumberOfRentals, SUM(r.total_cost) AS
     TotalRevenue
FROM rental.flats f
JOIN rental.reservations r ON f.id = r.flat_id
GROUP BY f.capacity
ORDER BY NumberOfRentals DESC, TotalRevenue DESC;
```

Listing 13: GetFlatsByCapacityAndRevenue_raw

```sql
SELECT f.capacity AS "Capacity", (
    SELECT count(*)::int
    FROM rental.flats AS f1
    INNER JOIN rental.reservations AS r1 ON f1.id = r1.flat_id
    WHERE f.capacity = f1.capacity)::bigint AS "NumberOfRentals", COALESCE(sum((
    SELECT COALESCE(sum(r2.total_cost), 0)
    FROM rental.reservations AS r2
    WHERE f.id = r2.flat_id)), 0) AS "TotalRevenue"
FROM rental.flats AS f
GROUP BY f.capacity
ORDER BY (
    SELECT count(*)::int
    FROM rental.flats AS f1
    INNER JOIN rental.reservations AS r1 ON f1.id = r1.flat_id
    WHERE f.capacity = f1.capacity)::bigint DESC, COALESCE(sum((
    SELECT COALESCE(sum(r2.total_cost), 0)
    FROM rental.reservations AS r2
    WHERE f.id = r2.flat_id)), 0) DESC;
```

Listing 14: GetFlatsByCapacityAndRevenue_generated

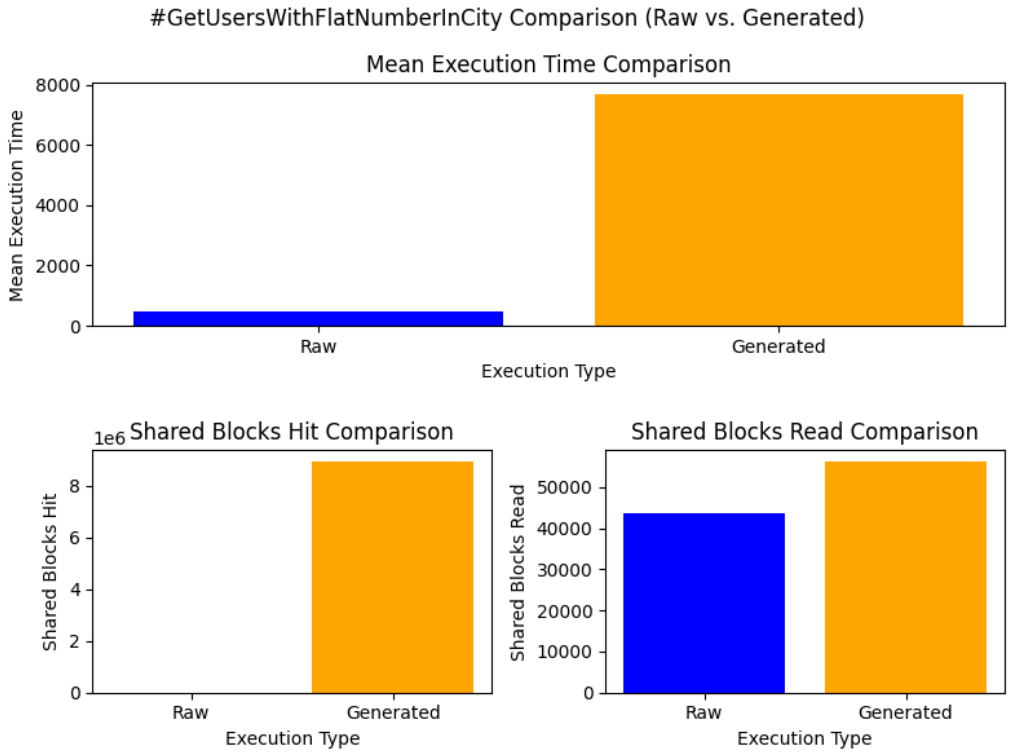
4 Wyniki

Poniżej znajdują się porównania dla wybranych scenariuszy testowych.

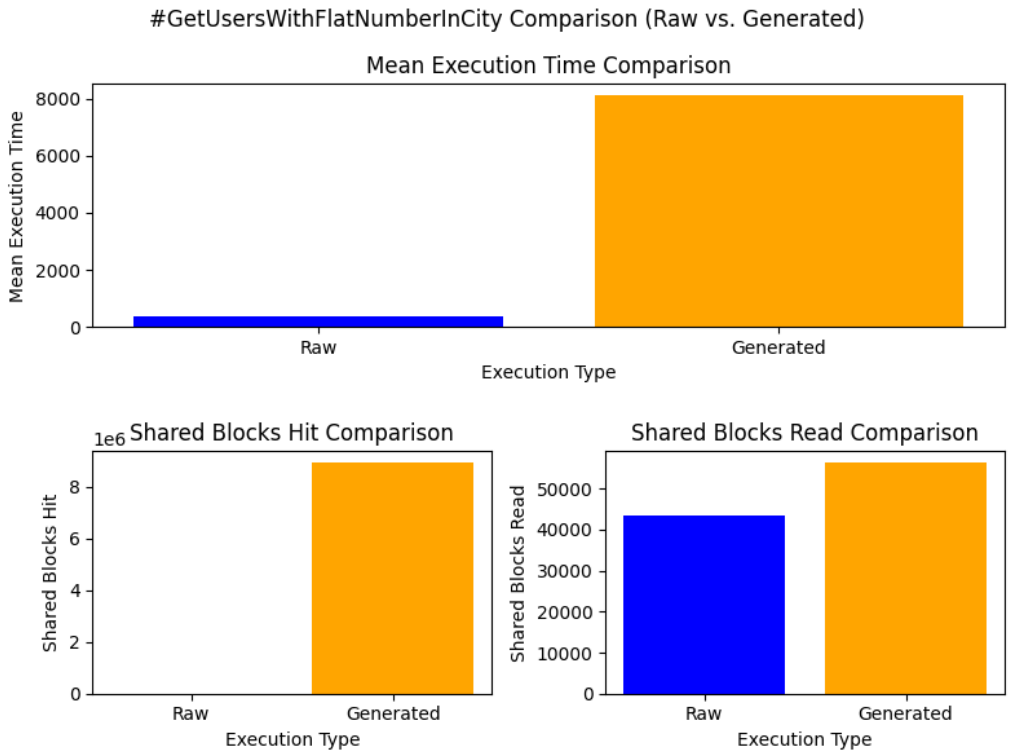
4.1 Statystyki zbierane przez silnik bazy danych

Silnik bazy danych PostgreSQL pozwala włączyć zbieranie dodatkowych statystyk do tabeli o nazwie *pg_stat_statements*. Poniżej znajdują się dane, które udało się pozyskać.

4.1.1 GetUsersWithFlatNumberInCity

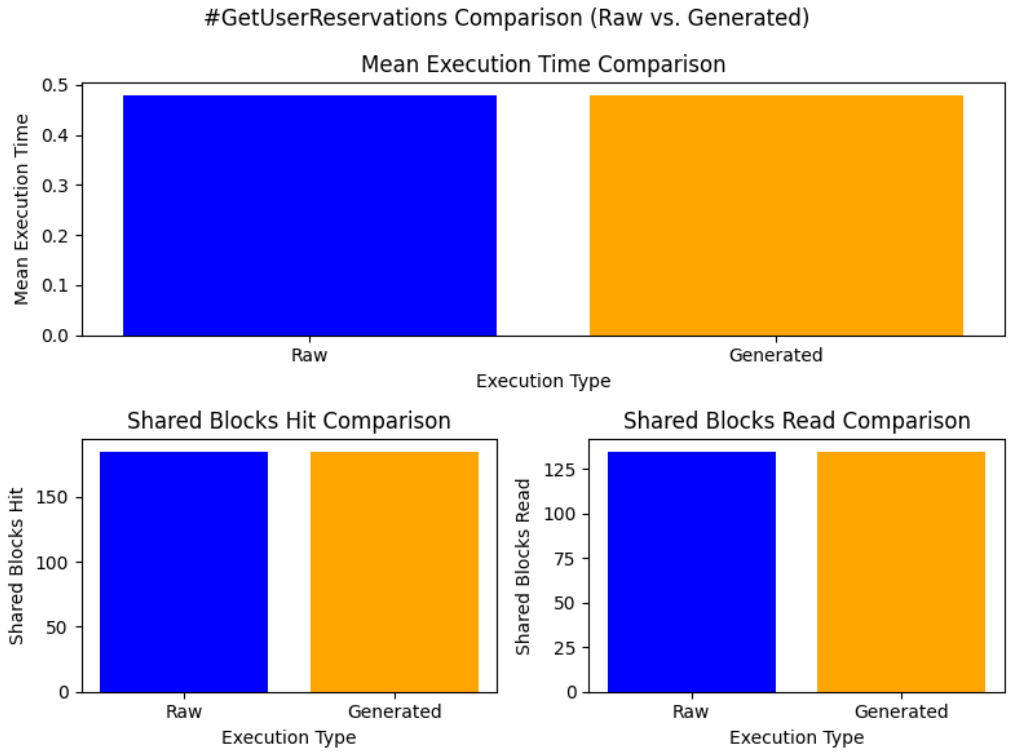


Rysunek 3: Porównanie dla zapytania na niepustym wyniku

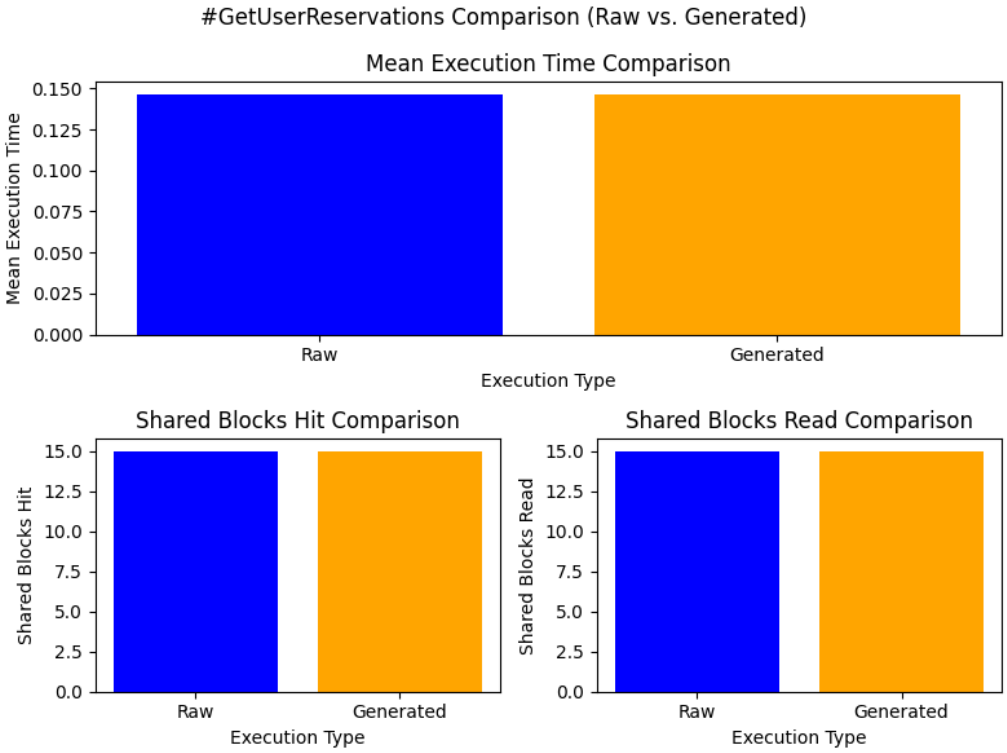


Rysunek 4: Porównanie dla zapytania na pustym wyniku

4.1.2 GetUsersReservations

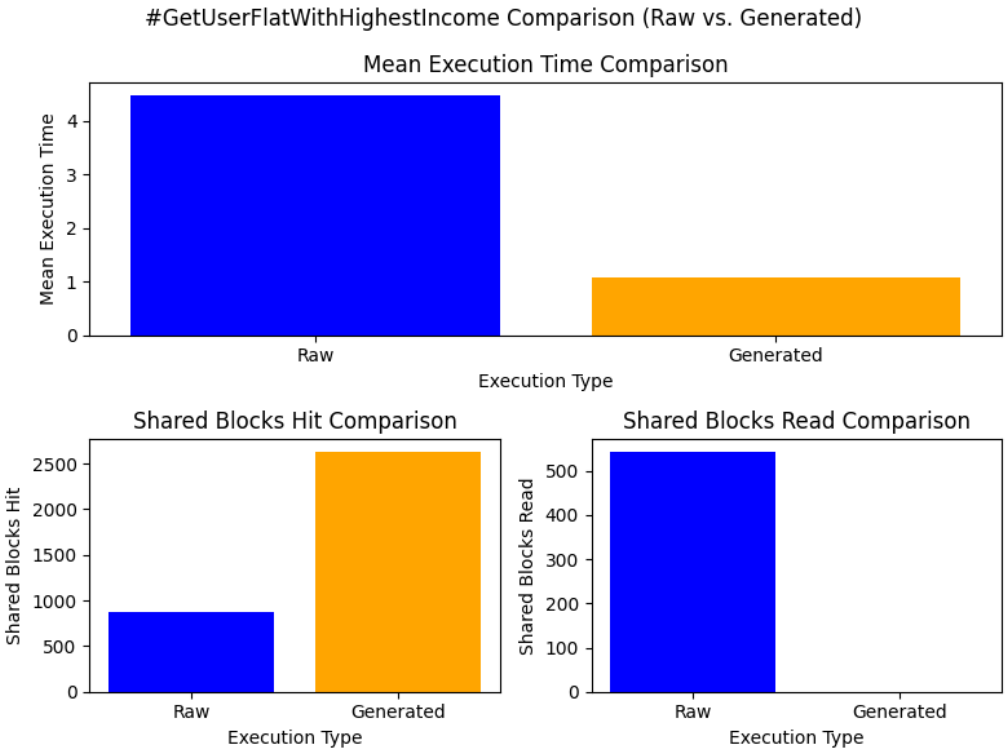


Rysunek 5: Porównanie dla zapytania na niepustym wyniku

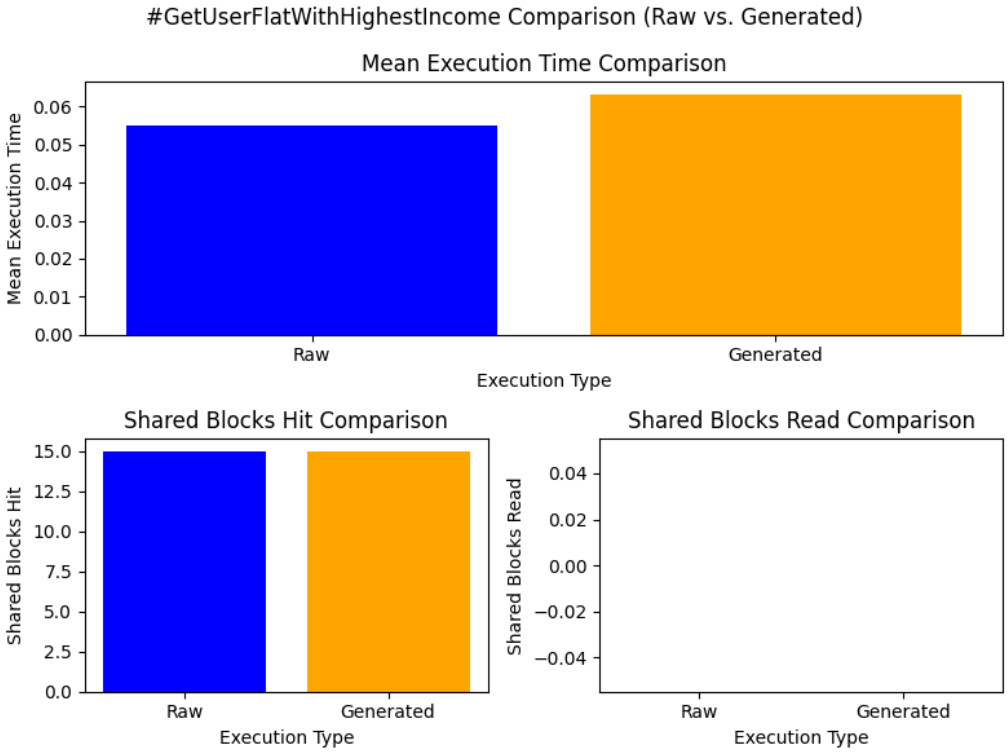


Rysunek 6: Porównanie dla zapytania na pustym wyniku

4.1.3 GetUserFlatWithHighestIncome

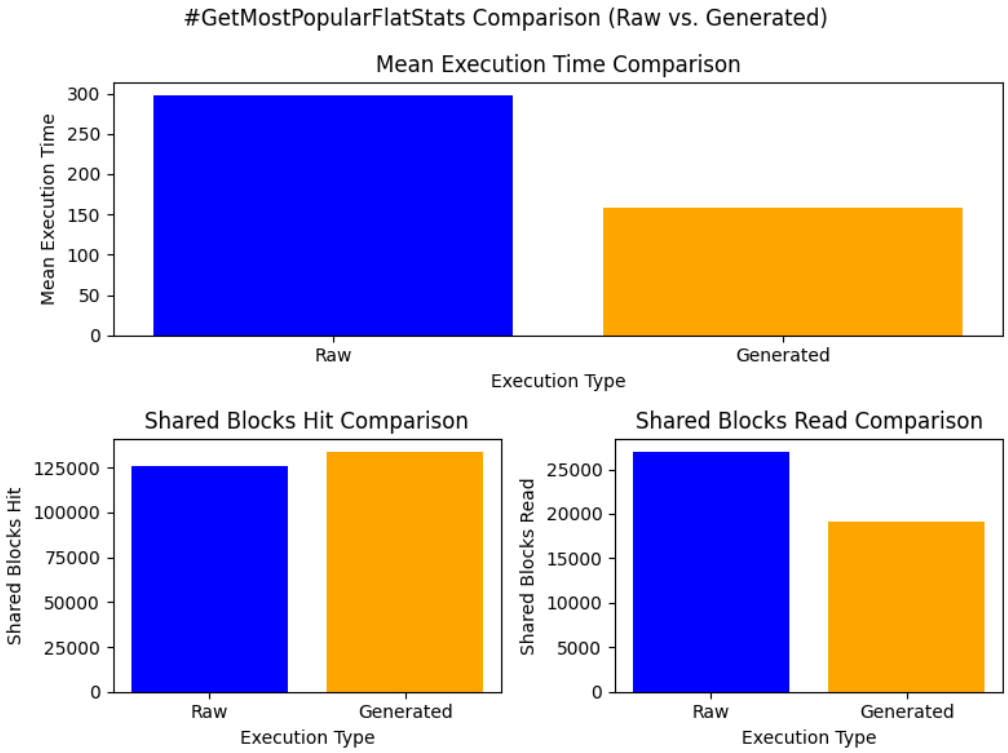


Rysunek 7: Porównanie dla zapytania na niepustym wyniku

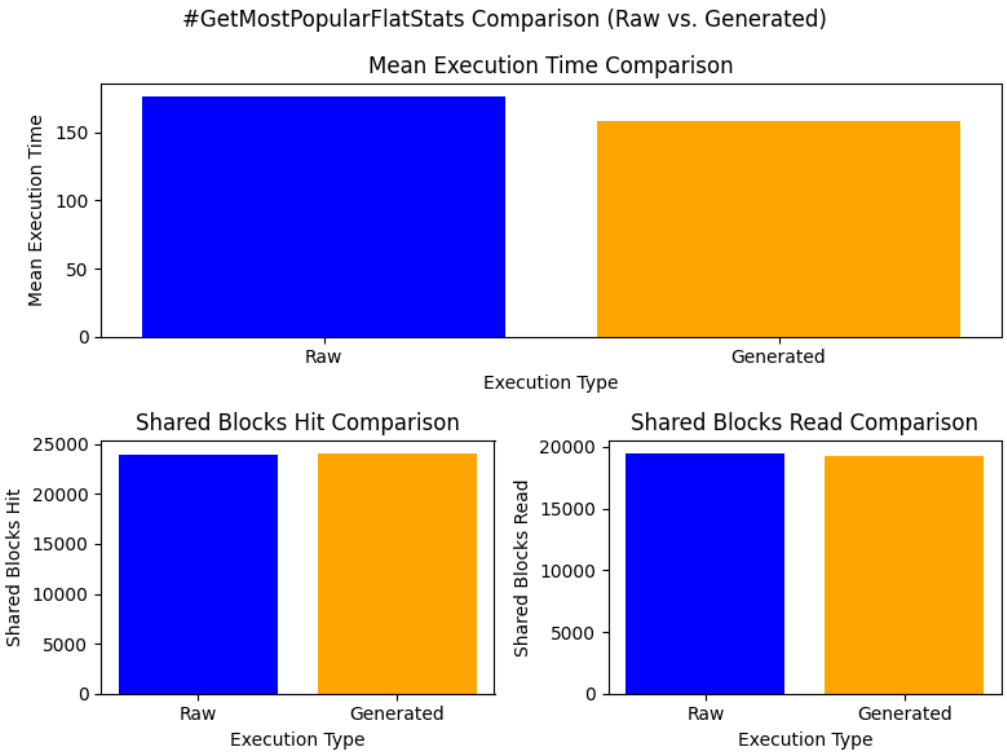


Rysunek 8: Porównanie dla zapytania na pustym wyniku

4.1.4 GetMostPopularFlatStats

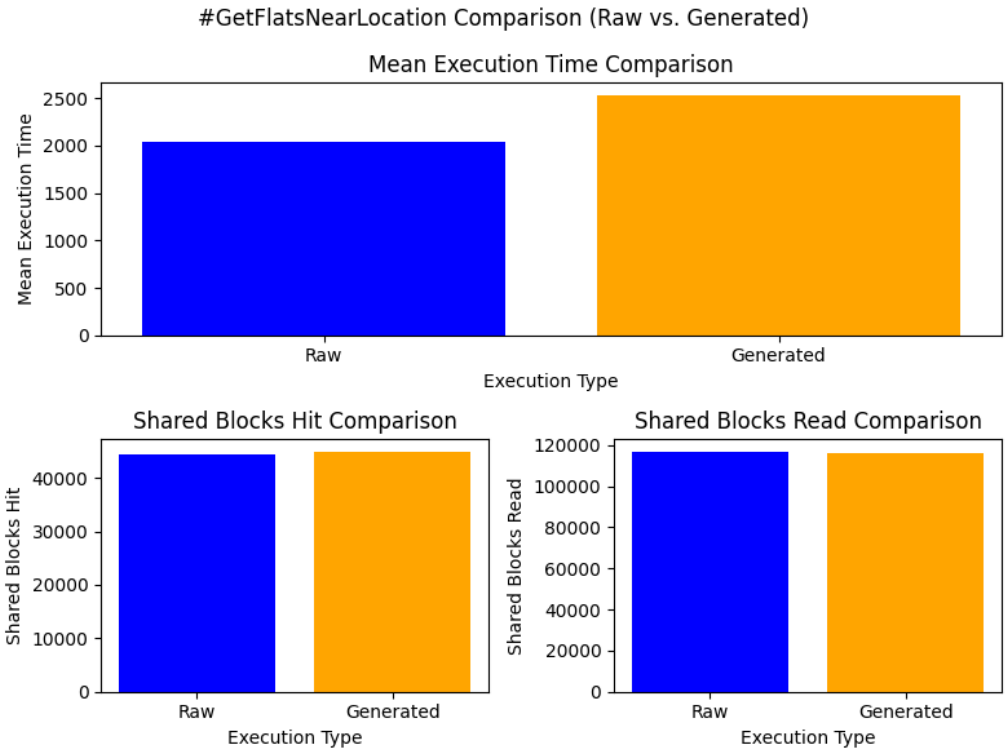


Rysunek 9: Porównanie dla zapytania na niepustym wyniku

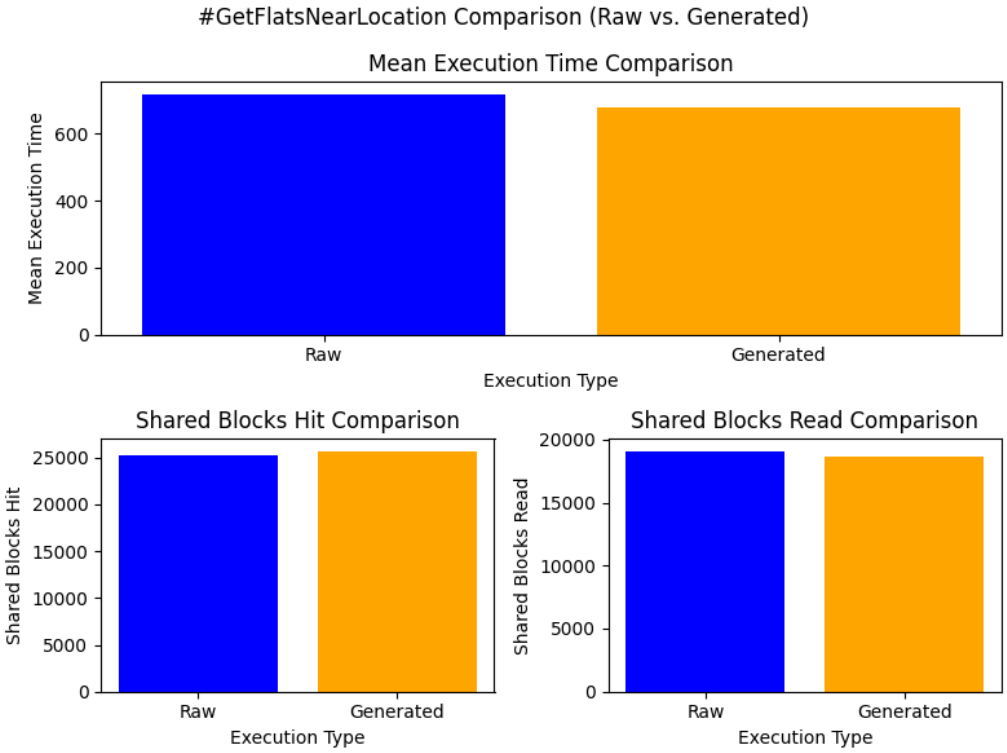


Rysunek 10: Porównanie dla zapytania na pustym wyniku

4.1.5 GetFlatsNearLocation

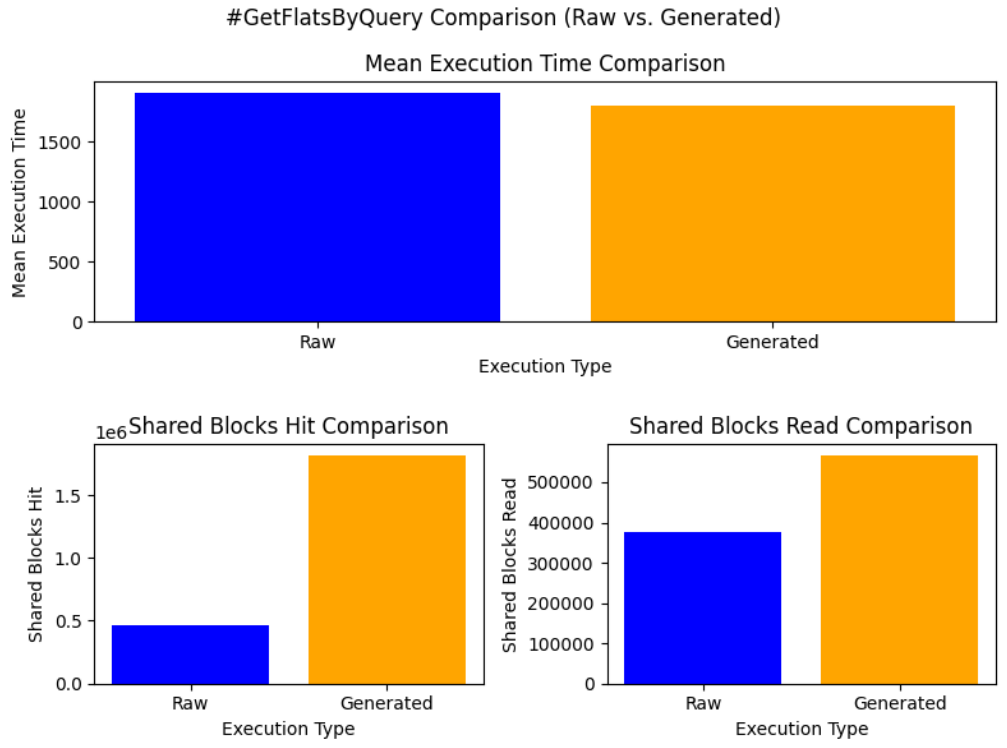


Rysunek 11: Porównanie dla zapytania na niepustym wyniku



Rysunek 12: Porównanie dla zapytania na pustym wyniku

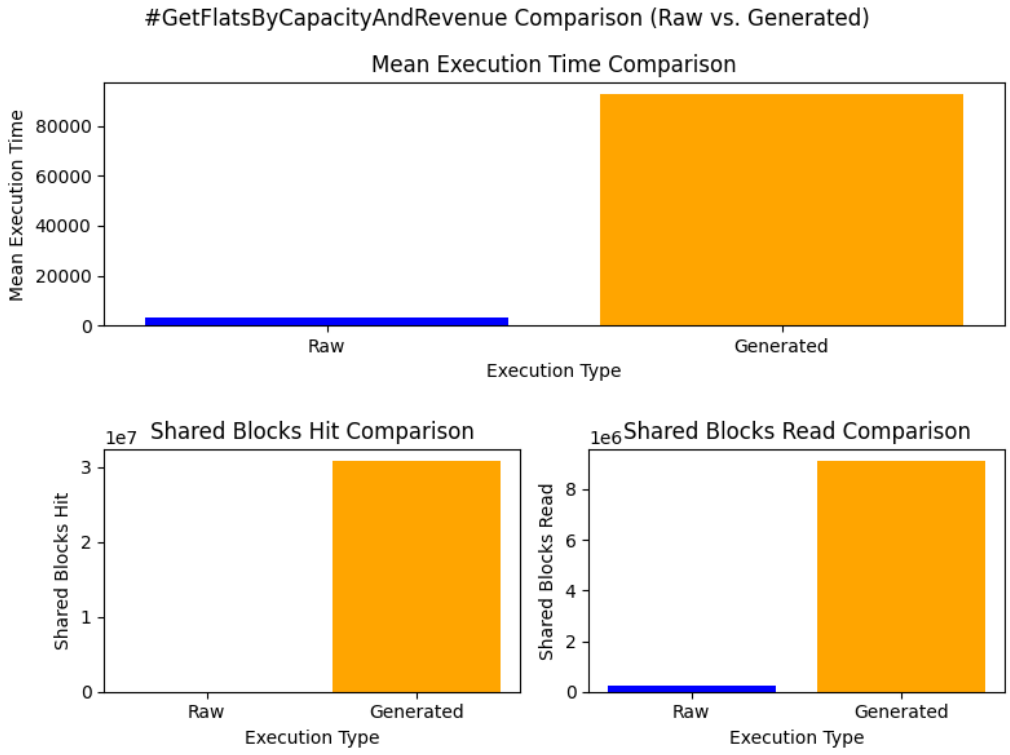
4.1.6 GetFlatsByQuery



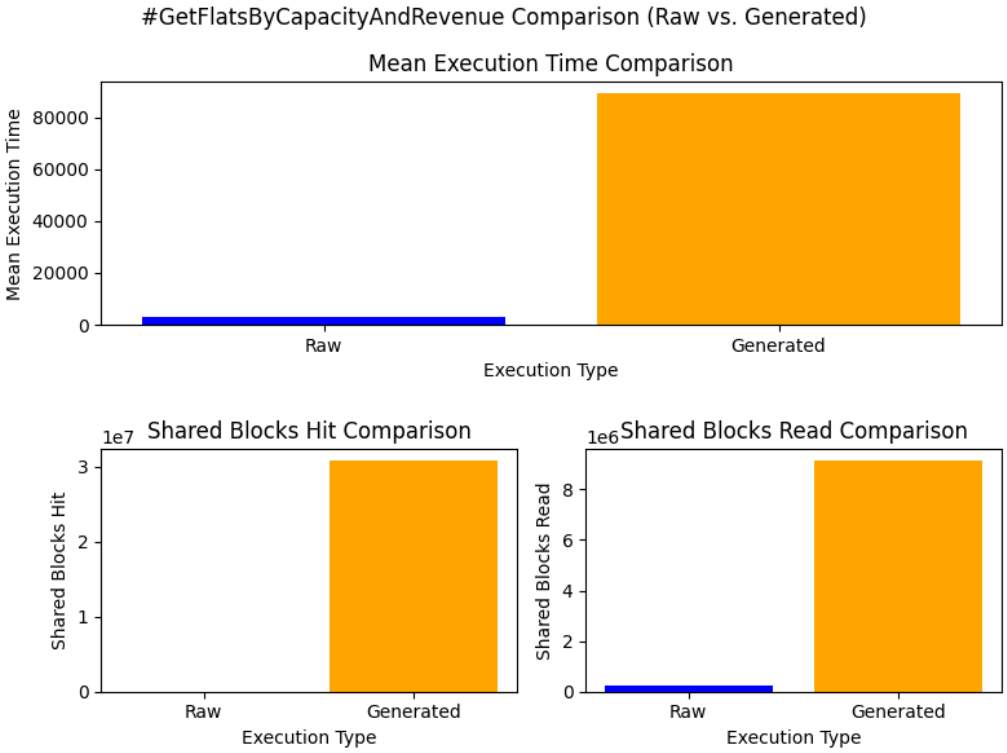
Rysunek 13: Porównanie dla zapytania na niepustym wyniku

Rysunek 14: Porównanie dla zapytania na pustym wyniku

4.1.7 GetFlatsByCapacityAndRevenue

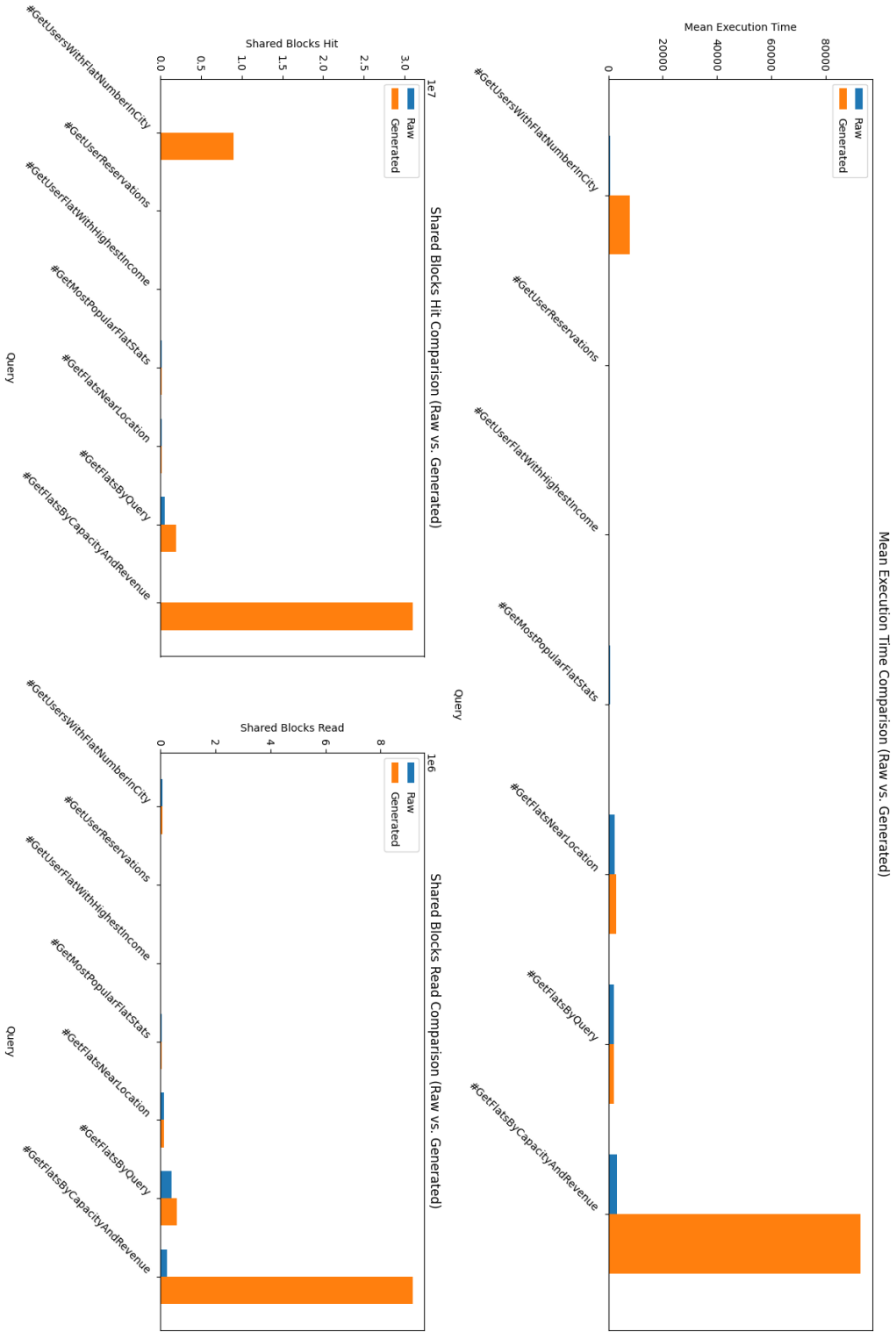


Rysunek 15: Porównanie dla zapytania na niepustym wyniku

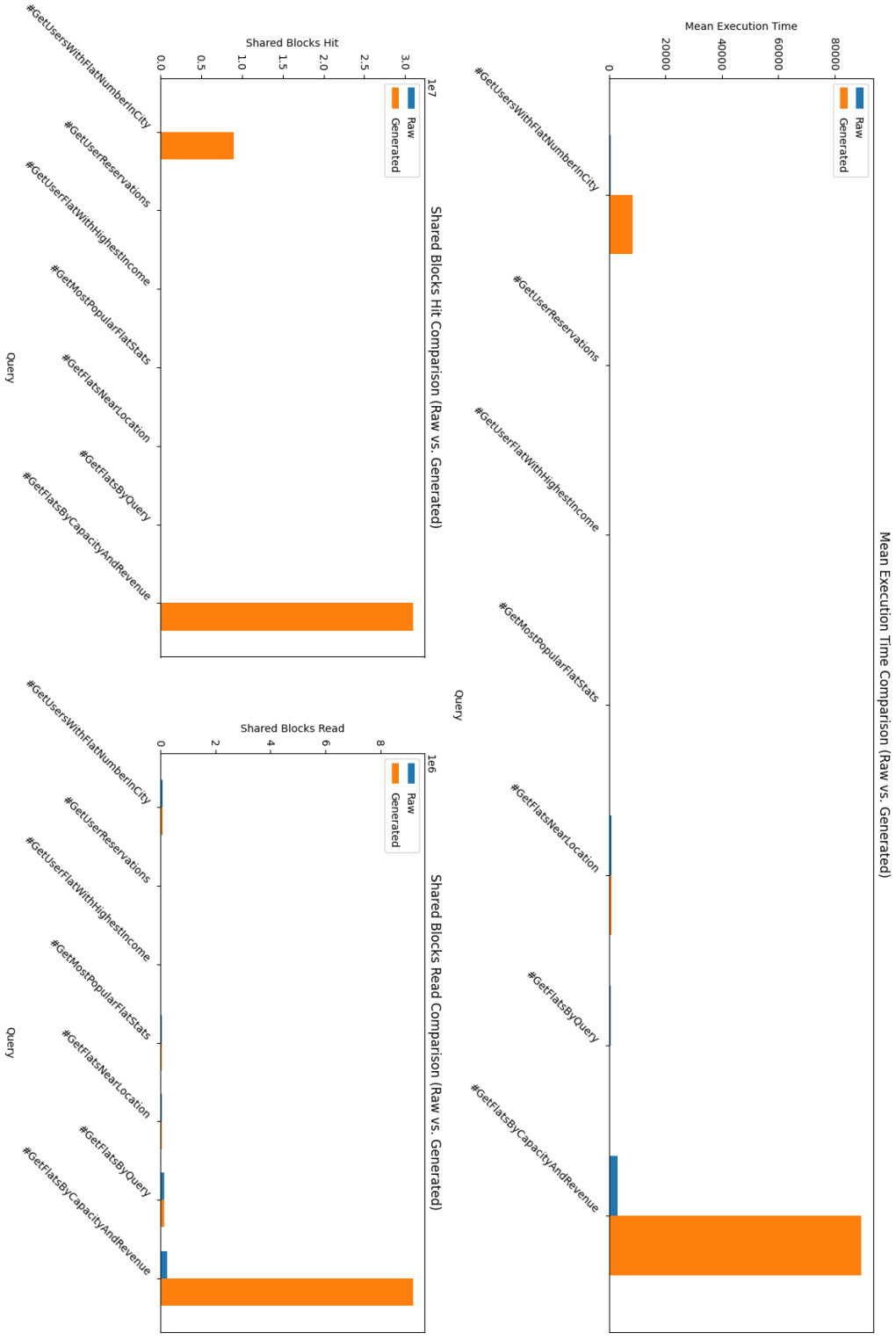


Rysunek 16: Porównanie dla zapytania na pustym wyniku

4.1.8 Porównanie



Rysunek 17: Porównanie dla zapytania na niepustym wyniku



Rysunek 18: Porównanie dla zapytania na pustym wyniku

4.2 Polecenie explain

1	Finalize GroupAggregate (cost=11243.96..11246.27 rows=1 width=21)
2	Group Key: u.id
3	Filter: (count(f.id) = 5)
4	-> Gather Merge (cost=11243.96..11245.94 rows=16 width=29)
5	Workers Planned: 2
6	-> Partial GroupAggregate (cost=10243.93..10244.07 rows=8 width=29)
7	Group Key: u.id
8	-> Sort (cost=10243.93..10243.95 rows=8 width=29)
9	Sort Key: u.id
10	-> Nested Loop (cost=1.14..10243.81 rows=8 width=29)
11	-> Nested Loop (cost=0.71..10242.04 rows=2 width=29)
12	-> Nested Loop (cost=0.42..10241.40 rows=2 width=16)
13	-> Parallel Seq Scan on addresses a (cost=0.00..10224.50 rows=2 width=8)
14	Filter: ((city)::text = 'Langworthhaven'::text)
15	-> Index Scan using "fki_fk_addressId" on building b (cost=0.42..8.44 rows=1 width=24)
16	Index Cond: (address_id = a.id)
17	-> Index Scan using "Users_pkey" on users u (cost=0.29..0.32 rows=1 width=21)
18	Index Cond: (id = b.owner_id)
19	-> Index Scan using "fki_fk_buildingId" on flats f (cost=0.43..0.82 rows=6 width=16)
20	Index Cond: (building_id = b.id)

Rysunek 19: Polecenie explain dla GetUsersWithFlatNumberInCity (raw)

1	Seq Scan on users u (cost=0.00..5861097.14 rows=500 width=21)
2	Filter: ((SubPlan 1) = 5)
3	SubPlan 1
4	-> Aggregate (cost=58.58..58.59 rows=1 width=4)
5	-> Nested Loop (cost=5.31..58.57 rows=5 width=0)
6	-> Nested Loop (cost=4.89..53.97 rows=1 width=8)
7	-> Bitmap Heap Scan on building b (cost=4.45..20.11 rows=4 width=16)
8	Recheck Cond: (u.id = owner_id)
9	-> Bitmap Index Scan on "fki_fk_ownerId" (cost=0.00..4.45 rows=4 width=0)
10	Index Cond: (owner_id = u.id)
11	-> Memoize (cost=0.43..8.45 rows=1 width=8)
12	Cache Key: b.address_id
13	Cache Mode: logical
14	-> Index Scan using "Addresses_pkey" on addresses a (cost=0.42..8.44 rows=1 width=8)
15	Index Cond: (id = b.address_id)
16	Filter: ((city)::text = 'Langworthhaven'::text)
17	-> Index Only Scan using "fki_fk_buildingId" on flats f (cost=0.43..4.53 rows=6 width=8)
18	Index Cond: (building_id = b.id)

Rysunek 20: Polecenie explain dla GetUsersWithFlatNumberInCity (generated)

5 Podsumowanie

Na podstawie otrzymanych wyników można zauważyć, że zapytania ORM dla bardziej złożonych operacji generują dodatkowe podzapytania. Skutkuje to prawdopodobnie większą liczbą odniesień do pamięci cache (Blocks Hit) oraz większą liczbą zapytań do dysku (Blocks Read).

Kiedy ORM tworzy zapytania działające szybciej niż natywne, różnice w czasie wykonania są zazwyczaj niewielkie, wynoszące maksymalnie kilkaset milisekund dla pustych rekordów. Natomiast w przypadku przewagi zapytań natywnych, różnice te mogą sięgać nawet kilkudziesięciu tysięcy milisekund.

Chociaż ORM podczas generowania zapytań na podstawie stworzonego modelu obiektowego działa zazwyczaj wolniej niż doświadczony programista piszący je ręcznie, posiada on istotne zalety. Przede wszystkim oferuje łatwość utrzymania i zarządzania modelem oraz zapytaniami, co znacząco ułatwia pracę z bazą danych w przypadku dużych i rozbudowanych aplikacji. Dzięki temu, ORM jest wartościowym narzędziem, szczególnie w projektach o większej skali, gdzie zarządzanie kodem staje się kluczowym elementem sukcesu. Dodatkowo warto zauważyć, że dla mało skomplikowanych zapytań wygenerwany SQL praktycznie nie różnił się od tego napisanego ręcznie.

Podsumowując, jeżeli zapytania są bardzo złożone, a ilość danych do przetworzenia jest bardzo dużo (rzędu setek tysięcy rekordów) warto rozważyć pisanie natywnych zapytań SQL. Ponieważ posiadamy pełną kontrolę nad ich wyglądem, co za tym idzie możliwe są dodatkowe optymalizacje, takie jak przykładowo leprze wykorzystanie indeksów. Natomiast w przeciwnym wypadku, gdy zapytania są względnie proste lub pracujemy z małą ilością danych różnice będą pomijalne, zazwyczaj już same silniki bazodanowe są bardzo wydajne i zoptymalizowany w wielu aspektach.