Vinsguru

search..

Recent Posts

Spring WebFlux Aggregation

Materialized View PostgreSQL

5 Comments / Architectural Design Pattern, Architecture, Articles, Data Stream / Event Stream, Design Pattern, Framework, Java, Kubernetes Design Pattern, MicroService, Spring, Spring Boot, Spring WebFlux / By vIns / November 23, 2019

Overview:

In this tutorial, I would like to demo **Materialized View PostgreSQL** with **Spring Boot** to increase the read performance of the application.

Materialized View:

Most of the web based applications are **CRUD** in nature with simple CREATE, READ, UPDATE and DELETE operations. It is also true that in the most of the applications, we do more READ operations than other INSERT, DELETE and UPDATE transactions. Sometimes the READ operations could be very heavy in such a way that we would join multiple tables with aggregate functions. It cloud slow down the performance of the read operation.

The goal of this article to show **Materialized View Pattern** to demo how we can retrieve the prepoluated views of data when the source data is NOT easy to query every time & to improve the performance of your Microservices.

Sample Application:

Choreography Saga Pattern With Spring Boot

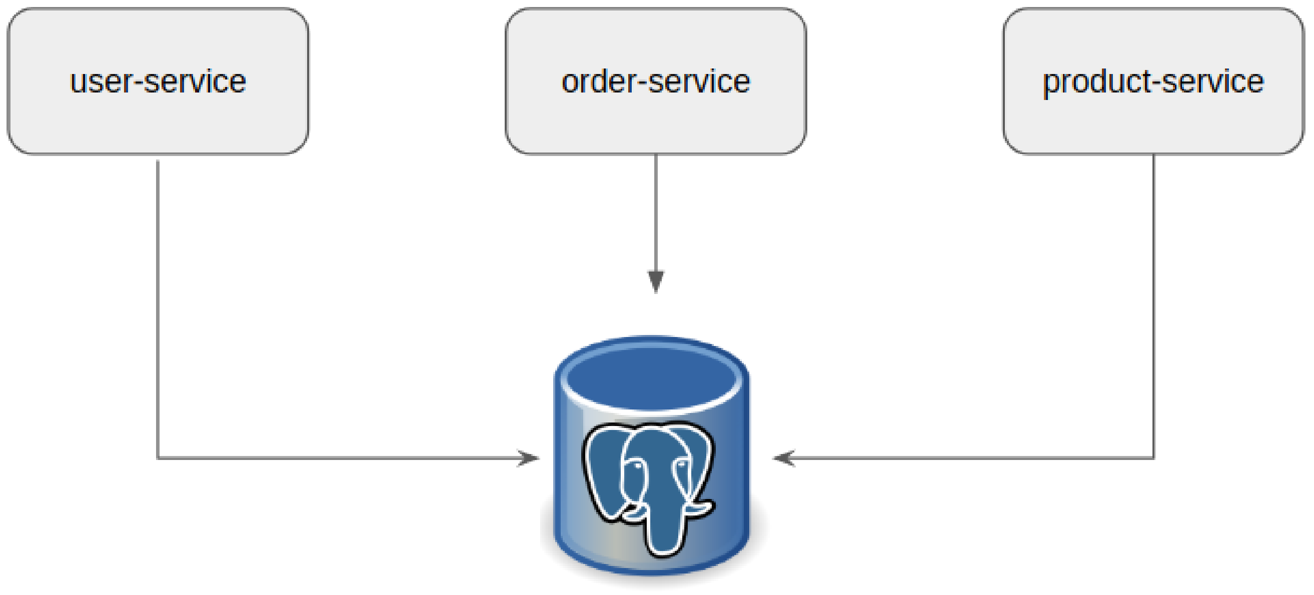
Spring WebFlux WebSocket

gRPC Web Example

Orchestration Saga Pattern With Spring Boot

Selenium WebDriver - How

Lets consider a simple application in which we have 3 services as shown below. (Ideally all these services should have different databases. Here just for this article, I am using same db)



- **user-service:** contains user related operations
- **product-service:** contains product related operations
- **order-service:** This is what we are interested in contains – user orders related functionalities.

Our order-service is responsible for placing an order for the user. It also exposes an end point which provides sale statistics. To understand that better, lets first see the

To Test REST API

Introducing PDFUtil - Compare two PDF files textually or Visually

JMeter - How To Run Multiple Thread Groups i

DB table structure.

```
CREATE TABLE users(
   id serial PRIMARY KEY,
   firstname VARCHAR (50),
   lastname VARCHAR (50),
   state VARCHAR(10)
);

CREATE TABLE product(
   id serial PRIMARY KEY,
   description VARCHAR (500),
   price numeric (10,2) NOT NULL
);

CREATE TABLE purchase_order(
    id serial PRIMARY KEY,
    user_id integer references users (id),
    product_id integer references product (id)
);
```

Order-service exposes an end point which provides the total sale values by users state.

Multiple Test Environments

Selenium WebDriver - Design Patterns in Test Automation - Factory Pattern

Kafka Stream With

```
select
    u.state,
    sum(p.price) as total_sale
from
    users u,
    product p,
    purchase_order po
where
    u.id = po.user_id
    and p.id = po.product_id
group by u.state
order by u.state
```

We could create a view to get the results we are interested in as shown here.

```
create view purchase_order_summary
as
select u.state,
       sum(p.price) as total_sale
from users u,
     product p,
     purchase_order po
where u.id = po.user_id
```

Spring Boot

JMeter - Real Time Results - InfluxDB & Grafana - Part 1 - Basic Setup

JMeter - Distributed Load Testing

```
    and p.id = po.product_id
  group by u.state
  order by u.state
```

So executing below query provides the **total_sale** by **state**

```
select * from purchase_order_summary;
```

Spring Boot Application:

- Lets create a simple spring boot application first before we dive into materialized view implementation.
- I use below dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
```

using Docker

JMeter - How To Test REST API / MicroServices

JMeter - Property File Reader - A custom config element

```
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

- User Entity

```
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstname;
    private String lastname;
    private String state;

    // getters & setters

}
```

Selenium WebDriver - How To Run Automated Tests Inside A Docker Container - Part 1

Categories

- Product Entity

```
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String description;
    private double price;

    // getters & setters

}
```

- Purchase Order Entity

```
@Entity
public class PurchaseOrder {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

Architecture (62)
Arquillian (9)
Articles (204)
AWS / Cloud (17)
AWS (4)
Best Practices (75)
CI / CD / DevOps (51)
Data Stream / Event Stream (27)
Database (9)

```
    private Long id;

    private Long userId;
    private Long productId;

    // getters & setters

}
```

- Purchase Order Summary Entity

```
@Entity
public class PurchaseOrderSummary {

    @Id
    private String state;
    private Double totalSale;

    // getters & setters

}
```

- DTO – Purchase Order Summary

Design
Pattern (41)
Architectural
Design
Pattern (26)
Factory
Pattern (1)
Kubernetes
Design
Pattern (18)
Strategy
Pattern (1)
Distributed
Load Test
(9)
Docker (24)
ElasticSearc
h (2)

```
public class PurchaseOrderSummaryDto {
    private String state;
    private double totalSale;

     // getters and setters

}
```

- Repository – DAO Layer. Here we use Spring data JPA.

```
@Repository
public interface UserRepository extends JpaRepository<User, L
}

@Repository
public interface ProductRepository extends JpaRepository<Produ
}

@Repository
public interface PurchaseOrderRepository extends JpaRepository
}

@Repository
```

EMail Validation (1)

Framework (104)

Functional Test Automation (83)

Puppeteer (1)

QTP (10)

Selenium (76)

Extend WebDriver (11)

Ocular (2)

Page Object Design (17)

```
public interface PurchaseOrderSummaryRepository extends JpaRep
}
```

- Purchase Order Service and Implementation

```
public interface PurchaseOrderService {
    void placeOrder(int userIndex, int productIndex);
    List<PurchaseOrderSummaryDto> getSaleSummary();
}

@Service
public class PurchaseOrderServiceImpl implements PurchaseOrder

    @Autowired
    private PurchaseOrderSummaryRepository purchaseOrderSummar

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private ProductRepository productRepository;

    @Autowired
```

Report (8)

Selenium

Grid (10)

TestNG (7)

gRPC (15)

Java (81)

Guice (2)

Reactor (41)

Jenkins (17)

Kafka (9)

Kubernetes

(8)

Linkerd (2)

Maven (7)

messaging

(11)

MicroService

(76)

```
    private PurchaseOrderRepository purchaseOrderRepository;

    private List<User> users;
    private List<Product> products;

    @PostConstruct
    private void init(){
        this.users = this.userRepository.findAll();
        this.products = this.productRepository.findAll();
    }

    @Override
    public void placeOrder(int userIndex, int productIndex) {
        PurchaseOrder purchaseOrder = new PurchaseOrder();
        purchaseOrder.setProductId(this.products.get(productI
        purchaseOrder.setUserId(this.users.get(userIndex).get
        this.purchaseOrderRepository.save(purchaseOrder);
    }

    @Override
    public List<PurchaseOrderSummaryDto> getSaleSummary() {
        return this.purchaseOrderSummaryRepository.findAll()
                                .stream()
                                .map(pos -> {
```

Mongo (4)

Monitoring (13)

FileBeat (1)

Grafana (5)

InfluxDB (7)

Kibana (2)

Multi Factor Authentication (2)

nats (4)

Performance Testing (44)

Extend JMeter (5)

JMeter (43)

Workload Model (2)

```
                            PurchaseOrderSummaryDto dto = new
                            dto.setState(pos.getState());
                            dto.setTotalSale(pos.getTotalSale(
                            return dto;
                        })
                        .collect(Collectors.toList());
    }
}
```

- REST Controller

```
@RestController
@RequestMapping("po")
public class PurchaseOrderController {

    @Autowired
    private PurchaseOrderService purchaseOrderService;

    @GetMapping("/sale/{userIndex}/{prodIndex}")
    public void placeOrder(@PathVariable final int userIndex,
                           @PathVariable final int prodIndex){
        this.purchaseOrderService.placeOrder(userIndex, prodIn
    }
    @GetMapping("/summary")
    public List<PurchaseOrderSummaryDto> getSummary(){
        return this.purchaseOrderService.getSaleSummary();
    }

}
```

Performance Test – DB View:

- I inserted 10000 users in the users table
- I inserted 1000 products into the product table
- I inserted 5 Million user orders for random user + product combination into the purchase_order table
- I run a performance test using JMeter with 11 concurrent users
    - 10 users for sending the requests for READ
    - 1 user for creating purchase order continuously

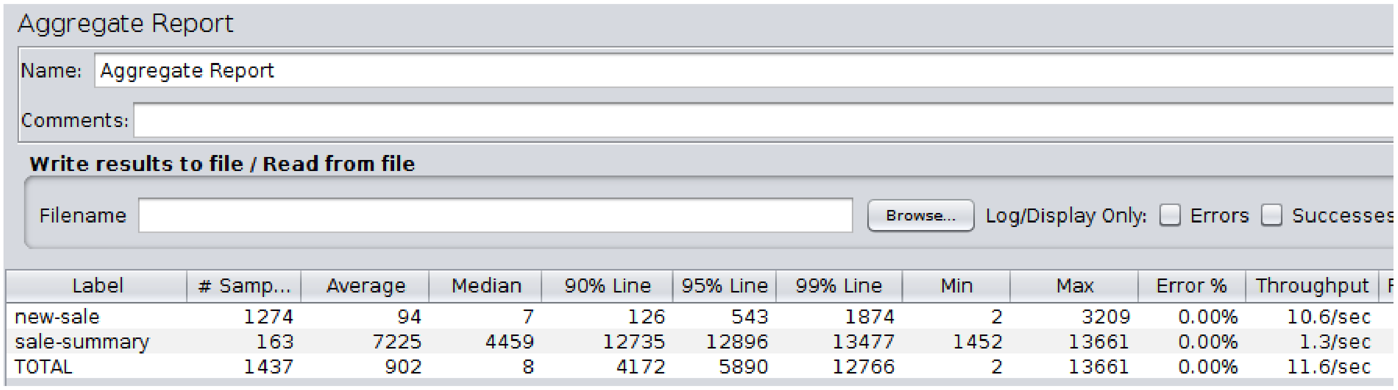
Spring Data (11)

Spring WebFlux (62)

Udemy Courses (5)

Utility (20)

WebSocket (2)

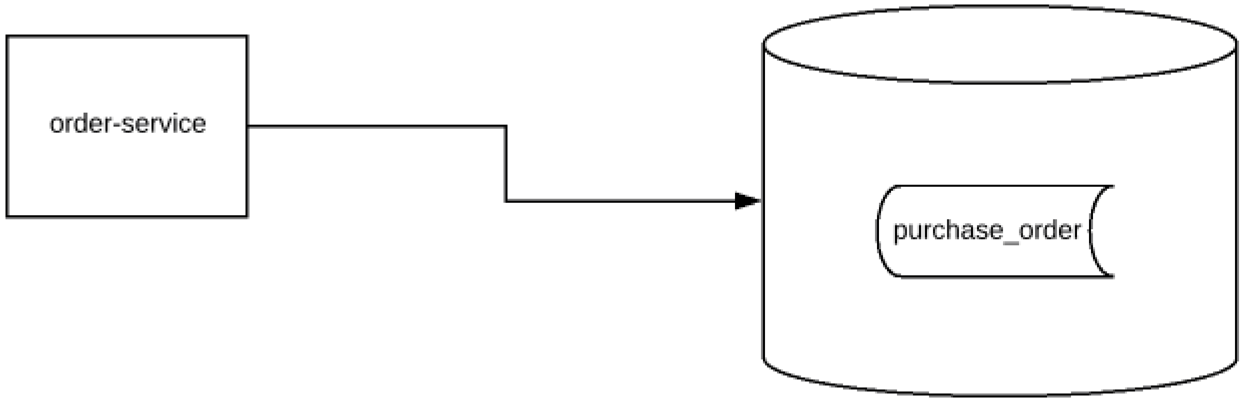

Aggregate Report
Name: Aggregate Report
Comments:
Write results to file / Read from file
Filename
Browse...
Log/Display Only: Errors Successes

Label	# Samp...	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput
new-sale	1274	94	7	126	543	1874	2	3209	0.00%	10.6/sec
sale-summary	163	7225	4459	12735	12896	13477	1452	13661	0.00%	1.3/sec
TOTAL	1437	902	8	4172	5890	12766	2	13661	0.00%	11.6/sec

- As we can see, sale-summary average response time is 7.2 second. It is trying to aggregate the information by state from the purchase_order table for every GET request.

Problem With Views:

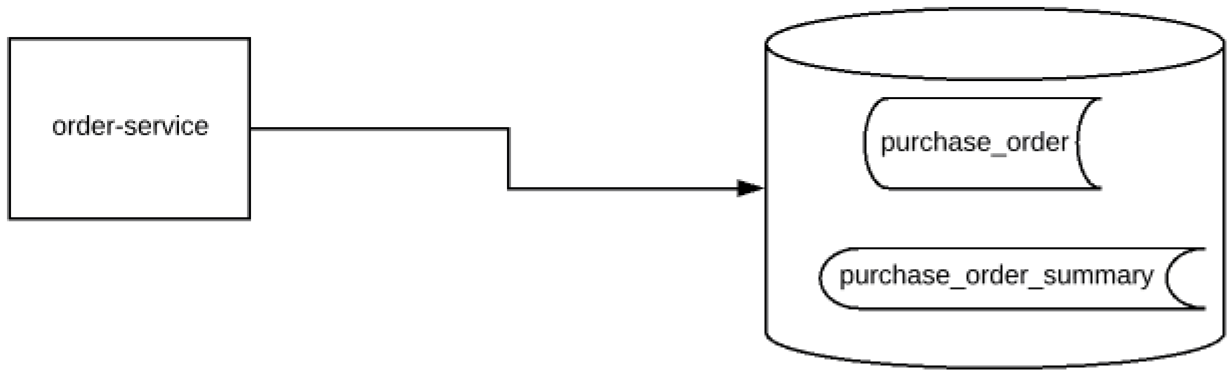
- Views are virtual tables in a DB
- Even though DB Views are great in hiding some sensitive information and provide data in a simpler table like structure, the underlying query is executed every time. It could be required in some cases where the data changes very frequently. However in most of the cases it could affect the performance of the application very badly!



Materialized View PostgreSQL:

Materialized Views are most likely views in a DB. But they are not virtual tables. Instead the data is actually calculated / retrieved using the query and the result is stored in the hard disk as a separate table. So when we execute below query, the underlying query is not executed every time. Instead the data is fetched directly from the table. This is something like using the cached data. So it improves the performance.

```
select * from purchase_order_summary;
```



```
CREATE MATERIALIZED VIEW purchase_order_summary
AS
select
    u.state,
    sum(p.price) as total_sale
from
    users u,
    product p,
    purchase_order po
where
    u.id = po.user_id
    and p.id = po.product_id
group by u.state
order by u.state
```

```
WITH NO DATA;
CREATE UNIQUE INDEX state_category ON purchase_order_summary (

-- to load into the purchase_order_summary
REFRESH MATERIALIZED VIEW CONCURRENTLY purchase_order_summary;
```

The obvious question would be what if the source data is updated. That is, if we make new entry into the purchase_order table, how the purchase_order_summary table will be updated!? It will not automatically update. We need to make some actions to do that.

Materialized View PostgreSQL – Auto Update With Triggers:

- We need to update **purchase_order_summary** only when we make entries into the **purchase_order**. (I ignore delete/update operations as of now). So lets create a trigger to update the materialized views whenever we make entries into purchase_order table.
- So lets start with creating a function first to update the materialized view.

```
CREATE OR REPLACE FUNCTION refresh_mat_view()
  RETURNS TRIGGER LANGUAGE plpgsql
  AS $$
  BEGIN
  REFRESH MATERIALIZED VIEW CONCURRENTLY purchase_order_summar
  RETURN NULL;
  END $$;
```

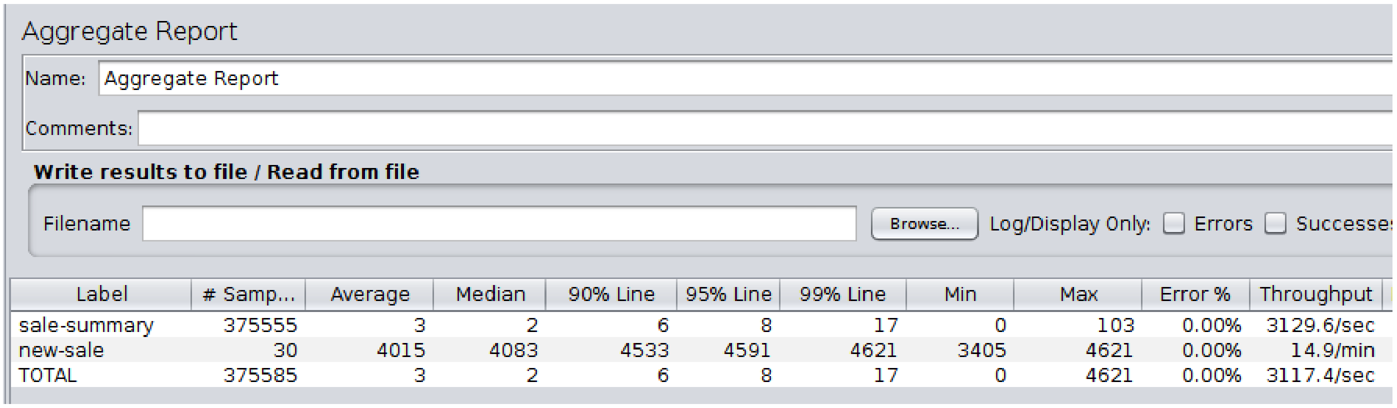
- The above function should be called whenever we make entries into the purchase_order table. So I create an after insert trigger.

```
CREATE TRIGGER refresh_mat_view_after_po_insert
  AFTER INSERT
  ON purchase_order
  FOR EACH STATEMENT
  EXECUTE PROCEDURE refresh_mat_view();
```

Materialized View – Performance Test:

- I re-run the same performance test.

- This time I get exceptionally great result for my sale-summary. As the underlying query is not executed for every GET request, the performance is great! The throughput goes above 3000 requests / second.
- However the performance of the new purchase_order request is affected as it is responsible for updating the materialized view.
- In some cases it could be OK if we are doing the new order placement asynchronously.


Aggregate Report

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: ☐ Errors ☐ Successe

Label	# Samp...	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput
sale-summary	375555	3	2	6	8	17	0	103	0.00%	3129.6/sec
new-sale	30	4015	4083	4533	4591	4621	3405	4621	0.00%	14.9/min
TOTAL	375585	3	2	6	8	17	0	4621	0.00%	3117.4/sec

But do we really need to update summary for every order. Instead, we could update the materialized view certain interval like 5 seconds. The data might not be very accurate for few seconds. It will eventually be refreshed in 5 seconds. This could be a nice solution to avoid the new order performance issue which we saw above.

- Lets drop the trigger and the function we had created.

- Lets create a simple procedure to refresh the view. This procedure would be called periodically via Spring boot.

```
-- drop trigger

drop trigger refresh_mat_view_after_po_insert ON purchase_orde

-- drop function
drop function refresh_mat_view();

-- create procedure
CREATE OR REPLACE PROCEDURE refresh_mat_view()
LANGUAGE plpgsql
AS $$
BEGIN
  REFRESH MATERIALIZED VIEW CONCURRENTLY purchase_order_summar
END;
$$;
```

Materialized View With Spring Boot:

- I add the new component which will be responsible for calling the procedure periodically.

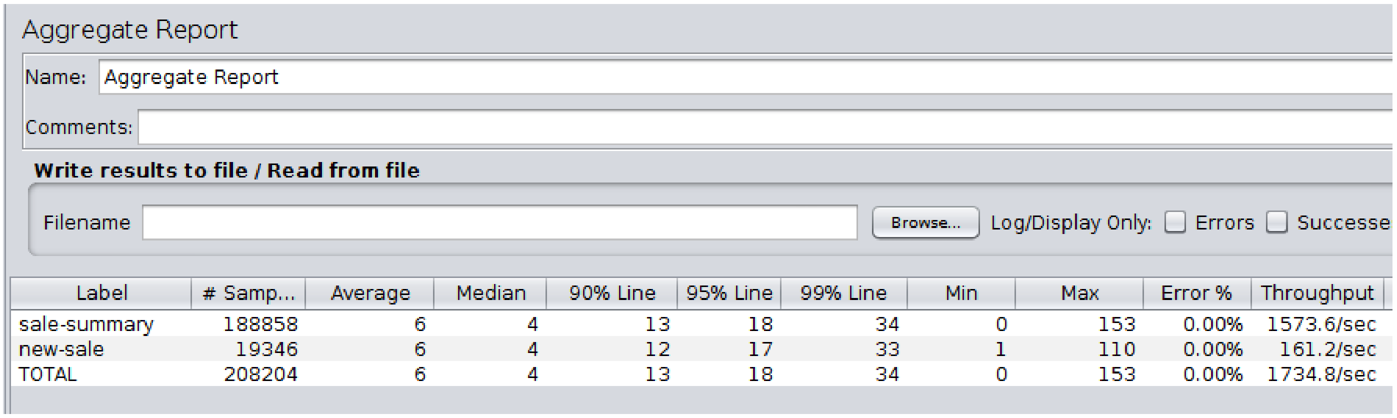
```
@Component
public class MaterializedViewRefresher {

    @Autowired
    private EntityManager entityManager;

    @Transactional
    @Scheduled(fixedRate = 5000L)
    public void refresh(){
        this.entityManager.createNativeQuery("call refresh_mat
    }

}
```

- I re-run the same performance test to get the below results.
- I get extremely high throughput for my both read and write operations.
- The average response time is 6 milliseconds in both cases.



Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename

Log/Display Only: Errors Successe

Label	# Samp...	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput
sale-summary	188858	6	4	13	18	34	0	153	0.00%	1573.6/sec
new-sale	19346	6	4	12	17	33	1	110	0.00%	161.2/sec
TOTAL	208204	6	4	13	18	34	0	153	0.00%	1734.8/sec

Summary:

We were able to demonstrate the usage of **Materialized View PostgreSQL with Spring Boot** to improve the performance of the read heavy operations for the Microservices architecture. Implementing this pattern will also enable us implementing **CQRS** pattern to improve the performance further of our microservices.

Read more about **Microservice Design Patterns**.

- CQRS Pattern With Spring Boot + Kafka
- Cache-Aside / Read-Through Pattern With Spring Boot + Redis

The source code is available here.

Share This:

« Selenium WebDriver with Docker – Brand New Udemy Course

Java Reactive Programming – Introduction Guide »

5 thoughts on “Materialized View PostgreSQL”

Anas Ellithy

June 12, 2021 at 8:31 PM

First of all, thanks for this good and detailed article. I found it very weird to say "Materialized View PostgreSQL with Spring Boot which is one of the Microservice Design Patterns"
What I understand is that materialized view is a DB caching technique regardless you are using a microservices architecture or not.
Kindly clarify.

Reply

vIns

June 13, 2021 at 3:18 PM

You are right! This would be the actual Microservice related pattern for materialzied view – https://www.vinsguru.com/event-carried-state-transfer/.
Will correct this.

Reply

Neha

August 4, 2022 at 3:10 PM

Can you share Jmeter file and how you've tested it ?

Reply

Neha

December 12, 2022 at 8:58 AM

Hi Vinoth – I dont see any reply from you yet. Could you please guide us on steps to how to create Aggregate report in Jmeter? I am completely new to this.

Reply

vIns

December 12, 2022 at 2:49 PM

Hi.. JMeter is a simple tool for load testing. It is difficult to explain via comments. Instead there is another tool call called "apache bench" which is simple command line tool you can use for load testing.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.

Post Comment