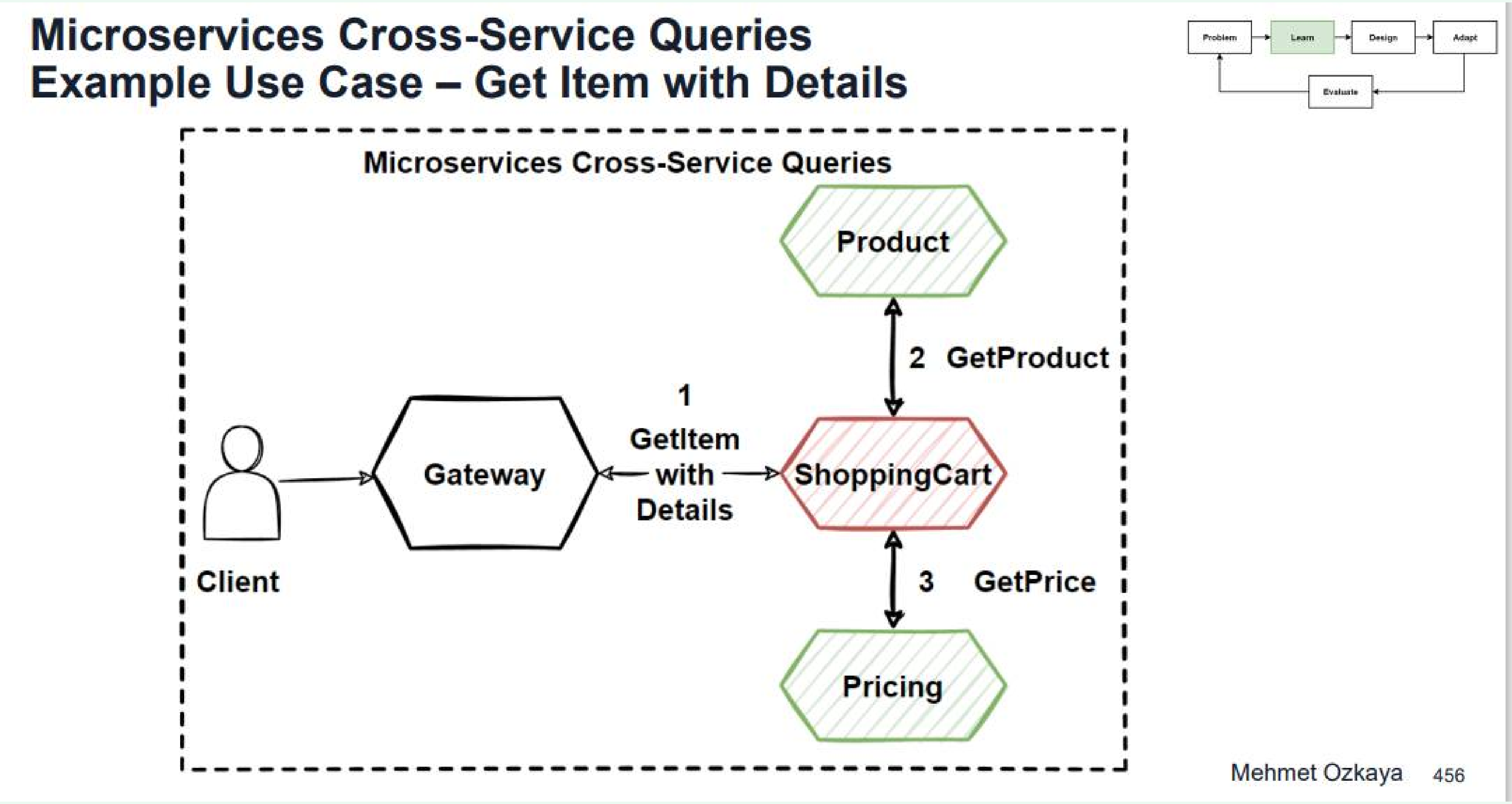
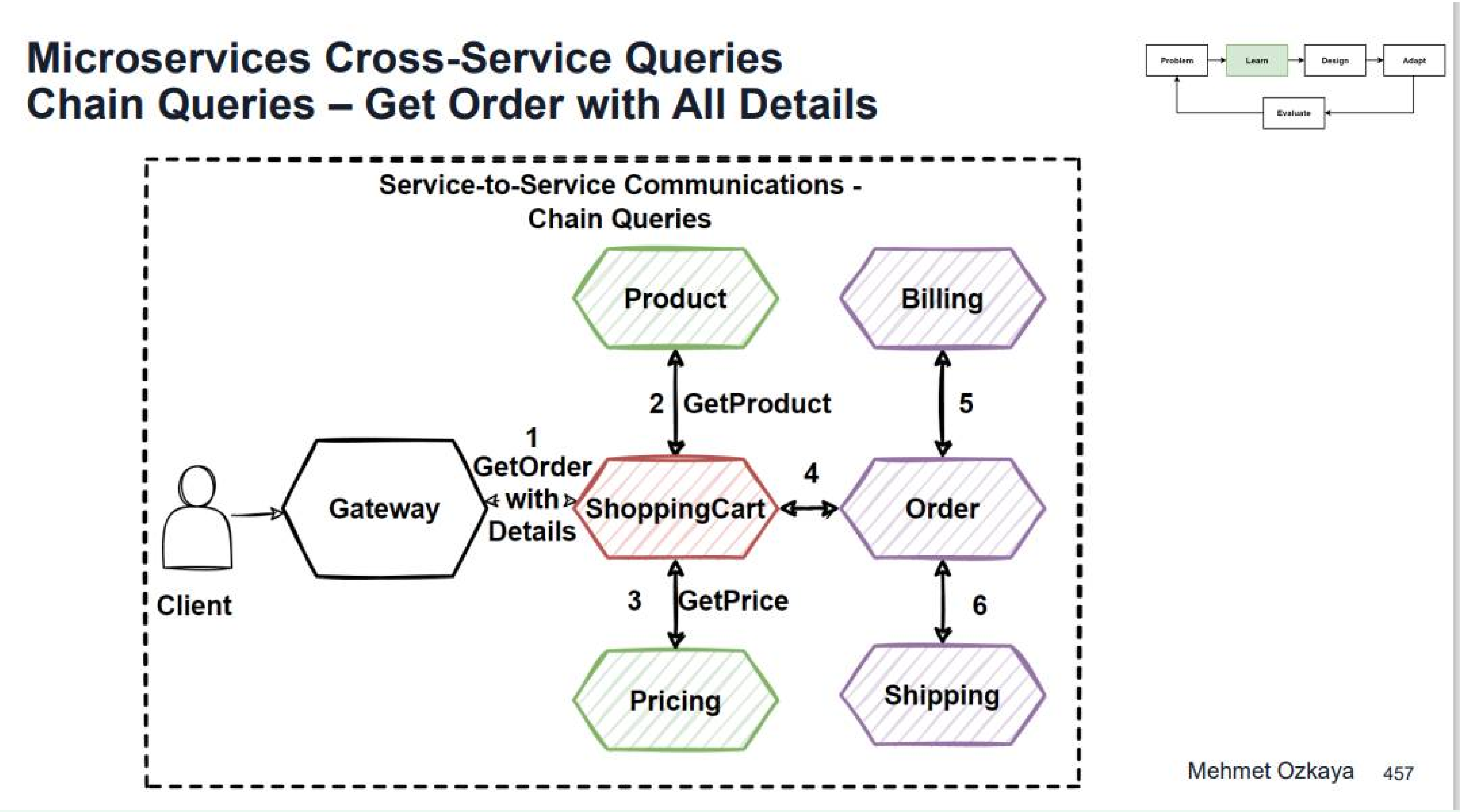
# Microservices Cross-Service Queries



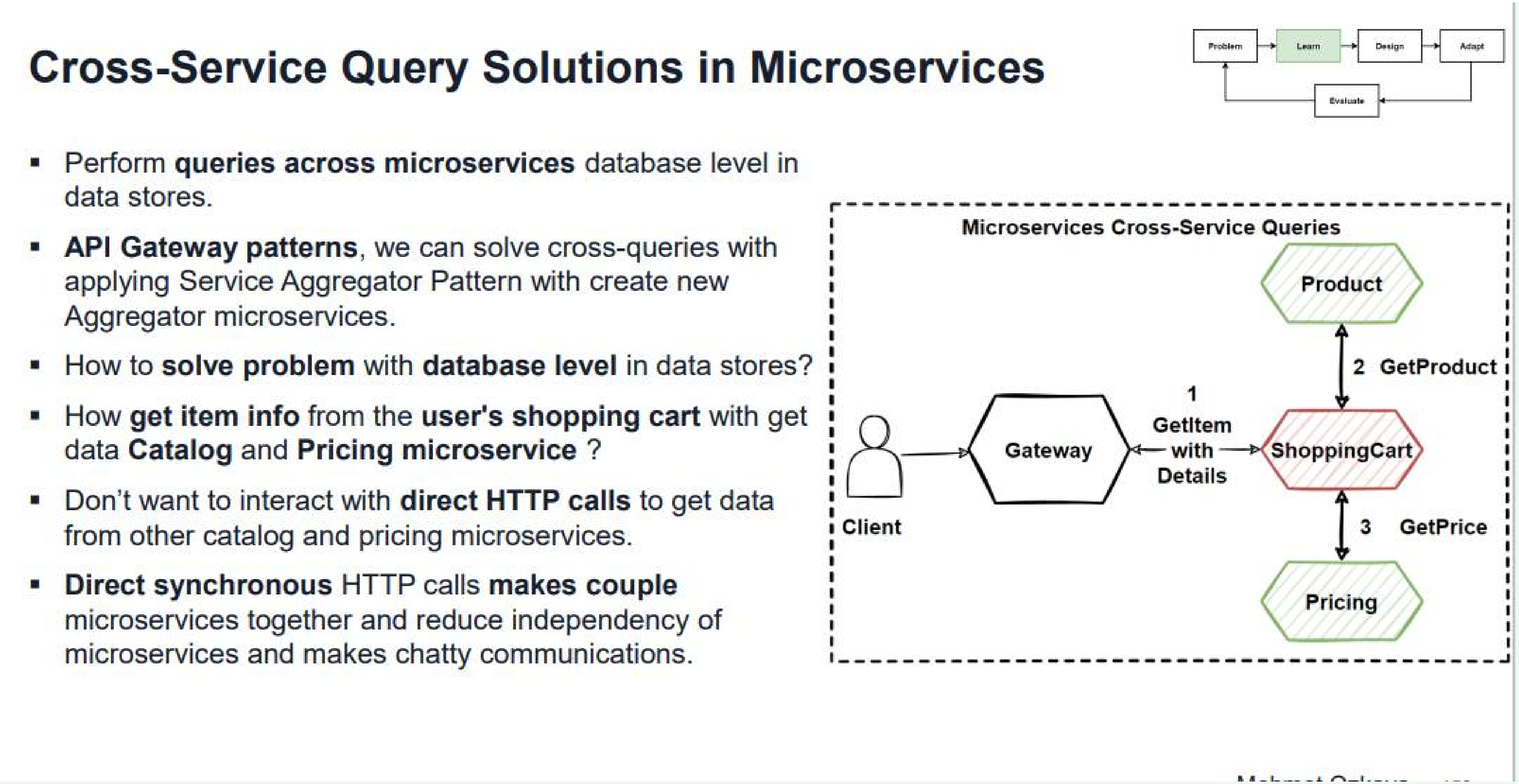
## Microservices Cross-Service Queries
## Example Use Case – Get Item with Details

Mehmet Ozkaya 456



**Microservices Cross-Service Queries**
**Chain Queries – Get Order with All Details**

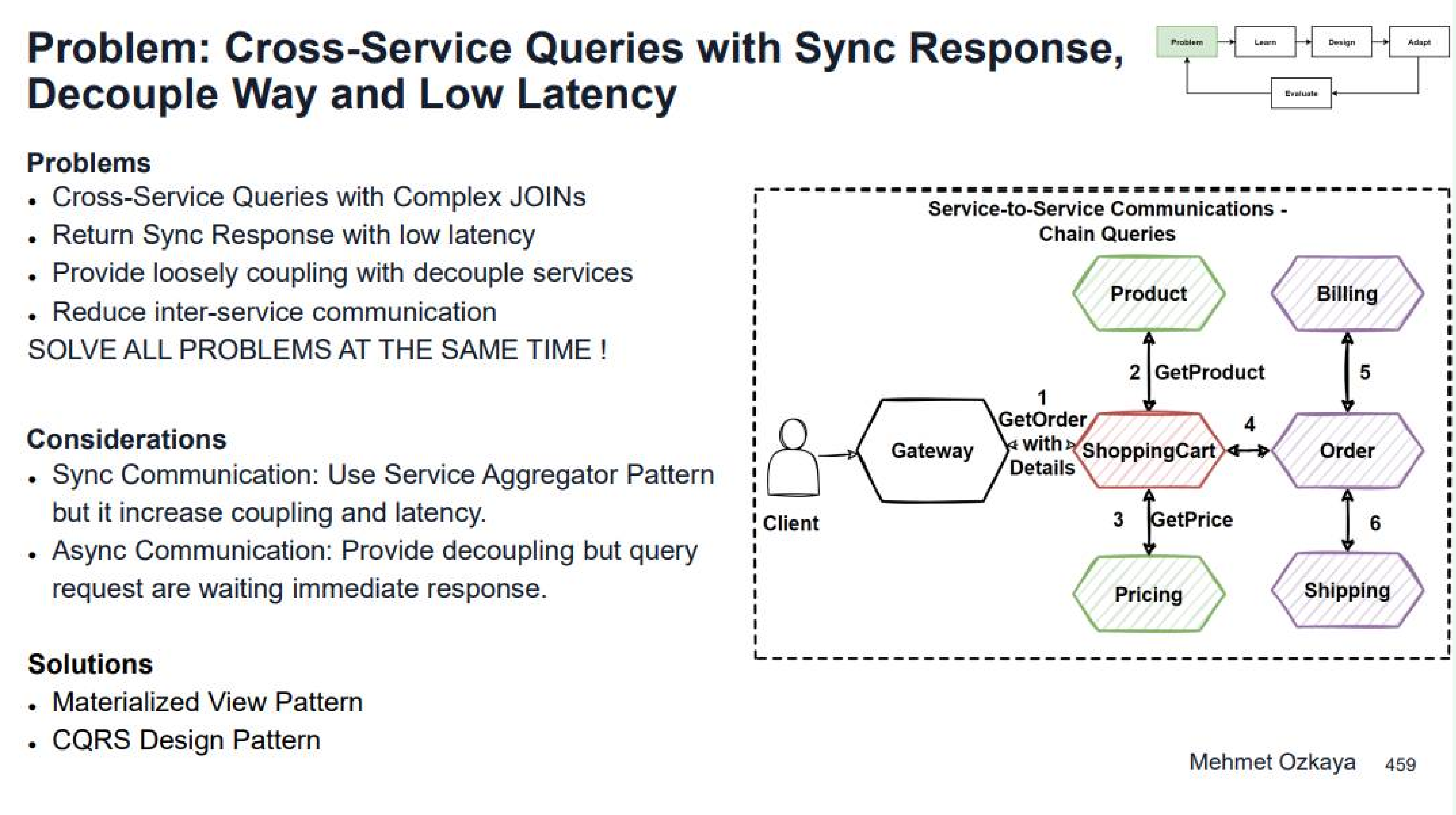
Mehmet Ozkaya 457



# Cross-Service Query Solutions in Microservices

- Perform **queries across microservices** database level in data stores.
- **API Gateway patterns**, we can solve cross-queries with applying Service Aggregator Pattern with create new Aggregator microservices.
- How to **solve problem** with **database level** in data stores?
- How **get item info** from the **user's shopping cart** with get data **Catalog** and **Pricing microservice** ?
- Don't want to interact with **direct HTTP calls** to get data from other catalog and pricing microservices.
- **Direct synchronous** HTTP calls **makes couple** microservices together and reduce independency of microservices and makes chatty communications.

Printed using Save ChatGPT as PDF, powered by PDFCrowd HTML to PDF API.



# Problem: Cross-Service Queries with Sync Response, Decouple Way and Low Latency

**Problems**

- Cross-Service Queries with Complex JOINs
- Return Sync Response with low latency
- Provide loosely coupling with decouple services
- Reduce inter-service communication

SOLVE ALL PROBLEMS AT THE SAME TIME !

**Considerations**

- Sync Communication: Use Service Aggregator Pattern but it increase coupling and latency.
- Async Communication: Provide decoupling but query request are waiting immediate response.

**Solutions**

- Materialized View Pattern
- CQRS Design Pattern

Mehmet Ozkaya 459

Kindly explain the above images in details

The images you uploaded depict cross-service query patterns in microservices, with use cases involving a shopping cart scenario and querying multiple services.

## 1. Microservices Cross-Service Queries: Example Use Case – Get Item with Details

In this scenario, the goal is for a client to retrieve an item along with its details, using a microservices architecture. Here's how the interaction flows:

- **Client** sends a `` `GetItem with Details` `` request to the **Gateway**.
- The **Gateway** forwards this request to the **ShoppingCart** service.
- The **ShoppingCart** service, in turn, calls two other services:
    1. **Product** service via a `` `GetProduct` `` call to retrieve product details.
    2. **Pricing** service via a `` `GetPrice` `` call to retrieve the pricing details.

This setup allows the **ShoppingCart** service to aggregate responses from the **Product** and **Pricing** services to send back a detailed item response to the client. It demonstrates how microservices communicate with each other in a cross-service query model, where different services contribute to constructing the final response.

## 2. Chain Queries: Get Order with All Details

This scenario extends the above example by introducing additional services involved in processing an order. The chain query model is used here, where the client requests detailed order information.

- The **Client** sends a `` `GetOrder with Details` `` request to the **Gateway**.
- The **Gateway** forwards the request to the **ShoppingCart** service.
- The **ShoppingCart** service makes a series of calls:
    1. To the **Product** service via `` `GetProduct` ``.
    2. To the **Pricing** service via `` `GetPrice` ``.
    3. Then it communicates with the **Order** service, sending along the order details.
- After the **Order** service processes the order, it calls:
    1. **Billing** service to handle billing information.
    2. **Shipping** service to handle shipping details.

This chain query model demonstrates how microservices need to communicate with multiple other services to compile all necessary information for an order, such as product, pricing, billing, and shipping data.

## 3. Cross-Service Query Solutions in Microservices

This section focuses on the strategies for handling cross-service queries efficiently. The key points include:

- **Queries across microservices**: Data is distributed across microservices, and queries must be able to retrieve data from different microservice databases.
- **API Gateway patterns**: By applying the **Service Aggregator Pattern**, API gateways can collect responses from multiple services and aggregate them for the client, reducing the need for direct client-to-service communication.
- **Direct synchronous HTTP calls**: This pattern leads to "coupling" between services, increasing communication overhead and dependencies between microservices. It can lead to "chatty" services, where multiple HTTP calls create latency.

This slide emphasizes reducing such direct calls and making services more independent, improving scalability and performance.

### 4. Cross-Service Queries with Sync Response, Decouple Way, and Low Latency

This final section addresses problems and solutions for complex cross-service queries, with a focus on achieving decoupling and low-latency responses:

**Problems:**

- **Complex JOINs across services**: Performing complex queries that require multiple services to communicate can introduce latency and tight coupling.
- **Synchronous communication**: Ensuring low-latency responses while still decoupling services is challenging.
- **Reducing inter-service communication**: A major goal is to minimize the amount of communication between services to improve overall performance.

**Considerations:**

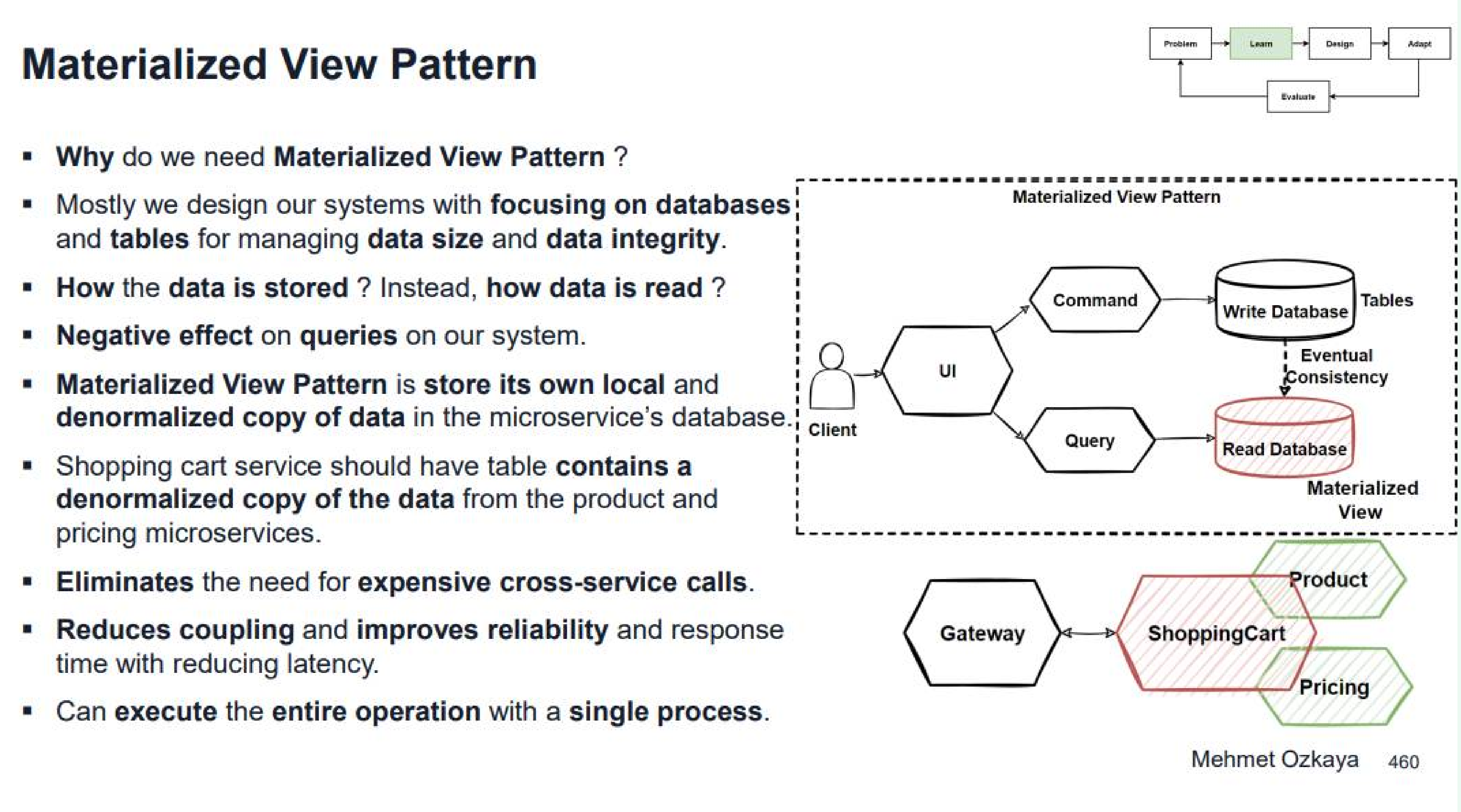
- **Sync communication** (Service Aggregator Pattern): While this ensures the client receives a combined response, it increases coupling and latency.
- **Async communication**: Decouples services, but since the response isn't immediate, it doesn't fit scenarios requiring immediate feedback.

**Solutions:**

- **Materialized View Pattern**: Precompute and store results from multiple services to reduce the need for complex cross-service queries.
- **CQRS (Command Query Responsibility Segregation) Design Pattern**: This pattern separates read and write operations, improving performance by allowing independent scaling of the query side of the system.

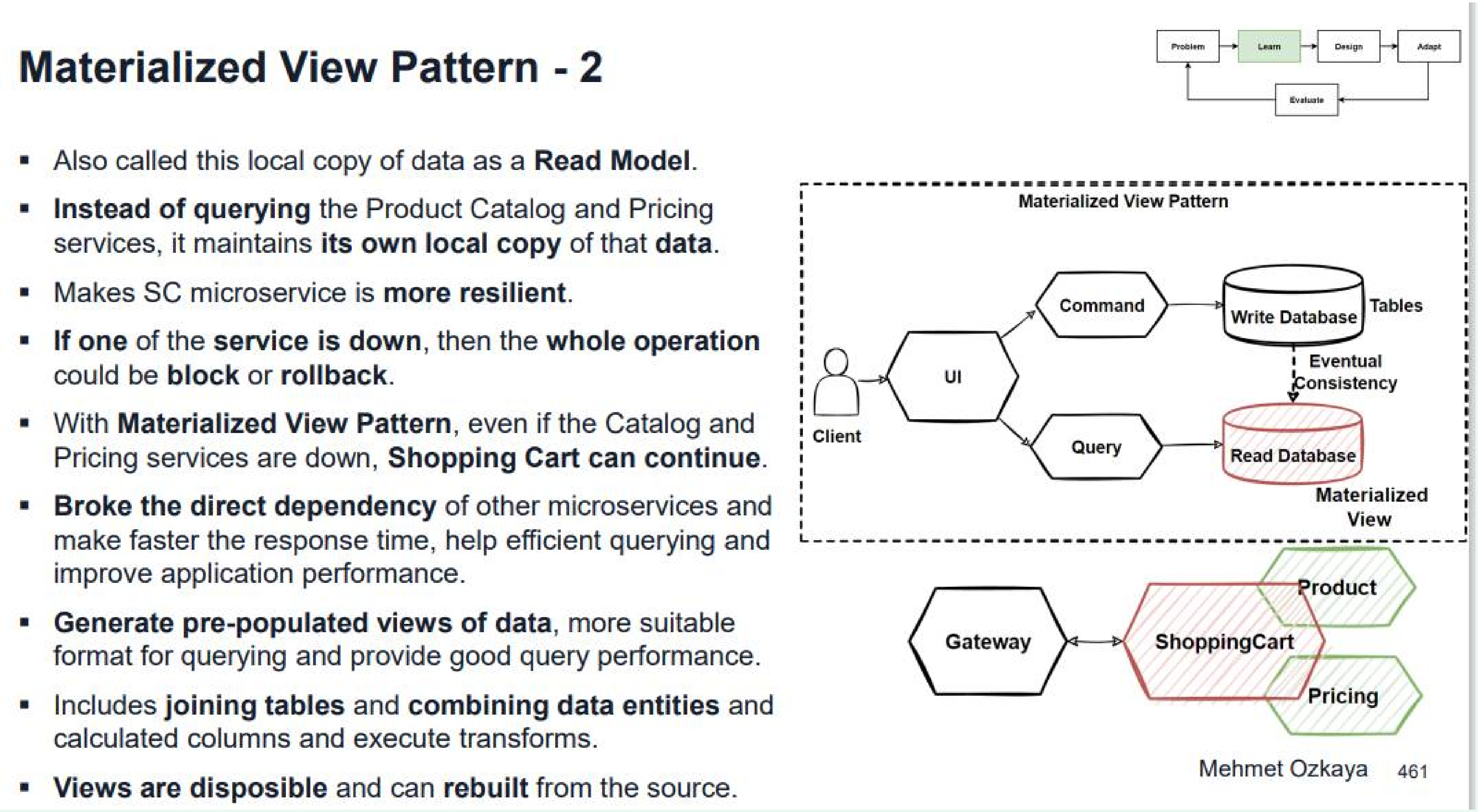
---

These slides emphasize the importance of managing communication between microservices carefully to avoid tightly coupling services, reduce latency, and optimize performance, especially in complex systems where multiple services need to work together.



# Materialized View Pattern

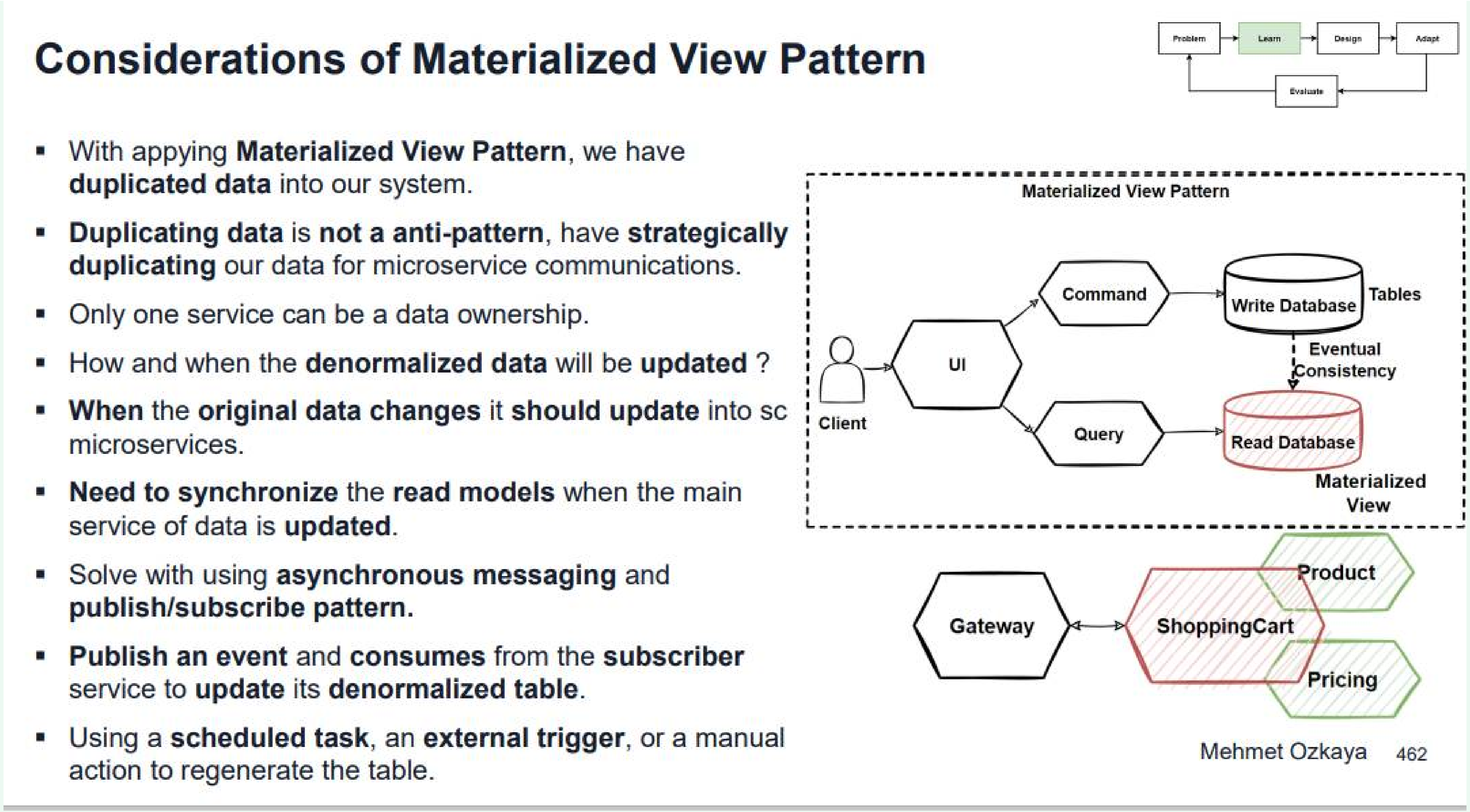
- **Why** do we need **Materialized View Pattern** ?
- Mostly we design our systems with **focusing on databases** and **tables** for managing **data size** and **data integrity**.
- **How** the **data is stored** ? Instead, **how data is read** ?
- **Negative effect** on **queries** on our system.
- **Materialized View Pattern** is **store its own local** and **denormalized copy of data** in the microservice's database.
- Shopping cart service should have table **contains a denormalized copy of the data** from the product and pricing microservices.
- **Eliminates** the need for **expensive cross-service calls**.
- **Reduces coupling** and **improves reliability** and response time with reducing latency.
- Can **execute** the **entire operation** with a **single process**.



# Materialized View Pattern - 2

- Also called this local copy of data as a **Read Model**.
- **Instead of querying** the Product Catalog and Pricing services, it maintains **its own local copy** of that **data**.
- Makes SC microservice is **more resilient**.
- **If one** of the **service is down**, then the **whole operation** could be **block** or **rollback**.
- With **Materialized View Pattern**, even if the Catalog and Pricing services are down, **Shopping Cart can continue**.
- **Broke the direct dependency** of other microservices and make faster the response time, help efficient querying and improve application performance.
- **Generate pre-populated views of data**, more suitable format for querying and provide good query performance.
- Includes **joining tables** and **combining data entities** and calculated columns and execute transforms.
- **Views are disposible** and can **rebuilt** from the source.

Mehmet Ozkaya 461



# Considerations of Materialized View Pattern

- With appying **Materialized View Pattern**, we have **duplicated data** into our system.
- **Duplicating data** is **not a anti-pattern**, have **strategically duplicating** our data for microservice communications.
- Only one service can be a data ownership.
- How and when the **denormalized data** will be **updated** ?
- **When** the **original data changes** it **should update** into sc microservices.
- **Need to synchronize** the **read models** when the main service of data is **updated**.
- Solve with using **asynchronous messaging** and **publish/subscribe pattern.**
- **Publish an event** and **consumes** from the **subscriber** service to **update** its **denormalized table**.
- Using a **scheduled task**, an **external trigger**, or a manual action to regenerate the table.

Mehmet Ozkaya 462

Kindly explain the above images in details

The images you uploaded provide a detailed explanation of the **Materialized View Pattern** and how it helps to solve cross-service queries in microservices, particularly in the context of reducing complexity, improving reliability, and lowering latency.

## 1. Materialized View Pattern Overview

- **Why use the Materialized View Pattern?**
    - In microservices, we often focus on the integrity and size of the data stored in databases. However, when performing cross-service queries, the focus should shift to how data is **read** rather than how it is stored.
    - Traditional approaches can lead to negative impacts on queries, causing performance bottlenecks.
    - **Materialized View Pattern** creates a **denormalized copy** of data that a microservice needs from other services, like product and pricing data in the shopping cart example.
    - By having its own local copy of the required data, the shopping cart service avoids making expensive cross-service calls, leading to better reliability and reduced latency.

- **How does the Materialized View Pattern work?**
    - The shopping cart service maintains its own table containing denormalized data from the **Product** and **Pricing** microservices.
    - The pattern eliminates the need for repeated calls to external services, as the necessary data is available locally.
    - It reduces coupling between services, improving both reliability and response times.
    - The entire operation can be executed in a single process by the shopping cart service without relying on external microservices.

## 2. Materialized View Pattern in Detail

- **Read Model:**
    - The denormalized data in the shopping cart service is also referred to as a **Read Model**.
    - Instead of querying the Product and Pricing services every time a request is made, the shopping cart service keeps a **local copy** of the product and pricing data.
    - This makes the system more resilient. If any dependent service (like the Product or Pricing service) goes down, the shopping cart can still continue operating.
    - The **direct dependency** between the shopping cart and other services is broken, which helps improve performance by speeding up query responses.

- **Pre-populated Views of Data:**
    - The Materialized View can generate **pre-populated views** of data, optimized for querying and better performance.
    - These views may include tables that join and combine data entities, allowing for more efficient query processing.

## 3. Considerations of Materialized View Pattern

- **Duplicating Data:**
    - Applying the Materialized View Pattern leads to **data duplication** within the system, but this is not considered an anti-pattern.
    - Data duplication in this context is a strategic choice made to improve system communication and performance.
- **Data Ownership and Synchronization:**
    - Only one service should be the owner of a particular piece of data. For example, the Product service owns product data, and the Pricing service owns pricing data.
    - When the original data (in the Product or Pricing service) changes, the denormalized copy in the shopping cart service needs to be updated.
    - Synchronizing the **read models** (denormalized copies) with the latest data requires using **asynchronous messaging** and the **publish/subscribe pattern**.
- **Update Mechanism:**
    - The system publishes events when data changes in the source services (like Product or Pricing).
    - The shopping cart service, as a subscriber, consumes these events to update its denormalized table.
    - Updates can also be triggered by scheduled tasks, external triggers, or manual actions.

## Summary

The **Materialized View Pattern** helps to improve the performance of cross-service queries in microservices by reducing dependency on synchronous service-to-service communication and eliminating the need for complex joins. By creating a local, denormalized copy of the data in a service, the pattern enhances system reliability, reduces coupling, and ensures that services like the shopping cart can continue to function even if dependent services are down. However, care must be taken to synchronize the denormalized data with the source of truth through mechanisms like asynchronous messaging.

Is this conversation helpful so far?

ChatGPT can make mistakes. Check important info.