Adobe Premiere Pro Uses

what is adobe premier pro used for

Adobe Premiere Pro is a professional video editing software widely used for editing videos, films, and other media projects. Here are some of its primary uses:

1. **Video Editing**: Premiere Pro allows users to edit videos by trimming, cutting, rearranging, and merging clips. It supports various video formats and resolutions, including 4K, 8K, and VR.

2. **Color Correction and Grading**: It offers advanced tools for adjusting colors, correcting exposure, and enhancing the overall look of the video, with features like Lumetri Color.

3. **Audio Editing**: The software includes audio editing features, enabling users to adjust sound levels, add effects, reduce noise, and synchronize audio with video.

4. **Transitions and Effects**: Premiere Pro provides a wide range of transitions, visual effects, and filters that can be applied to video clips for creative enhancement.

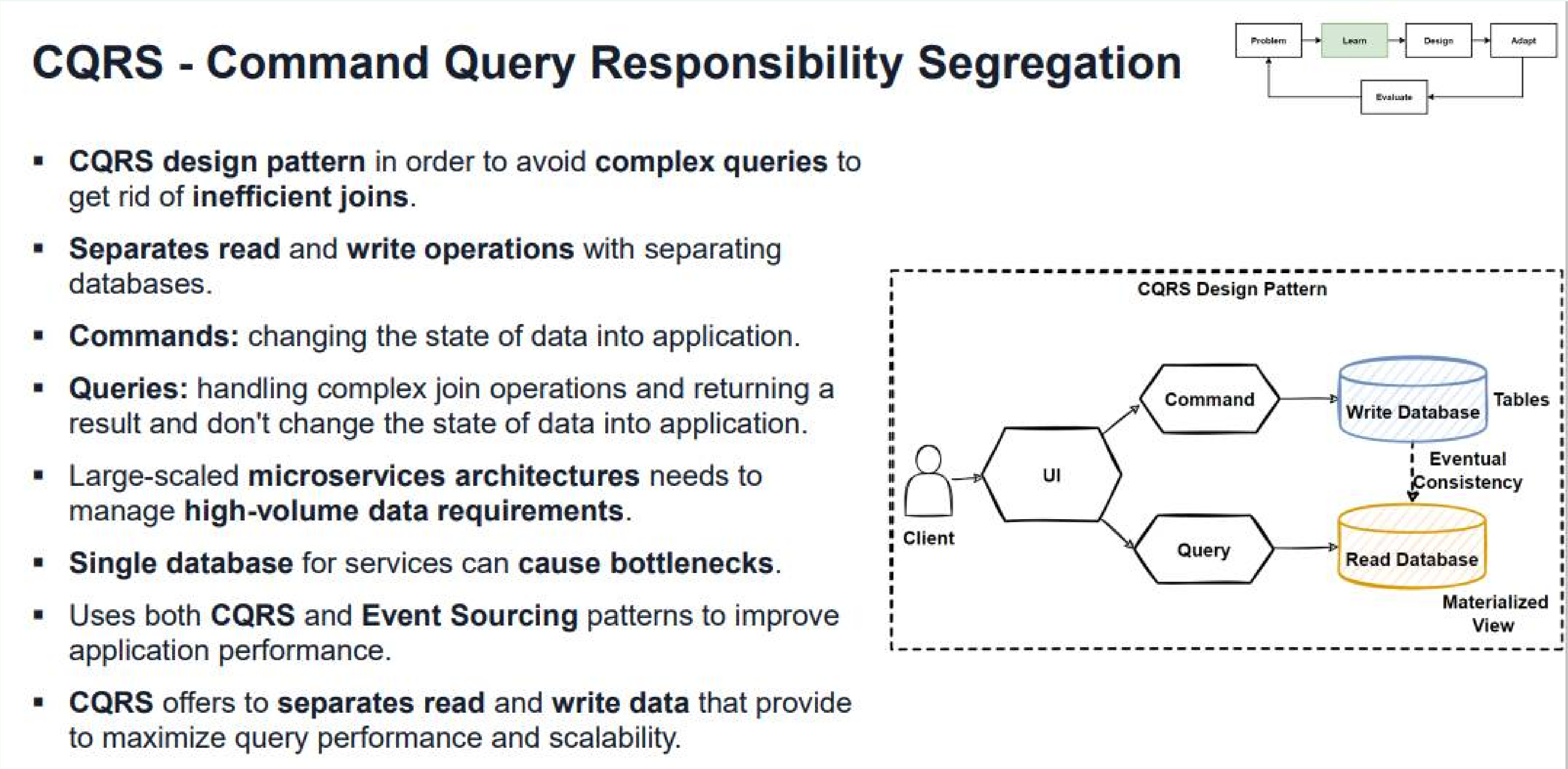
5. **Motion Graphics and Titles**: Users can create dynamic titles, captions, and lower thirds, as well as add motion graphics to enhance their projects.

6. **Multi-Cam Editing**: It supports multi-camera sequences, allowing users to edit footage from different camera angles simultaneously.

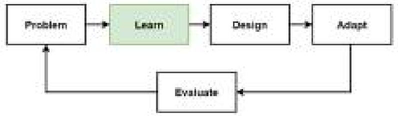
7. **Collaboration**: Premiere Pro integrates with other Adobe Creative Cloud apps, like After Effects and Photoshop, making it easy to import and export media between different software. It also has features for team collaboration on projects.

8. **Exporting for Multiple Platforms**: After editing, Premiere Pro allows exporting videos optimized for various platforms, including YouTube, social media, television, and film.

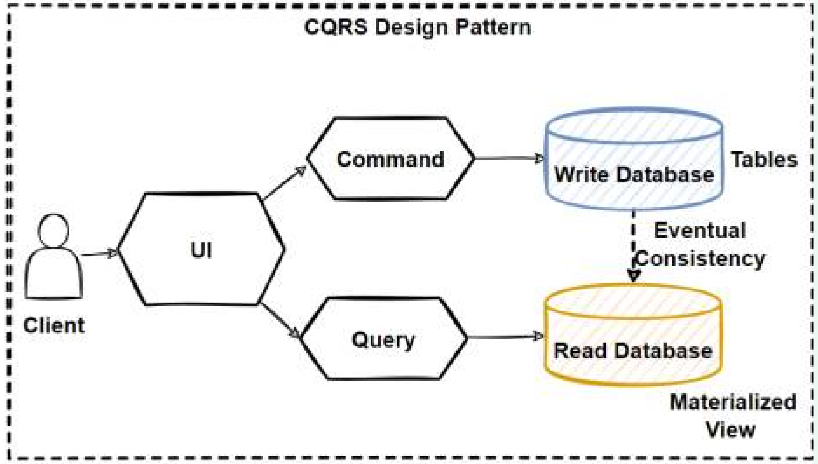
It is used by professionals across industries such as film production, broadcasting, and online content creation due to its flexibility, powerful tools, and industry-standard features.



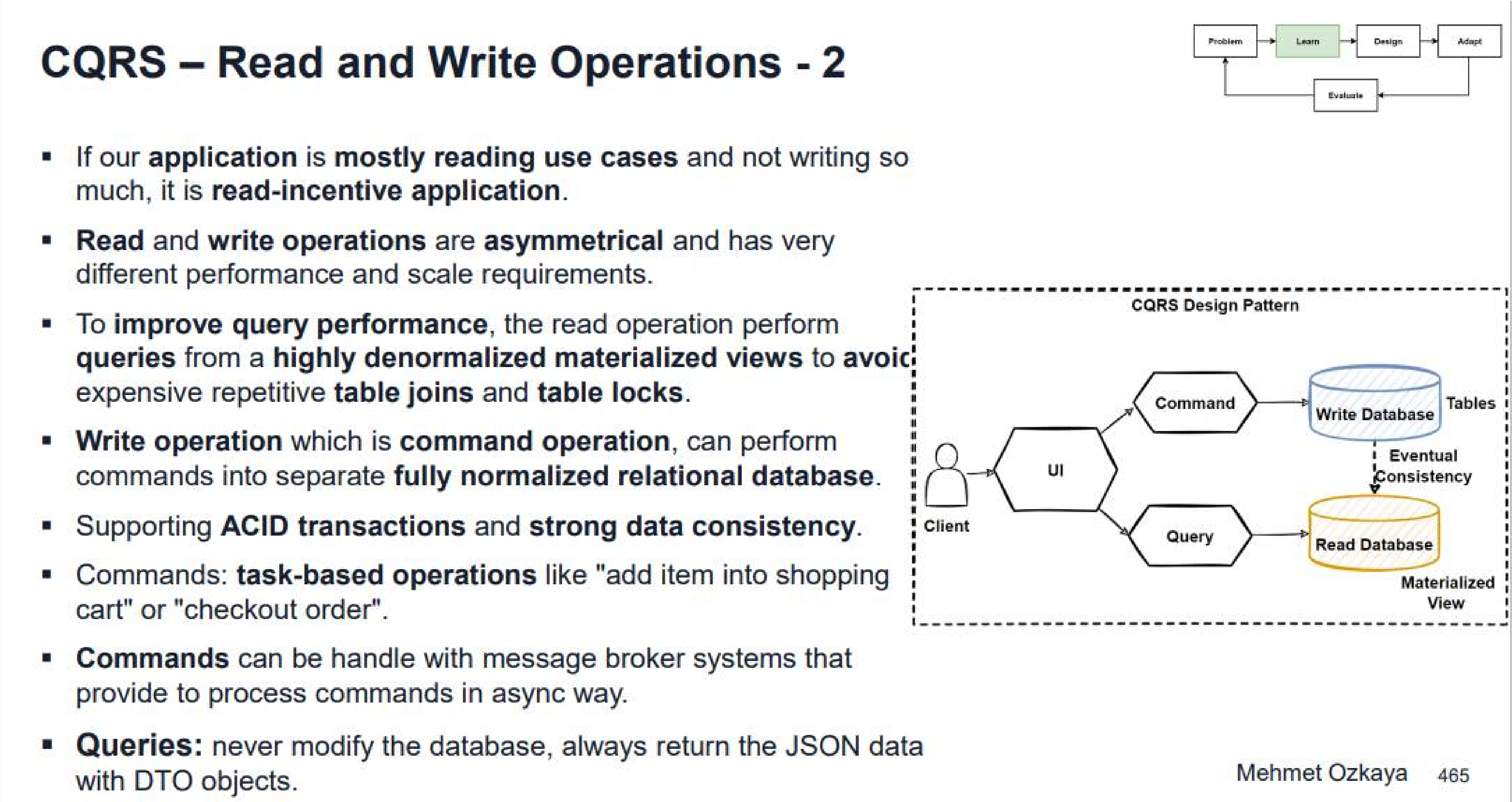
CQRS – Read and Write Operations



- **Monolithic** has **single database** is both working for **complex join queries**, and also perform **CRUD operations**.
- When **application** goes **more complex**, this **query** and **CRUD** operations will become **un-manageable situation**.
- Application required some **query** that **needs** to **join** more than **10 table**, will **lock** the **database** due to **latency** of **query computation**.
- Performing **CRUD operations** need to make **complex validations** and **process long business logics**, will cause to **lock database** operations.
- **Reading** and **writing database** has different approaches, define different strategy.
- «**Separation of concerns**» principles: separate reading database and the writing database with 2 database.
  - **Read database** uses No-SQL databases with denormalized data.
  - **Write database** uses Relational databases with fully normalized and supports strong data consistency.



Mehmet Ozkaya 464



Kindly explain the above images in details

The images you have shared provide an overview of the **CQRS (Command Query Responsibility Segregation)** design pattern, its purpose, and its implementation in handling read and write operations separately within large-scale applications, particularly microservices. Let me break down the key points for each image in detail:

Image 1: CQRS - Command Query Responsibility Segregation

- **Overview of CQRS Design Pattern**:
  - The CQRS pattern is used to handle complex queries more efficiently and avoid performance bottlenecks caused by inefficient joins.
  - It separates read operations (queries) and write operations (commands) into different databases to handle them independently, ensuring optimized performance.

- **Commands**:
  - Commands are operations that change the state of data (e.g., adding or updating records). These are sent to the write database, which may eventually propagate changes to the read database.

- **Queries**:
  - Queries are used to retrieve data without modifying the state. These can involve complex joins or aggregations, which are optimized for read performance. They retrieve data from the read database, which is often a denormalized, materialized view for faster reads.

- **Microservices and CQRS**:
  - Large-scale microservices architectures need this separation to handle high-volume data requirements and prevent bottlenecks caused by having a single database for both read and write operations.

- **Eventual Consistency**:
  - The separation of read and write databases may result in eventual consistency between them, meaning updates to the write database may take some time before being reflected in the read database.

- **Use of Event Sourcing**:
  - CQRS often integrates with Event Sourcing, where state changes are captured as events and stored sequentially, which can further enhance performance and scalability.

Image 2: CQRS - Read and Write Operations

- **Problems with Monolithic Architectures**:
  - Monolithic systems typically use a single database for both reads (queries) and writes (CRUD operations). As applications grow in complexity, this single database structure becomes inefficient, especially when performing complex joins or large queries.
- **Scaling Issues**:

  - As the number of tables and query complexity increases, database performance degrades, leading to latency issues and bottlenecks during query computation and CRUD operations.
- **Separate Databases for Read and Write**:
  - CQRS solves these problems by applying the principle of "separation of concerns." It introduces two distinct databases: one for handling write operations (CRUD) and another optimized for handling read operations (queries).
- **Read Database**:
  - The read database is typically a NoSQL or denormalized database, designed for quick retrievals of data with minimal joins.
- **Write Database**:
  - The write database remains relational, with normalized tables to support strong data consistency and ACID (Atomicity, Consistency, Isolation, Durability) transactions.

Image 3: CQRS - Read and Write Operations 2

- **Read-Heavy Applications**:
  - If an application is primarily focused on reading data rather than writing, CQRS is particularly beneficial. It allows the application to handle the heavy read load by using a highly denormalized, materialized view for quick query performance.
- **Asymmetrical Read and Write Operations**:
  - CQRS recognizes that read and write operations have different performance characteristics and scale requirements. Reads are optimized for speed, while writes are optimized for consistency and integrity.
- **Improving Query Performance**:
  - The read database can avoid complex table joins by using precomputed, highly denormalized views, which drastically improve performance and reduce the need for expensive database operations.
- **Write Operations**:
  - Commands that change the state of data are executed in the write database, which is fully normalized and optimized for strong data consistency, following ACID principles.
- **Supporting Business Logic**:
  - Write operations also involve executing complex business logic, like processing long transactions or validating data, which can lock database operations in monolithic systems. CQRS avoids this by separating write concerns.
- **Task-Based Commands**:
  - Commands can be task-based, such as adding an item to a shopping cart or completing a checkout. These commands are processed asynchronously, often handled by message brokers, ensuring they don’t interfere with read operations.
- **Queries and DTOs**:
  - Queries are designed to return read-only data (like DTOs – Data Transfer Objects) without modifying the database. This ensures that the read and write responsibilities remain separate.

Key Takeaways from the Images:

1. **Separation of Read and Write**: The core principle of CQRS is the separation of read (query) and write (command) operations into different databases to improve scalability and performance.

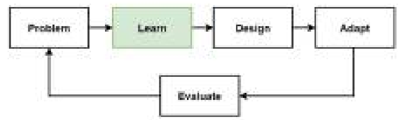
2. **Handling Complexity**: As applications become more complex, traditional monolithic architectures with a single database face performance bottlenecks due to complex joins, CRUD operations, and locking issues. CQRS helps to alleviate these issues by distributing responsibilities.

3. **Eventual Consistency**: With two separate databases, data changes in the write database may take time to propagate to the read database, leading to eventual consistency between the two.

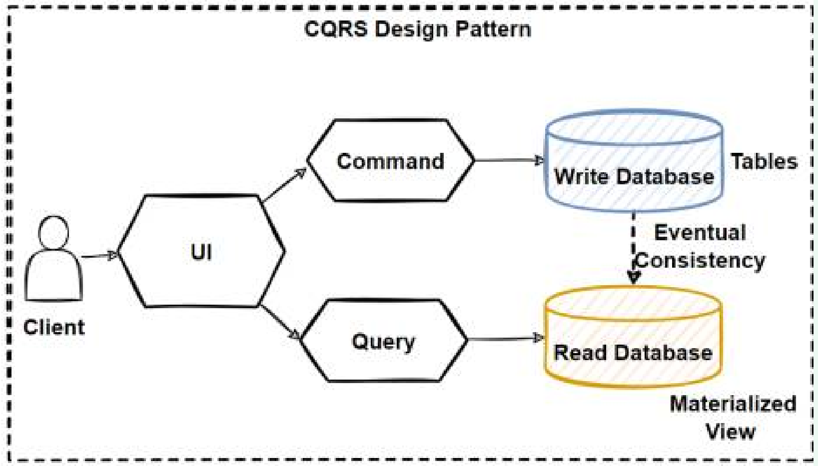
4. **Optimization for Scale**: CQRS, often combined with Event Sourcing, is particularly useful in large-scale, distributed systems where scalability, performance, and flexibility are critical.

Would you like to dive deeper into any of these topics or explore a specific use case?

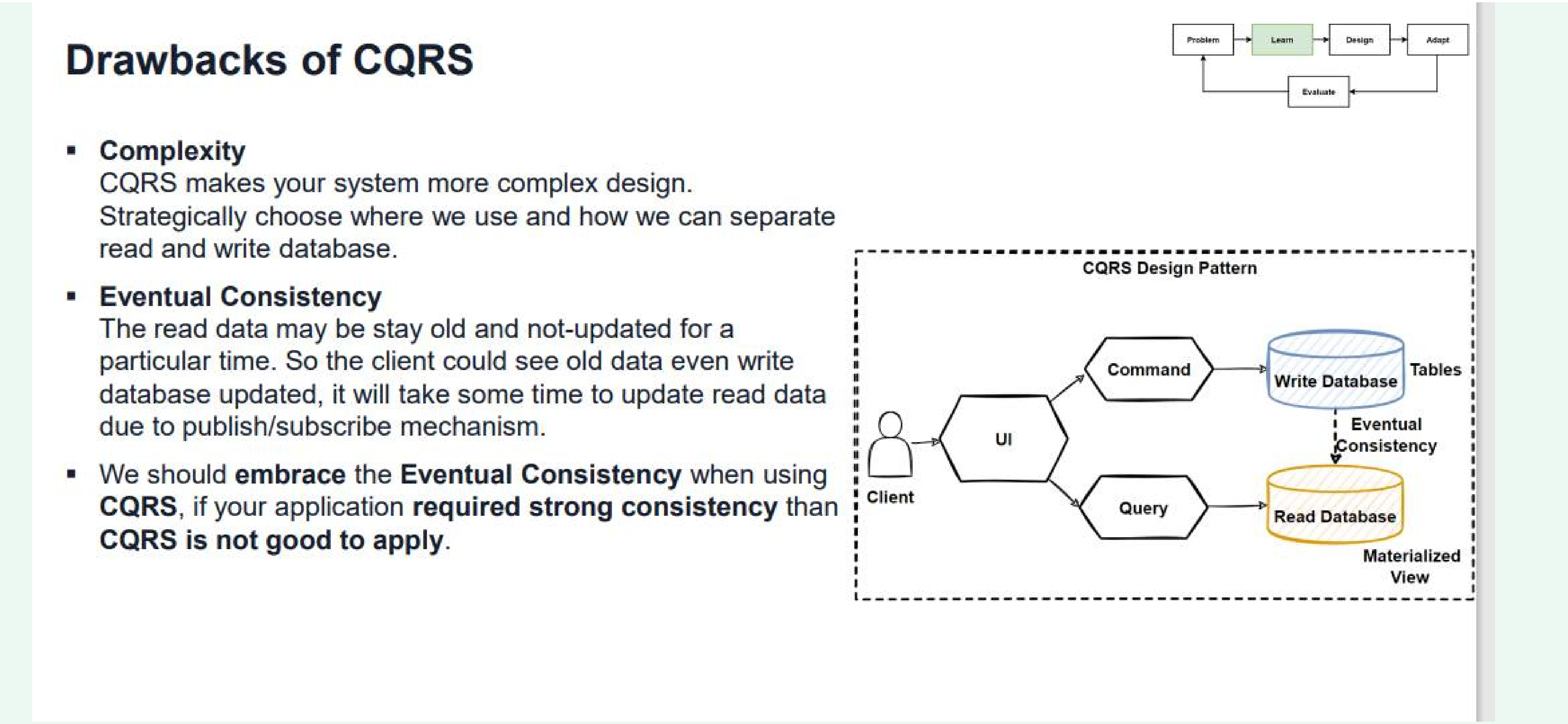
Benefits of CQRS



- **Scalability**
  When we separate Read and Write databases, we can also scale these separate databases independently. Read databases follows denormalized data to perform complex join queries.
- If application is read-incentive application, we can scale read database more then write database.
- **Query Performance**
  The read database includes denormalized data that reduce to comlex and long-running join queries. Complex business logic goes into the write database. Improves application performance for all aspects.
- **Maintability and Flexibility**
  Flexibility of system that is better evolve over time and not affected to update commands and schema changes by separating read and write concerns into different databases.
- Better implemented if we physically separate the read and write databases.



Mehmet Ozkaya 467



Kindly explain above in details

The new images you've uploaded provide insights into the **Benefits** and **Drawbacks** of the CQRS (Command Query Responsibility Segregation) pattern. Let's break them down in detail:

Benefits of CQRS

1. **Scalability**:
    - By separating the read and write operations into different databases, it becomes easier to scale each independently based on the load they handle.
    - For example, if the application is read-heavy, you can scale the read database independently to handle more queries without affecting the write operations.
    - Similarly, you can optimize the write database for transactional operations and scale it based on the system's need for write-heavy operations.

2. **Query Performance**:
    - The read database contains **denormalized data**, which allows for faster query performance, especially when dealing with complex joins and long-running queries.
    - Since complex business logic is handled by the write database (in a fully normalized format), the separation ensures that each part of the system is optimized for its specific purpose, leading to better overall performance.
    - This separation eliminates the bottlenecks caused by traditional CRUD (Create, Read, Update, Delete) operations in a single database setup.

3. **Maintainability and Flexibility**:
    - CQRS offers increased flexibility, especially in systems that need to evolve over time. By having separate databases for reading and writing, you can update the schema or commands without impacting the entire system.
    - It allows for better evolution of commands and queries over time since the concerns are isolated. Developers can update one side (either read or write) without affecting the other.
    - This architecture can be more easily maintained and adapted to growing needs, making it suitable for complex, long-term projects.

Drawbacks of CQRS

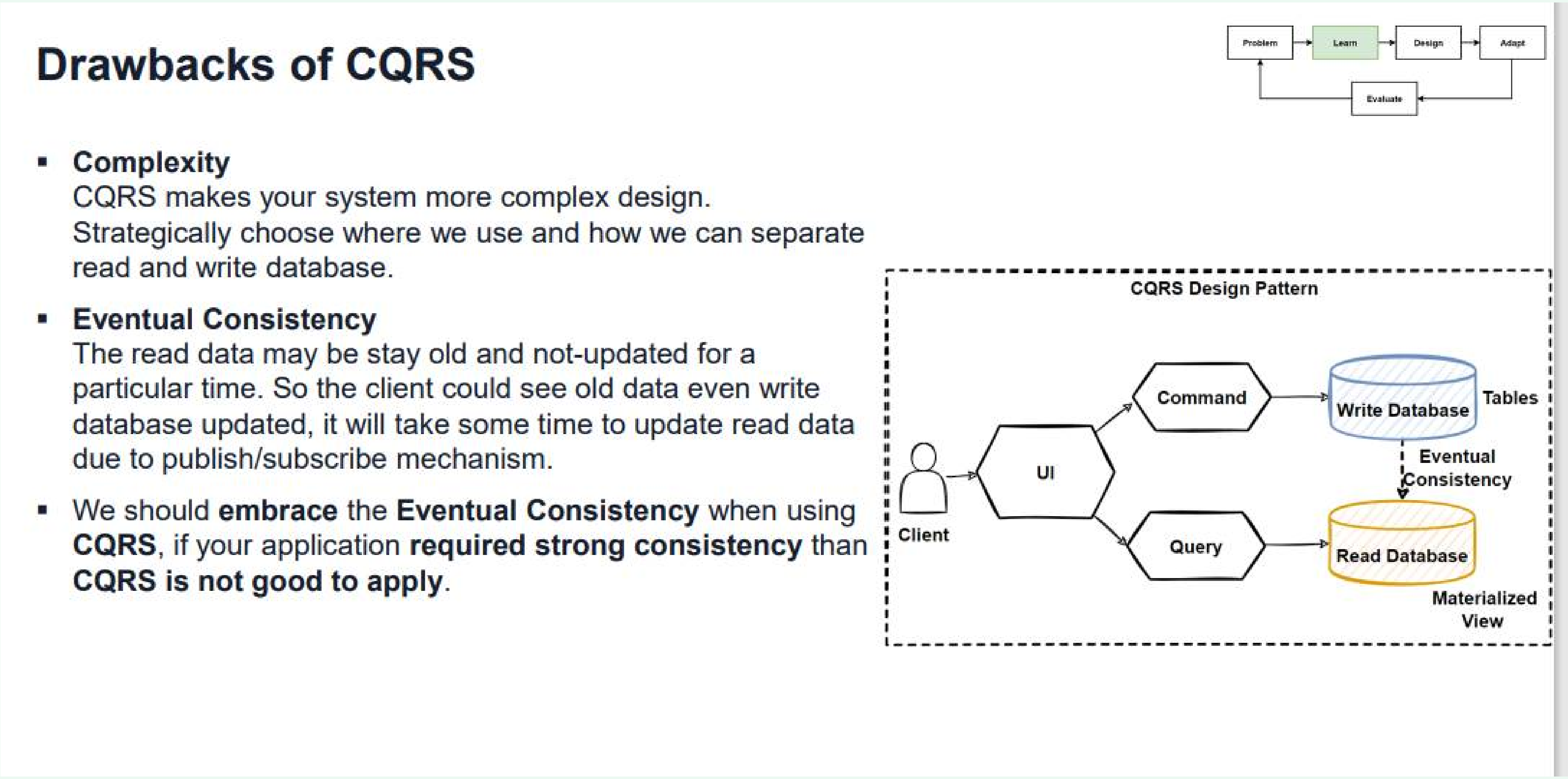
1. **Complexity**:
    - Implementing CQRS adds complexity to the system. You now have two databases (read and write), and managing them requires careful design choices and more infrastructure.
    - It’s important to strategically decide where to apply CQRS. Not all applications benefit from this level of complexity. It’s mostly useful for systems with high performance and scaling requirements, but smaller applications may not need it.
2. **Eventual Consistency**:
    - One of the key trade-offs of CQRS is **eventual consistency**. Since the read and write databases are separate, changes in the write database may take some time to reflect in the read database. This delay can result in the client seeing stale data temporarily.
    - The delay occurs because updates from the write database are propagated to the read database through a publish/subscribe mechanism, which can take time.

  - If your application requires **strong consistency** (where changes are immediately visible to users), CQRS may not be a suitable choice. In such cases, having a unified system where read and write operations happen in the same database may be more appropriate.

Key Takeaways from the Benefits and Drawbacks:

- **Scalability and Performance**: CQRS allows for optimized performance and scaling by separating read-heavy and write-heavy workloads, making it ideal for large-scale systems.
- **Complexity vs. Consistency**: While CQRS offers better performance and flexibility, it comes with the trade-off of increased complexity and potential issues with eventual consistency. Applications requiring immediate consistency might need a different approach.

Would you like further details on how CQRS can be applied or in what types of systems it's most beneficial?



Kindly explain this in details

The image you've shared highlights two major **Drawbacks of CQRS**: **Complexity** and **Eventual Consistency**. Let's dive into both in detail:

1. Complexity:

- **System Complexity**:
  - CQRS inherently introduces complexity into the system. This is because instead of managing a single database that handles both reads and writes, you are now dealing with two separate systems—one for handling commands (writes) and another for handling queries (reads).
  - Maintaining two different data models (one for reading and one for writing) means that developers need to ensure that the flow between the two remains consistent, even though they are independent systems.
  - Additional infrastructure and mechanisms are required to keep these two databases in sync, such as event handling systems, publish/subscribe mechanisms, or message brokers. This complexity increases the development and operational overhead.
  - Strategic decision-making is necessary when adopting CQRS, especially in determining **where** and **how** to apply this pattern. It's not always suitable for every part of an application. Applying it without careful thought can over-complicate the architecture and lead to more difficult maintenance in the long run.

2. Eventual Consistency:

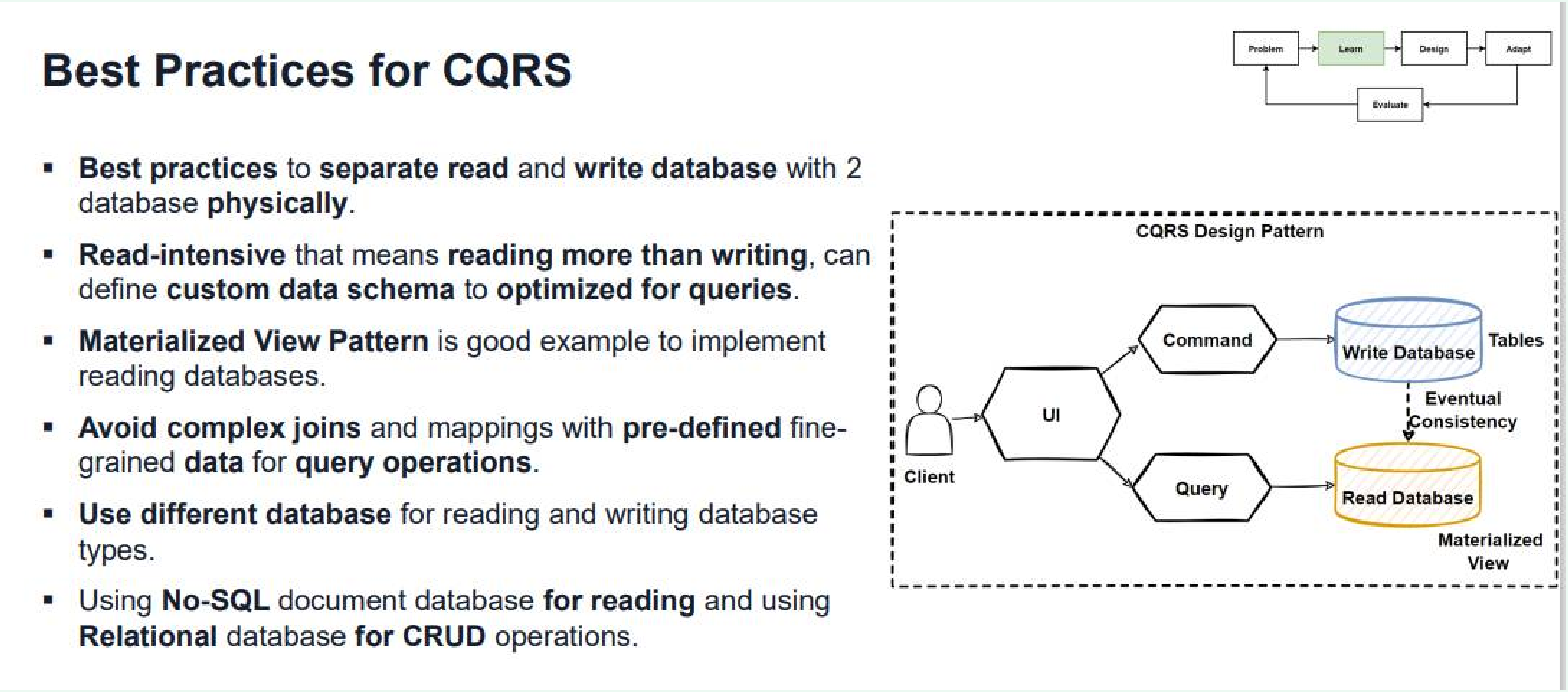
- **What is Eventual Consistency?**
  - In a CQRS system, the write operations (commands) happen on the **write database**, while the read operations (queries) happen on the **read database**. The changes made to the write database are not instantly reflected in the read database.
  - This delay in synchronization results in what is known as **eventual consistency**. It means that at any given point in time, the data in the read database may be slightly outdated compared to the write database.
- **Impact of Eventual Consistency**:
  - A client querying the system might get **old or stale data**, even though the write database has been updated. The client will only see the updated data once the synchronization process (often using event propagation via a publish/subscribe mechanism) completes and the read database gets the updated information.
  - This can be problematic for applications that require **strong consistency**, where the user needs to see the most up-to-date information immediately after it's written. For example, in a financial transaction system, seeing delayed information could lead to incorrect decisions.
- **When Eventual Consistency is Acceptable**:
  - In many systems, a slight delay in consistency is acceptable, especially if the system prioritizes high performance and scalability over strict consistency. For example, in social media applications, a slight delay in post visibility might not be critical, so eventual consistency can work.
  - However, in applications that require **strong consistency**, like banking systems, CQRS might not be suitable. If you cannot tolerate this temporary inconsistency, then using a single database model or an alternative approach may be necessary.

Key Takeaway:

- **Complexity** and **Eventual Consistency** are inherent challenges with the CQRS pattern. It is crucial to carefully evaluate whether the trade-offs are worth it for your particular application.

- If your application requires strong, immediate consistency, CQRS might not be the best option, as it is designed with the assumption that eventual consistency will be acceptable.
- On the other hand, for high-performance, read-intensive applications, CQRS can be a powerful pattern, allowing you to optimize read and write operations independently.

Would you like more examples of where CQRS is a good fit or how to mitigate some of these challenges?



Kindly explain this in details

The image you've shared provides insights into **Best Practices for CQRS** (Command Query Responsibility Segregation). These best practices are critical for implementing CQRS effectively while ensuring performance, scalability, and maintainability. Let's break it down:

1. Separate Read and Write Databases Physically:

- **Best Practice**: It's recommended to physically separate the databases for read and write operations. This separation allows the system to optimize each for its respective tasks (read or write), ensuring scalability and reducing potential performance bottlenecks.
- **Why Important?**: By separating the databases, you avoid conflicts between read-heavy and write-heavy operations, allowing each database to be scaled independently based on its workload.

2. Read-Intensive Applications:

- **Best Practice**: If your application is **read-intensive** (i.e., the system handles more read queries than write operations), you should design the data schema specifically for optimizing read queries.
- **Custom Data Schema**: You can create a data schema in the read database that is highly optimized for querying, without worrying about the complexities introduced by write operations (such as normalization). This schema can include denormalized tables, precomputed values, and indexes that make querying faster and more efficient.
- **Why Important?**: Optimizing for reads is essential in applications where the read-to-write ratio is skewed towards reads, such as in reporting systems or dashboards.

3. Materialized View Pattern:

- **Best Practice**: Implement the **Materialized View Pattern** in the read database. This means creating a precomputed and stored query result (the materialized view) that can be updated periodically or based on events from the write database.
- **Why Important?**: Materialized views help avoid the cost of performing complex joins or aggregations at query time, as the view is already computed and ready to be queried. This drastically improves the performance of read operations.
- **Example**: In a financial application, instead of recalculating a user's account balance on every query (which may involve multiple transactions and complex joins), the balance is stored in a materialized view and quickly retrieved.

4. Avoid Complex Joins and Mappings:

- **Best Practice**: Avoid using complex joins and mappings in the read database by creating **pre-defined fine-grained data**. This means structuring the read database to minimize the need for joins or expensive queries.
- **Why Important?**: Complex joins can degrade query performance, especially when dealing with large datasets. Pre-defining the structure of your read database with denormalized tables ensures faster querying without expensive operations like joins.

- **Example**: Instead of performing a join between orders and customer tables, you can create a single table that contains all necessary data for orders and customers, reducing the need for joins.

5. Use Different Databases for Reading and Writing:

- **Best Practice**: Utilize different types of databases for reading and writing operations to maximize their efficiency. The choice of database should be tailored to the specific needs of the read or write operations.
- **Why Important?**: Using the same database technology for both read and write operations may not always be optimal. Each type of operation has different performance characteristics and requirements.
- **Examples**:
    - For **read** operations, where high performance and quick access are required, a **NoSQL database** (such as MongoDB or Cassandra) might be a better fit due to its ability to handle large volumes of denormalized data and its schema flexibility.
    - For **write** operations, where strong consistency and ACID transactions are important, a **Relational Database** (such as MySQL or PostgreSQL) is more suitable because it provides strong consistency, data integrity, and transactional support for CRUD operations.

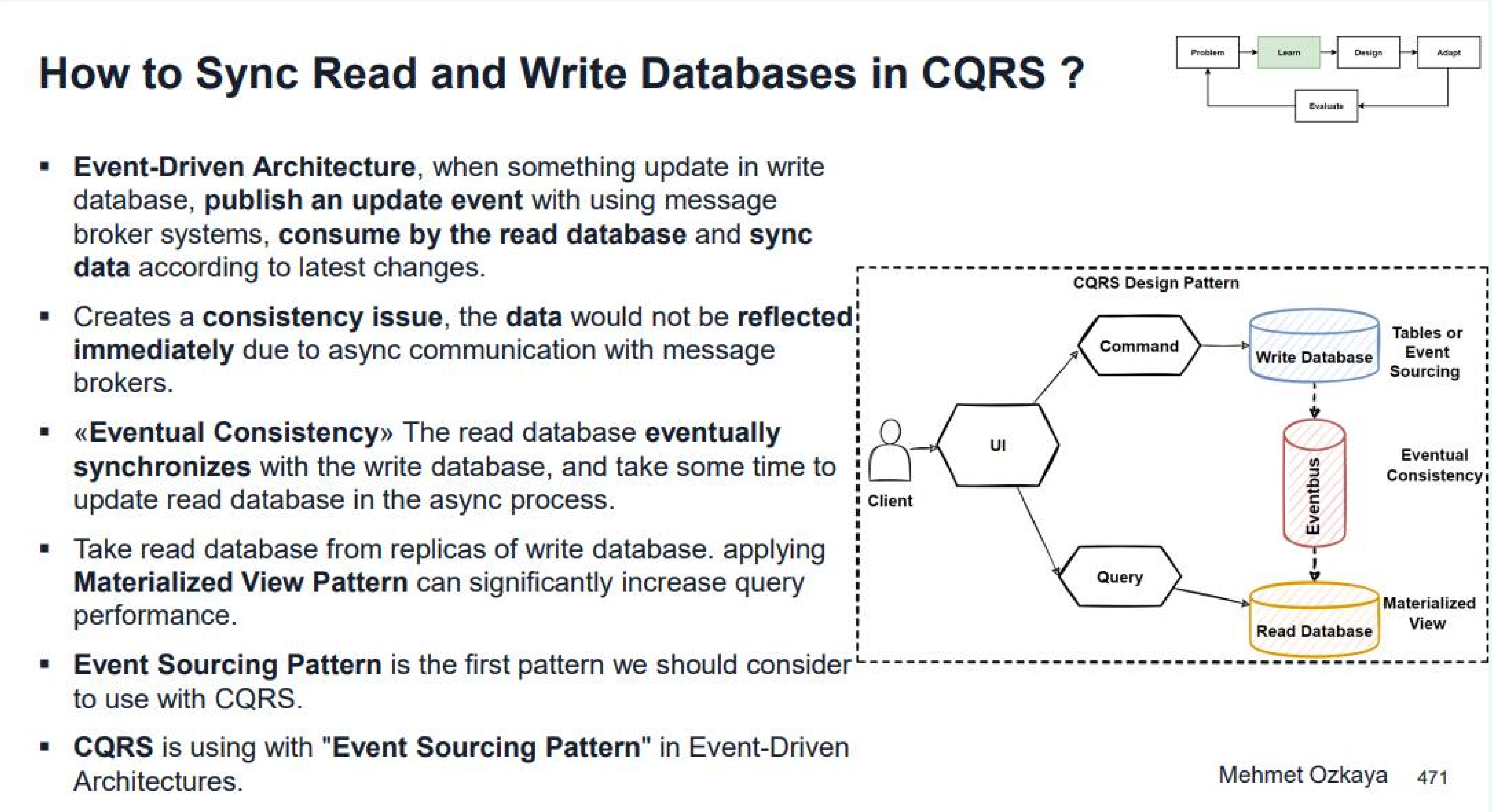
6. NoSQL for Reads, Relational Database for Writes:

- **Best Practice**: Use a **NoSQL document database** for the read operations and a **relational database** for the write operations (CRUD).
- **Why Important?**: This allows the system to take advantage of the strengths of each database type:
    - **NoSQL databases** like MongoDB are optimized for fast, scalable reads with denormalized data, making them ideal for read-heavy use cases.
    - **Relational databases** like MySQL or PostgreSQL are optimized for data integrity and transactional consistency, making them the right choice for write-heavy operations.
- **Example**: In an e-commerce system, user product views and product recommendations (read-heavy operations) can be handled by a NoSQL database, while order processing and inventory management (write-heavy, transactional operations) can be managed by a relational database.

Key Takeaways:

- **Separation of Databases**: Physically separating the read and write databases allows each to be optimized for its specific workload and scaled independently.
- **Optimizing Reads**: For read-intensive applications, customize the read database schema for fast querying, potentially using denormalized data and materialized views to improve performance.
- **Avoiding Complexity**: Avoiding complex joins and utilizing pre-defined, fine-grained data structures ensures fast read operations.
- **Tailored Databases**: Choosing different databases for reads (NoSQL) and writes (Relational) can enhance the overall performance and efficiency of the system, leveraging the strengths of both database types.

Would you like further clarification on how to implement some of these best practices in a specific use case?



Kindly explain it in details

The image you've shared explains **how to synchronize the read and write databases in CQRS (Command Query Responsibility Segregation)**, focusing on using **event-driven architecture** and **event sourcing**. Here's a detailed breakdown of each concept:

1. Event-Driven Architecture:

- **How it works**:
  - In an event-driven architecture, whenever a change or update is made to the **write database** (such as an insert, update, or delete operation), the system **publishes an update event**. This event represents the change that occurred in the write database.
  - This update event is handled using **message broker systems** (such as Kafka, RabbitMQ, etc.). The message broker acts as an intermediary, delivering the event to other services that need to be aware of the change.
  - The **read database** listens for these events and consumes them. Based on the received events, the read database **synchronizes** itself by updating the necessary records according to the latest changes in the write database.

- **Why Important?**:
  - This architecture allows the system to handle data changes asynchronously. Instead of blocking the user interface (UI) until both databases are synchronized, the system updates the read database in the background after a change has been committed to the write database.

2. Consistency Issue:

- **Problem**:
  - One major downside of this approach is that the **data will not be reflected immediately** in the read database due to the asynchronous communication between the write and read databases via the message broker.
  - This means that there is a time gap between when the write database is updated and when the read database reflects those changes.
- **Why Important?**:
  - This delay leads to **eventual consistency** rather than immediate consistency. Users may temporarily see outdated data in the read database, which can cause issues in applications that require real-time accuracy.

3. Eventual Consistency:

- **What is Eventual Consistency?**:
  - **Eventual consistency** means that while the system will eventually synchronize, there will be a delay in getting the updated information from the write database to the read database.

- The read database will synchronize with the write database **asynchronously**, and it takes time for the update event to be processed and the read database to be updated.
- **Why Important?**:
  - In systems where this small delay in consistency is acceptable, this model works well. However, if **strong consistency** is required (i.e., you need the data to be updated immediately in both databases), this approach may not be suitable.

4. Materialized View Pattern:

- **Materialized Views**:
  - To improve query performance, a **materialized view** is often applied in the read database. This materialized view is a precomputed, stored version of the query result. It is updated periodically or based on events.
  - Instead of performing expensive queries at runtime, the system can pull precomputed data from the materialized view, which is significantly faster and reduces the load on the database.
- **Why Important?**:
  - By using materialized views, the system improves read performance by eliminating the need to execute complex joins or aggregations on the fly. This is particularly useful in high-read scenarios where query performance is critical.

5. Event Sourcing Pattern:

- **How it Works**:
  - The **Event Sourcing Pattern** is an approach where every state change (event) in the system is stored as a sequence of events, rather than simply storing the final state in a traditional database.
  - Each event represents a specific change to the state, and the system can reconstruct the current state of the database by replaying all of these events.
- **Why Important?**:
  - Event sourcing provides a robust mechanism for **auditability**, as every change is recorded as an event. It also allows the system to handle **compensation** and rollback scenarios since the sequence of events is preserved.
  - When combined with CQRS, event sourcing ensures that both the read and write databases are consistently updated based on the same event stream, which is the foundation of the synchronization between the two databases.

6. Using Event Sourcing in CQRS:

- **CQRS with Event Sourcing**:
  - CQRS is often implemented alongside the **Event Sourcing Pattern**. This means that the write operations are captured as events and stored in an event store. The read side then listens for these events and updates the read database accordingly.

  - By using this pattern, the write database stores the sequence of events that represent changes to the system. The read database (or materialized view) can then be updated based on these events to ensure that both the read and write sides are consistent (eventually).

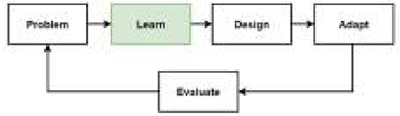
- **Why Important?**:
  - This pattern ensures that the synchronization between the read and write databases is consistent, and any changes in the system are reliably captured and propagated.
  - It also allows for scalability since the events can be processed asynchronously, and different components (such as the read database) can react to those events at their own pace without blocking the system.

Key Takeaways:

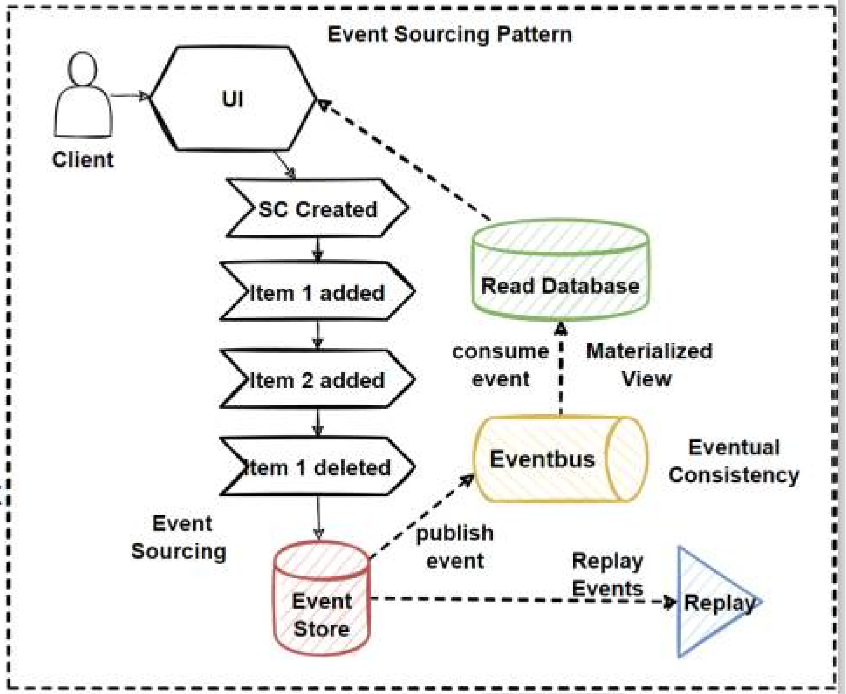
- **Event-Driven Synchronization**: Updates in the write database are communicated to the read database asynchronously through an event-driven architecture using message brokers.
- **Eventual Consistency**: There is a delay between when the write database is updated and when the read database is synchronized, leading to eventual consistency, which may be acceptable depending on the application's needs.
- **Materialized Views**: Using precomputed materialized views in the read database helps optimize query performance, especially when handling large-scale data and complex queries.
- **Event Sourcing**: This pattern captures all changes as events, providing a reliable and scalable mechanism for synchronizing the read and write databases in a CQRS architecture.

Would you like more details on how to implement event sourcing or how it integrates with message brokers in CQRS?

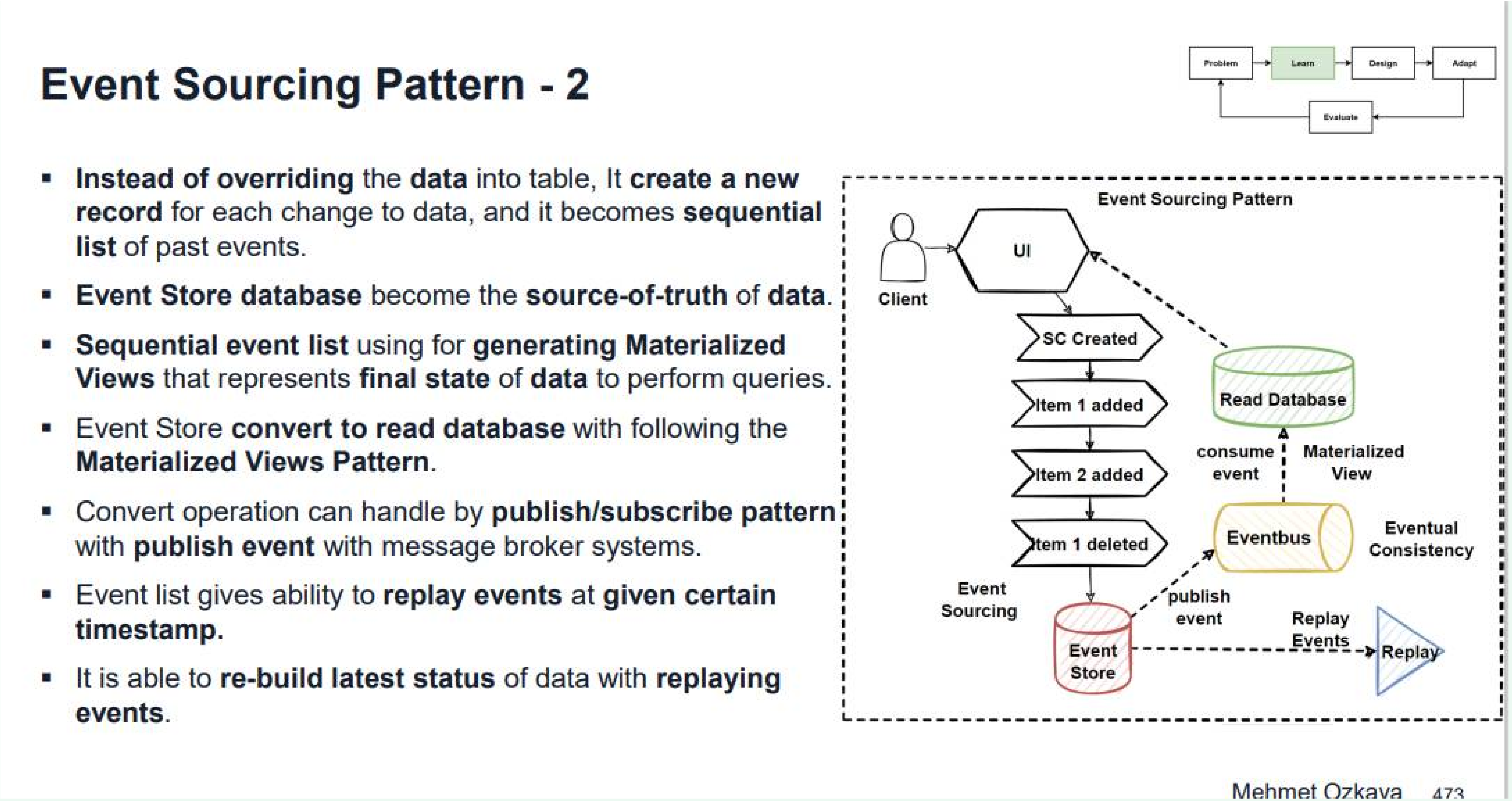
Event Sourcing Pattern



- Most applications **saves data** into databases with **the current state of the entity**. I.e. user change the email address table, email field updated with the latest updated one. Always know the **latest status** of the **data**.
- In large-scaled architectures, **frequent update database** operations can **negatively impact database performance**, **responsiveness**, and **limits** of **scalability**.
- **Event Sourcing pattern** offers to persist each action that affects to data into **Event Store database**. And call all these **action** as a **event**.
- **Instead of saving latest status** of data into database, Event Sourcing pattern offers to **save all events into database** with **sequential ordered** of data events.
- This **events database** called **Event Store**.



Mehmet Ozkaya 472



Kindly explain this in details

The images you've shared provide a detailed explanation of the **Event Sourcing Pattern**, a crucial architectural pattern often used in combination with CQRS (Command Query Responsibility Segregation) to capture and store changes in data as a series of events. Let's break it down:

Event Sourcing Pattern

1. **Traditional Data Storage (Current State) vs. Event Sourcing**:

    - **Traditional Approach**:

        - In most applications, data is stored in its **current state**. For example, if a user updates their email, the system will overwrite the old email with the new one, and the database only reflects the latest state.
        - This approach allows us to always see the **latest status** of the data, but we lose historical information about **how the data changed** over time.

    - **Event Sourcing Approach**:

        - Instead of storing just the latest state, the **Event Sourcing pattern** stores each change as a separate **event** in an **Event Store**. These events represent each action that affects the state of an entity. Examples include events like "Email Changed," "Item Added to Cart," etc.
        - Every action or change to the data is stored in a **sequential, ordered list of events**, meaning you can reconstruct the full history of how the data evolved.

2. **Benefits of Event Sourcing**:

    - **Improved Scalability and Performance**:
        - In large-scale systems, frequent updates to the database can negatively impact performance. Event sourcing helps by storing immutable events instead of constantly overwriting records. This reduces contention and locking issues.
    - **Auditability**:
        - Since every action is stored as an event, it provides a complete audit trail of how the data has changed over time. This is highly useful for systems that require a detailed history of transactions or changes (e.g., financial systems).
    - **Rebuild State**:
        - The entire history of events allows you to **rebuild the latest state** of any entity by replaying the events in the correct order. This can be useful for debugging, auditing, or even recreating lost data.

How Event Sourcing Works (Diagram Explanation):

- **Client Interaction**:

    - The **client** interacts with the system by performing actions (like creating or updating data), which in turn triggers events.

- **Event Store**:

    - Instead of directly saving the latest state of the data in a traditional database, each action is recorded as an **event** and stored in a specialized database called the **Event Store**.

- The **Event Store** becomes the **source-of-truth** for all changes in the system. Every state change is stored as a sequential event (for example: "Item Added," "Item Deleted").

- **Eventbus and Event Publication**:
  - When an event occurs (such as an item being added to a shopping cart), it is published to the **Eventbus**. The event bus handles the communication between different parts of the system.
  - The event is then consumed by other services, such as the read database, which listens to these events and updates its data accordingly.

- **Read Database**:
  - The **read database** may use a **Materialized View Pattern** to consume the events and update the state of its data. This materialized view represents the current state of the data after applying all the events.
  - This is particularly useful in CQRS systems, where the read database is optimized for fast queries, and updates are handled by asynchronously consuming events from the write side.

- **Replay Events**:
  - Since all events are stored in the Event Store, they can be **replayed** at any point in time to rebuild the current state of the data. For example, if you need to recover lost data or analyze how the data changed at a specific point in time, you can replay the events sequentially to reconstruct the state.
  - This feature is especially helpful for scenarios like auditing, data recovery, or debugging.

Event Sourcing Pattern - 2

1. **Sequential Event List**:

   - Instead of overwriting data with each change, the system records every change as a new event, creating a **sequential list of events**. This sequential list becomes a permanent record of everything that has happened to an entity.

   - **Why Important?**: This sequential event list is critical because it allows you to recreate the entity's state at any point in time by simply replaying the events in order. This also allows generating **Materialized Views** that represent the **final state** of the data after all events are applied.

2. **Publish/Subscribe Mechanism**:

   - Changes are propagated to other parts of the system via the **publish/subscribe pattern**. When an event is stored in the Event Store, it is published on the Eventbus. Systems or services subscribed to the Eventbus consume the event and update their state accordingly.

- **Why Important?**: This mechanism ensures that various parts of the system (like the read database in a CQRS system) stay updated as new events are published.

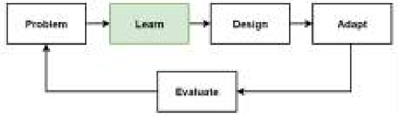
3. **Replaying Events**:
   - **Replay** capability allows you to **rebuild the latest state** of the system by processing all stored events in sequence. This is useful for:
     - **Rebuilding data** in case of failures or data loss.
     - **Auditing** changes over time to understand how the data evolved.
     - **Debugging** issues by replaying events leading up to the problem.
   - **Why Important?**: Replaying events makes it possible to recreate the state of an entity at any point in time, allowing for better diagnostics and fault recovery.

Summary of Key Points:

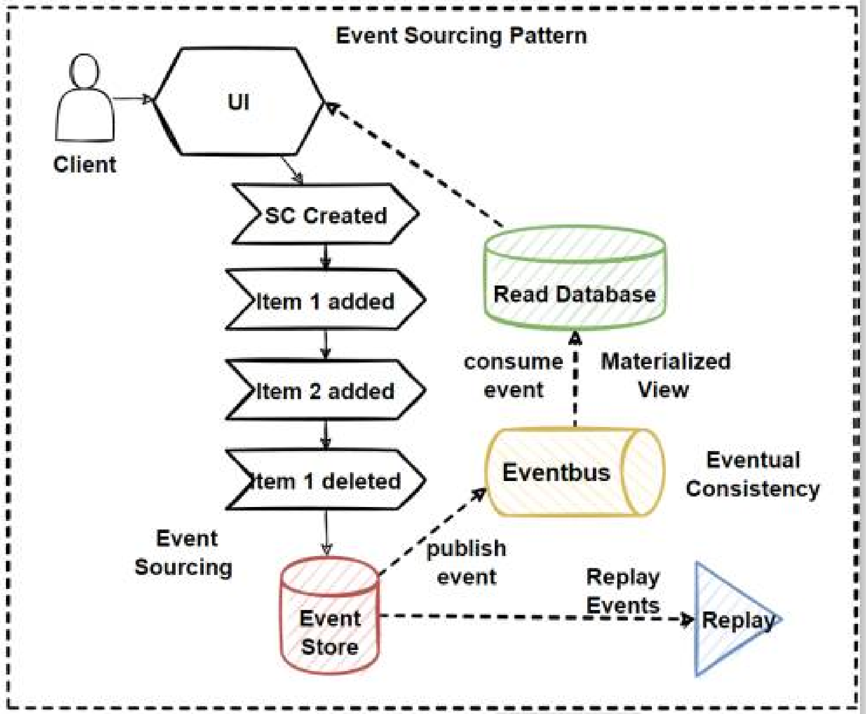
- **Event Sourcing** stores each change to an entity as a discrete event, capturing the full history of changes over time.
- The **Event Store** becomes the source-of-truth, allowing you to audit, replay, and rebuild the state of the system at any time.
- **Sequential events** make it easy to regenerate the current state or past states by simply replaying events.
- **Materialized Views** are created by consuming the event stream, allowing the read database to stay in sync with the events published by the Eventbus.
- **Publish/Subscribe** mechanisms ensure asynchronous communication between the write side (Event Store) and the read side (materialized views or read database).

Would you like to know more about implementing event sourcing in a specific system or combining it with CQRS?

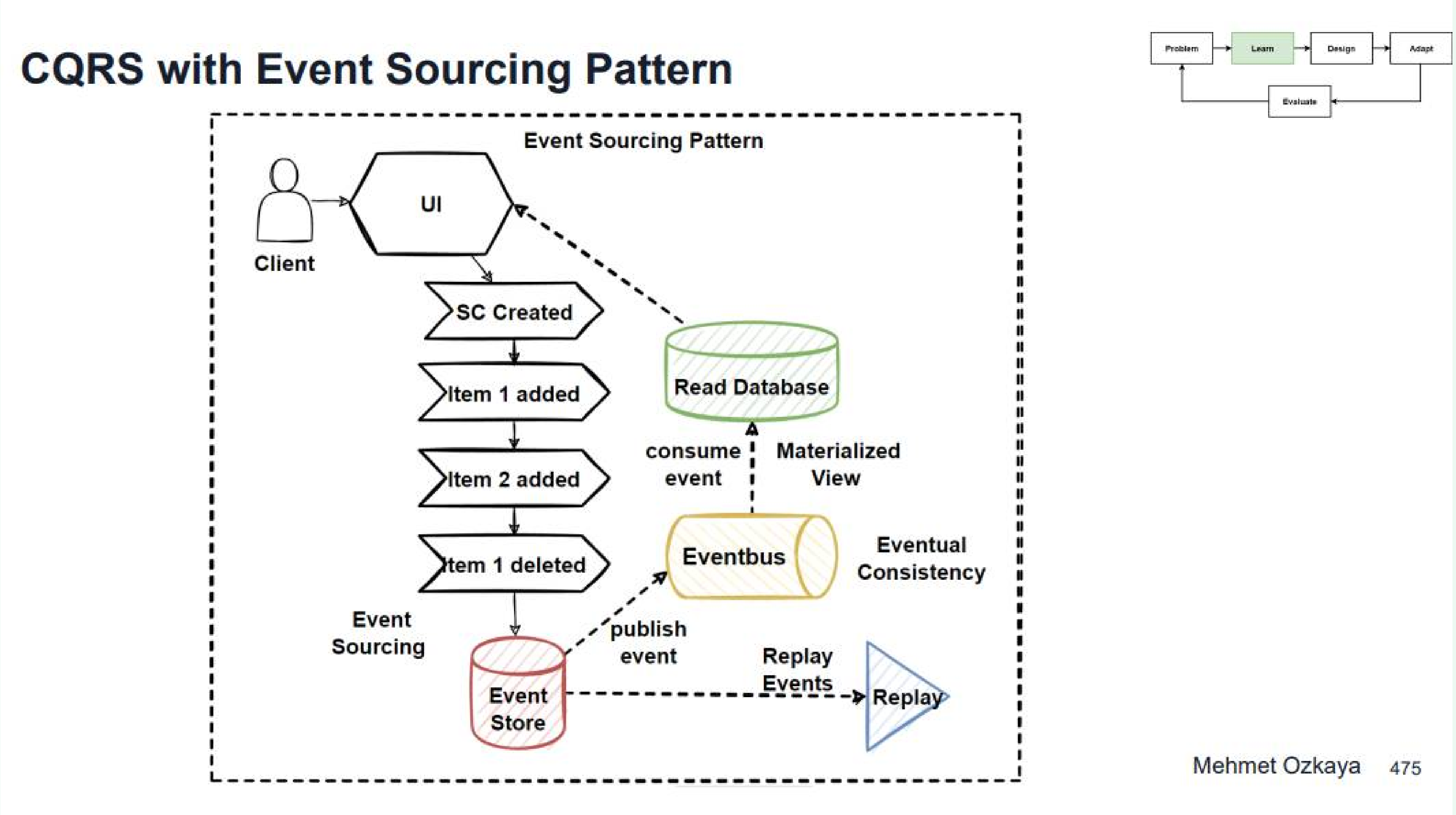
CQRS with Event Sourcing Pattern



- **CQRS** pattern is mostly **using with** the **Event Sourcing pattern.**
- **Store events** into the **write database**; **source-of-truth events** database.
- **Read database** of CQRS pattern provides **materialized views** of the data with **denormalized tables**.
- **Materialized views** read database **consumes events** from write database **convert them** into **denormalized views**.
- The **writing database** is **never save status** of data only events actions are stored.
- **Store history of data** and able to **reply any point of time** in order to **re-generate status** of data.
- System can **increased query performance** and **scale databases independently.**



Mehmet Ozkaya 474



Kindly explain the above images in details

The images you've shared explain how **CQRS (Command Query Responsibility Segregation)** can be used in conjunction with the **Event Sourcing Pattern**. Let's break down the explanation step by step, focusing on the key concepts and how they work together.

1. CQRS and Event Sourcing Pattern Overview:

- **CQRS Pattern**:
  - CQRS separates the responsibilities of commands (write operations) and queries (read operations) into distinct models and databases. This separation helps in optimizing both read and write performance for complex systems.

- **Event Sourcing**:
  - Instead of storing just the current state of the data in the database, Event Sourcing stores every change to the state as an event in an **Event Store**. This way, the history of changes is preserved, and the state can be rebuilt by replaying the events.
  - This combination of CQRS and Event Sourcing helps in scaling systems and improving performance.

2. How CQRS Works with Event Sourcing:

- **Write Database with Event Store**:
  - Instead of writing the current state to a database, every change is saved as an event to the **Event Store**, which acts as the source of truth. These events represent every change made to the data (e.g., an item added, deleted, etc.).
  - The **Event Store** stores each of these events in a sequential, immutable way. These events can later be used to reconstruct the state by replaying them.

- **Read Database with Materialized Views**:
  - The **read database** in CQRS is usually optimized for fast queries by using **materialized views**. These views are precomputed representations of data that make querying much faster because they store denormalized tables.
  - In this system, the read database consumes events from the Event Store via the **Eventbus** and updates the materialized views accordingly. The read database is never directly written to; instead, it is updated based on the events it consumes.

3. Event Flow and Updates:

- **Event Publication and Consumption**:
  - When a user action (like adding an item) occurs, it generates an event, which is stored in the **Event Store**. This event is then **published** via an **Eventbus**.
  - The read database (which handles queries) listens to the Eventbus and **consumes the event** to update its materialized views. This ensures that the read side remains synchronized with the write side, albeit with some delay.

- **Eventual Consistency**:
  - Since the read database is updated asynchronously by consuming events, there can be a slight delay between when the write operation happens and when the read database reflects the updated data. This is known as **eventual consistency**.
  - Although there is a delay, the system ensures that the read side will eventually reflect the correct state once all events are processed.

4. Advantages of Using CQRS with Event Sourcing:

- **Improved Query Performance**:
  - The **read database** can be optimized for fast query performance by using denormalized, precomputed materialized views, which are updated based on events. This separation of concerns allows the system to handle complex queries without affecting the performance of write operations.
- **Scalability**:
  - The **write and read databases can be scaled independently**, allowing each to handle its specific workload. The write side can focus on capturing events, while the read side focuses on serving fast queries.
- **Replay and Reconstruction**:
  - Because all changes are stored as events, the system can **replay events** to rebuild the state at any point in time. This capability is useful for debugging, auditing, and recovering data.
- **Event Store as the Source of Truth**:
  - The **Event Store** becomes the **source of truth** for the system. All changes to the state are captured as events, providing a full history of the system's behavior. This is highly beneficial for auditing and ensuring that no information is lost.

5. Eventbus and Replay Mechanism:

- **Eventbus**:
  - The **Eventbus** is responsible for delivering events from the write side (Event Store) to the read side (materialized views). It acts as the communication layer, ensuring that events are published and consumed correctly.

- **Replay Events**:
  - One of the key features of Event Sourcing is the ability to **replay events**. If needed, the system can replay all or selected events to reconstruct the current state of the data. This is useful in cases of data corruption, debugging, or recovering from failures.

6. Materialized Views in Read Database:

- **Materialized Views**:

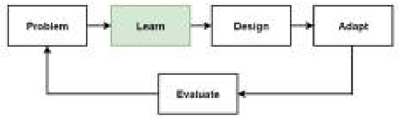
    - These views are **denormalized representations** of data designed to improve query performance. When events are consumed, they are applied to the materialized view to update it accordingly. This denormalization helps in faster querying but does require careful design to ensure that the right data is precomputed.

- **Denormalized Tables**:
    - In the read database, tables may be denormalized (i.e., data is duplicated across tables) to speed up queries. This approach is possible because the read database is optimized for fast reads rather than maintaining data integrity (which is managed on the write side).

Summary of the CQRS with Event Sourcing Pattern:

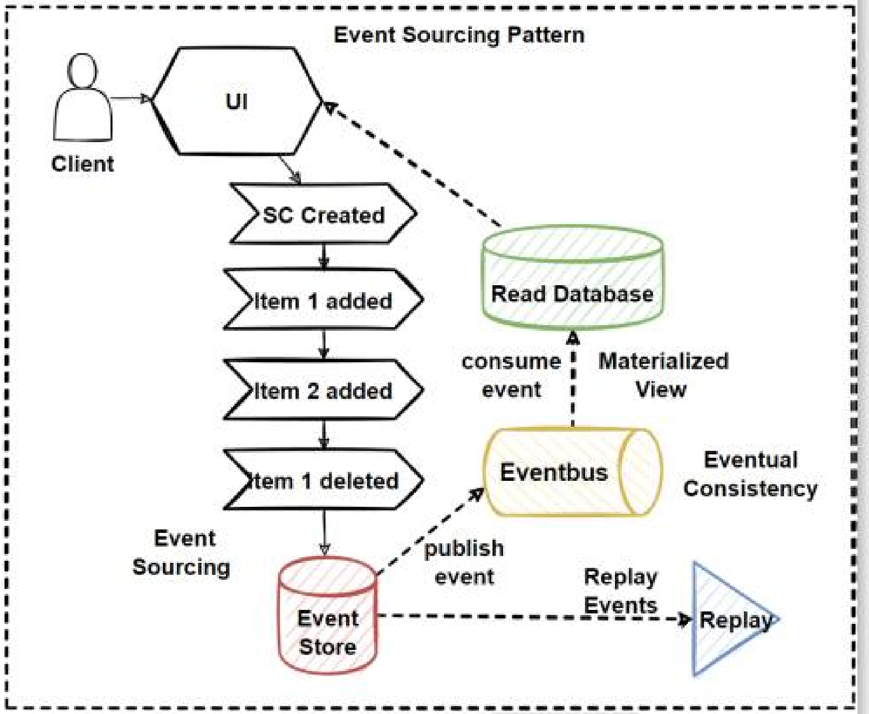
- **CQRS**: Separates read and write operations into different models and databases, optimizing both for their respective operations.
- **Event Sourcing**: Stores changes as events in an Event Store, providing a full history of data changes and enabling event replay for debugging or recovery.
- **Eventbus**: Publishes events to ensure that the read side is updated asynchronously, leading to **eventual consistency** between the read and write sides.
- **Materialized Views**: Precomputed views in the read database allow for faster queries, making the system more performant and scalable.

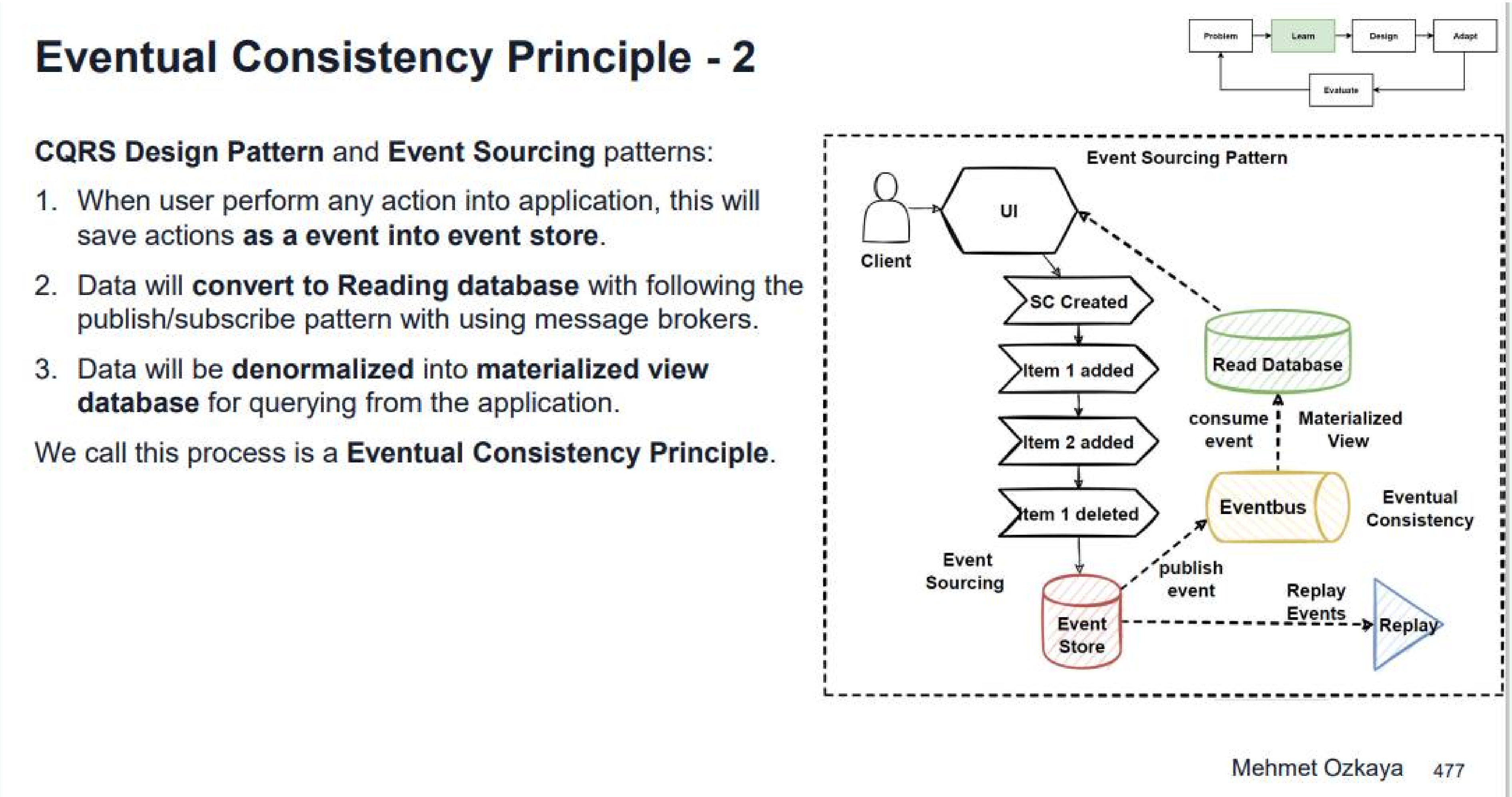
Would you like further examples of how this pattern can be applied or information on implementation in a specific context?



Eventual Consistency Principle

- **CQRS** with **Event Sourcing Pattern** leads **Eventual Consistency.**
- **Eventual Consistency** is especially used for systems that **prefer high availability** to **strong consistency**.
- The system will **become consistent after a certain time**.
- We called this latency is a **Eventual Consistency** Principle and offers to be **consistent after a certain time**.
- There are **2 type of "Consistency Level":**
- **Strict Consistency:** When we save data, the data should affect and seen immediately for every client.
- **Eventual Consistency:** When we write any data, it will take some time for clients reading the data.





Kindly explain the above in details

The images you shared explain the **Eventual Consistency Principle**, which arises when using **CQRS** (Command Query Responsibility Segregation) along with **Event Sourcing Pattern**. Let's break down the detailed explanation.

What is Eventual Consistency?

- **Eventual Consistency** is a consistency model used in distributed systems where, instead of having strong immediate consistency, the system guarantees that data will **eventually become consistent** after a certain amount of time.
- This approach is often used in systems that prioritize **high availability** over strict consistency. While the data may not immediately reflect the most recent changes in all parts of the system, it ensures that the system remains available for users and performs well.

1. Eventual Consistency in CQRS with Event Sourcing:

- When combining **CQRS** and **Event Sourcing**, the system operates with an **eventual consistency model** because the read and write sides are **separated** and data propagation between them is handled **asynchronously**.
- The **write operations** happen on the **write database**, where events are stored in an **Event Store**. The **read database**, on the other hand, consumes these events from an **Eventbus** asynchronously to update its **materialized views**. This results in a temporary delay before the changes in the write database are reflected in the read database.

2. Key Aspects of Eventual Consistency:

- **Asynchronous Update**: When an event occurs (like a user action), it is saved in the Event Store and **published asynchronously** to the read database. Since the update to the read database is not immediate, it leads to a short period where the data is **temporarily inconsistent** between the write and read sides.

- **High Availability vs. Strong Consistency**: Eventual consistency is particularly useful for systems that value **availability** over strict consistency. This model is used in scenarios where it's acceptable for users to see slightly outdated information for a short time but the system remains highly available and performant.

- **System Becomes Consistent After Some Time**: After an event occurs and the read side consumes it through the Eventbus, the system will eventually become consistent. The **Eventual Consistency Principle** guarantees that the system will reconcile the data across the write and read sides, though not immediately.

3. Two Types of Consistency:

- **Strict Consistency**:
  - In a system with strict consistency, once data is written, it is **immediately visible** to all clients. There is no delay in synchronization between the write and read sides. This is common in systems that require strong guarantees of correctness, such as financial systems.
- **Eventual Consistency**:
  - In systems with eventual consistency, after writing data, it takes some time for that data to become visible to all clients. During this period, clients might see old or stale data until the system completes the process of synchronizing the data.

4. Event Flow and Synchronization (Diagram Explanation):

- **User Action**:
  - The process begins when a client performs an action, such as creating a shopping cart (SC Created) or adding an item to the cart. Each of these actions triggers the creation of an **event**.
- **Event Store**:
  - These events are stored in the **Event Store**, which becomes the **source of truth** for the write operations. Every state change is recorded as an event, ensuring a full history of all actions.
- **Eventbus and Event Publication**:
  - The events are then **published** to the **Eventbus**, which distributes these events to other parts of the system that need to be aware of the changes, such as the read side.
- **Read Database (Materialized Views)**:
  - The read database **consumes** these events asynchronously from the Eventbus. It uses the events to update its **Materialized Views**—precomputed, optimized structures designed to make querying fast and efficient.
- **Eventual Consistency**:
  - The system will eventually synchronize the read and write sides, ensuring consistency. However, there is a small delay between when the data is updated on the write side (Event Store) and when it is reflected in the read database (Materialized Views).

5. How Eventual Consistency Works (Second Image):

- The second image further explains the **sequence of actions** in the eventual consistency model when using CQRS with Event Sourcing:

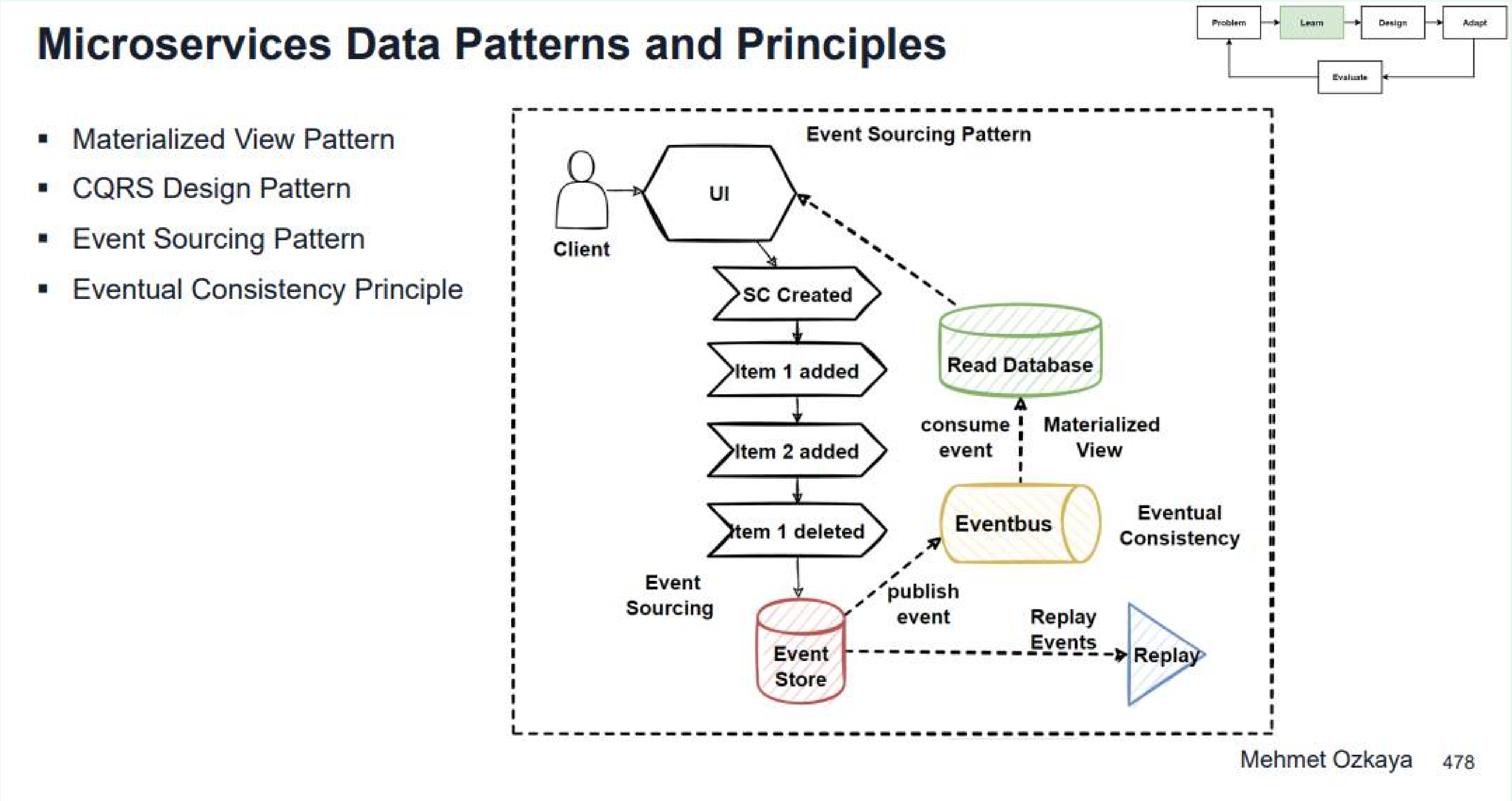
1. **Action Triggered by User**:
   - When a user performs an action (e.g., adding or deleting an item), this triggers the creation of an event that is saved to the Event Store.
2. **Publishing and Conversion to the Read Database**:
   - The event is **published** to the Eventbus using a **publish/subscribe pattern** (with message brokers). The read side **consumes** this event, which is then **converted into a materialized view**.
3. **Denormalization**:
   - The event is **denormalized** in the read database (for optimized query performance), updating the materialized view to reflect the current state.

- **This entire process is referred to as the **Eventual Consistency Principle** because the system is eventually consistent after the asynchronous synchronization is completed.

Summary:

- **CQRS with Event Sourcing** leads to **Eventual Consistency**, where there is a delay between when the write operation occurs and when the read side reflects the change.
- **Eventual Consistency** allows the system to remain highly available and performant, even if data is not immediately synchronized between the read and write sides.
- **Strict Consistency** requires immediate synchronization, which is not feasible in systems that prioritize high availability and performance over real-time consistency.
- **Materialized Views** help optimize query performance on the read side by consuming events and converting them into denormalized structures, but there is always a delay before they reflect the latest data.

Would you like to explore specific use cases or further examples where eventual consistency is applied?



Kindly explain the above image in details

The image you have shared presents the key **Microservices Data Patterns and Principles** using a combination of **Materialized View Pattern**, **CQRS Design Pattern**, **Event Sourcing Pattern**, and the **Eventual Consistency Principle**. Let's break down each component and how they interact in the context of microservices architecture:

1. Materialized View Pattern:

- **What is it?**: The **Materialized View Pattern** involves creating precomputed views (or snapshots) of data, typically to optimize read operations. In this approach, the system stores **denormalized data** that is optimized for fast querying, reducing the need to perform complex, expensive joins or aggregations at query time.
- **Why it matters**: This pattern is crucial in systems where the read side needs to be highly performant, as it allows queries to retrieve data quickly by using the materialized view that has already been computed based on incoming events.

2. CQRS (Command Query Responsibility Segregation) Design Pattern:

- **What is it?**: The **CQRS Design Pattern** separates the responsibilities of handling write operations (commands) and read operations (queries). In this pattern, the write side (which handles commands) is distinct from the read side (which handles queries).
- **Why it matters**: By separating these two responsibilities, CQRS optimizes the system for both efficient write operations (commands that update state) and high-performance reads (queries). Each side can be scaled and optimized independently, which is particularly useful in large-scale, high-throughput systems.

3. Event Sourcing Pattern:

- **What is it?**: In the **Event Sourcing Pattern**, instead of storing the current state of an entity, the system stores a sequence of events that represent all the changes made to that entity over time. These events are stored in an **Event Store**, and the current state can be reconstructed by replaying the events in the correct order.
- **Why it matters**: This pattern allows for complete traceability and auditability of all changes made to the system. It also provides the ability to **replay events** to recover the current state or investigate how the state evolved over time.

4. Eventual Consistency Principle:

- **What is it?**: The **Eventual Consistency Principle** arises when there is a temporary delay in synchronizing the read and write sides of the system. The system guarantees that after a certain amount of time, all the updates will be reflected in the read side, even though they may not be immediate.
- **Why it matters**: This principle allows the system to remain **highly available and performant** by allowing some flexibility in consistency. It is suitable for systems that prioritize availability and performance over strict, immediate consistency.

5. How These Patterns Work Together (Diagram Explanation):

The diagram illustrates the interaction between the client and the system, showing how the **Event Sourcing Pattern** feeds into the **Materialized View Pattern** and **CQRS** to achieve eventual consistency:

1. **User Interaction (Client)**:
   - A client (user) interacts with the system through the UI by performing various actions, such as creating a shopping cart (SC Created), adding items (Item 1 added, Item 2 added), or deleting items (Item 1 deleted).

2. **Events and Event Sourcing**:
   - Each of these actions triggers the creation of an **event**, which is then stored in the **Event Store**. The Event Store captures all state changes as a sequence of events, ensuring that the history of changes is preserved.
   - **Event Sourcing** stores every change (event) and allows the system to replay these events to rebuild the current state.

3. **Eventbus**:
   - The events stored in the Event Store are **published** via the **Eventbus**, which acts as the communication medium between the write and read sides. The Eventbus broadcasts these events to the components of the system that need to be updated.

4. **Read Database (Materialized Views)**:
   - The **read database** consumes events from the Eventbus and uses them to update the **Materialized Views**. These views are denormalized representations of data designed to optimize read performance.
   - For example, after consuming an event like "Item 1 added," the read database updates its materialized view to reflect the current state of the shopping cart, making it quick and efficient for the system to query this updated data.

5. **Eventual Consistency**:
   - Since the updates to the read side happen asynchronously, there is a period where the data on the read side may not be immediately up-to-date with the data on the write side. This leads to **eventual consistency**, where the system ensures that the data will **eventually become consistent** after the events have been processed.
   - The **Eventual Consistency Principle** is especially useful in microservices architectures where **high availability** and performance are prioritized over immediate consistency.

6. **Replay Events**:
   - The ability to **replay events** from the Event Store is another advantage of this architecture. If the system needs to rebuild the state (due to failure, debugging, or auditing), it can replay the events sequentially to restore the current state.

Summary of the Flow:

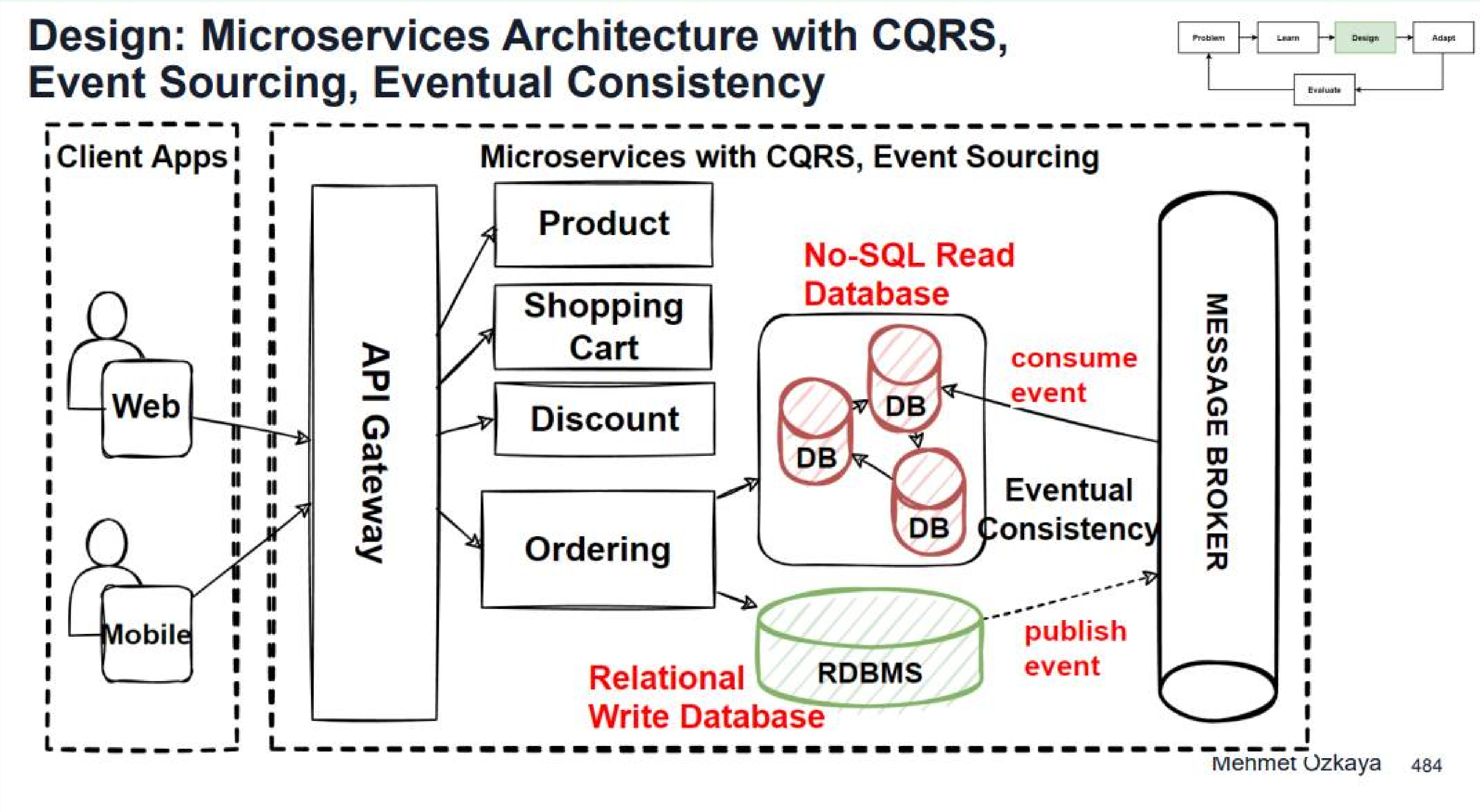
- The user performs actions that create events.

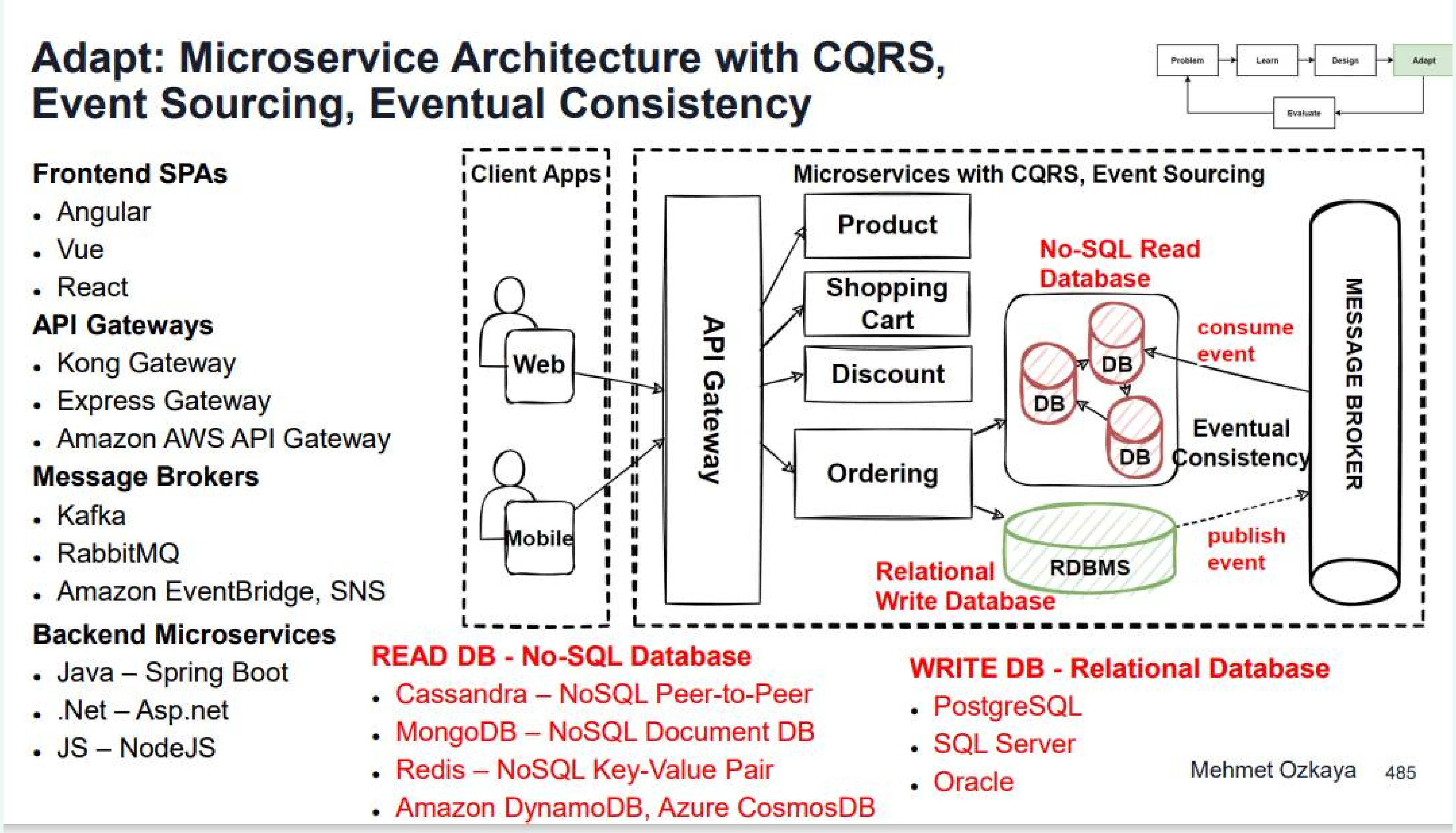
- Events are stored in the **Event Store** and published to the **Eventbus**.
- The **read database** (with materialized views) consumes these events asynchronously and updates the views to reflect the latest data.
- The system operates with **eventual consistency**, meaning the read and write sides may not be immediately synchronized, but will eventually become consistent after the events have been processed.
- **Event replay** allows the system to rebuild state at any point in time by replaying past events.

Why This Matters for Microservices:

- **Scalability**: Each component (read and write sides) can be scaled independently, improving system performance.
- **Performance**: The use of materialized views ensures that queries are highly performant, as they don't need to process complex joins or aggregations in real time.
- **Flexibility and Auditability**: The use of Event Sourcing allows the system to track every change and easily recover or inspect the system state by replaying events.
- **High Availability**: The Eventual Consistency Principle allows the system to maintain high availability even in distributed environments, ensuring that minor delays in synchronization do not disrupt overall functionality.

Would you like more details on how this setup could be implemented in a real-world microservices architecture?


Design: Microservices Architecture with CQRS, Event Sourcing, Eventual Consistency
Problem
Learn
Design
Adapt
Evaluate
Client Apps
Web
Mobile
Microservices with CQRS, Event Sourcing
API Gateway
Product
Shopping Cart
Discount
Ordering
No-SQL Read Database
DB
DB
DB
consume event
Eventual Consistency
MESSAGE BROKER
RDBMS
Relational Write Database
publish event
Mehmet Ozkaya
484



Kindly explain the above image in details

The images you've shared illustrate the architecture of a microservices system that uses **CQRS (Command Query Responsibility Segregation)**, **Event Sourcing**, and **Eventual Consistency** principles. Let's break down the details of each component and how they work together in this architecture.

Image 1: Design: Microservices Architecture with CQRS, Event Sourcing, and Eventual Consistency

1. Client Apps (Web & Mobile)

- Users interact with the system through **web** and **mobile** applications.
- These client apps make requests to the system, such as creating orders, adding items to the shopping cart, or fetching product information.

2. API Gateway

- The **API Gateway** acts as an intermediary between the client apps and the microservices.
- It handles all incoming requests from the web and mobile clients, directing them to the appropriate microservices. API gateways often provide cross-cutting concerns like rate limiting, authentication, and routing.

3. Microservices with CQRS and Event Sourcing

- **Microservices** are individual, self-contained services responsible for specific parts of the application. In this example, the microservices include:
  - **Product**: Handles product-related operations, such as retrieving product details.
  - **Shopping Cart**: Manages users' shopping cart operations (adding/removing items).
  - **Discount**: Manages discount-related logic, such as applying coupon codes.
  - **Ordering**: Handles orders placed by users.
- Each of these microservices follows the **CQRS pattern**, meaning they separate the read and write operations.

4. Relational Write Database (RDBMS)

- Each microservice writes its data to a **Relational Database** (RDBMS) using the command (write) side of the CQRS pattern.
- The relational database stores the latest state of data (e.g., order details, product inventory) in a fully normalized format and ensures **strong consistency** when data is written.

5. Event Sourcing and Event Bus

- Whenever a change occurs (such as an item being added to the cart or an order being placed), it triggers the creation of an **event** that is stored in the **Event Store** (represented by the relational database).
- These events are then **published** to a **Message Broker** (e.g., Kafka, RabbitMQ), which broadcasts them to other interested components, including the read side of the microservices.

6. NoSQL Read Database

- The read side of the microservices uses a **NoSQL database** (denormalized and optimized for fast querying) to store **materialized views** of the data.
- The NoSQL database consumes events from the Message Broker, updating its data based on the events (for example, updating the product information or order details).
- The read side operates under **eventual consistency**—meaning that while the updates might not be immediate, they will eventually reflect the latest data after the events are processed.

7. Message Broker

- The **Message Broker** (e.g., Kafka, RabbitMQ) plays a crucial role in distributing events across the system. It ensures that events are propagated from the command (write) side to the query (read) side, ensuring that the data is synchronized, even if delayed.

8. Eventual Consistency

- Since the system uses separate databases for read and write operations, there is a delay in synchronizing the data between the two sides. This delay leads to **eventual consistency**, where the read database eventually reflects the state of the write database after all events are processed.

Image 2: Adapt: Microservices Architecture with CQRS, Event Sourcing, and Eventual Consistency

This image extends the previous one by providing additional details on how this architecture can be implemented with various technologies and tools.

1. Frontend SPAs (Single Page Applications)

- **Angular**, **Vue**, and **React** are popular JavaScript frameworks for building modern web applications (SPAs). These frameworks can be used to create responsive and interactive client applications that consume APIs exposed by the backend microservices.

2. API Gateways

- The API Gateway options mentioned include:
  - **Kong Gateway**: A popular open-source API gateway for managing microservices.
  - **Express Gateway**: Another open-source API gateway built on Node.js.
  - **Amazon AWS API Gateway**: A fully managed service from AWS for creating, deploying, and managing APIs.

3. Message Brokers

- To handle event-driven communication between the services, message brokers like:
  - **Kafka**: Distributed event streaming platform for building real-time event-driven architectures.
  - **RabbitMQ**: Message broker used for asynchronous communication.
  - **Amazon EventBridge/SNS**: AWS services that provide event-driven communication capabilities for distributed applications.

4. Backend Microservices

- The backend microservices can be built using different frameworks and languages, such as:
  - **Java** with **Spring Boot** for building scalable and production-ready microservices.
  - **.NET** for building applications with Microsoft's framework.
  - **Node.js** for building lightweight, event-driven services using JavaScript.

5. Read Databases (NoSQL)

- The NoSQL databases used for the **read side** are optimized for fast queries and scalability:
  - **Cassandra**: A distributed NoSQL database that supports peer-to-peer architecture.
  - **MongoDB**: A document-based NoSQL database.
  - **Redis**: An in-memory key-value store often used for caching and fast lookups.
  - **Amazon DynamoDB** and **Azure CosmosDB**: Managed NoSQL services from AWS and Azure, respectively.

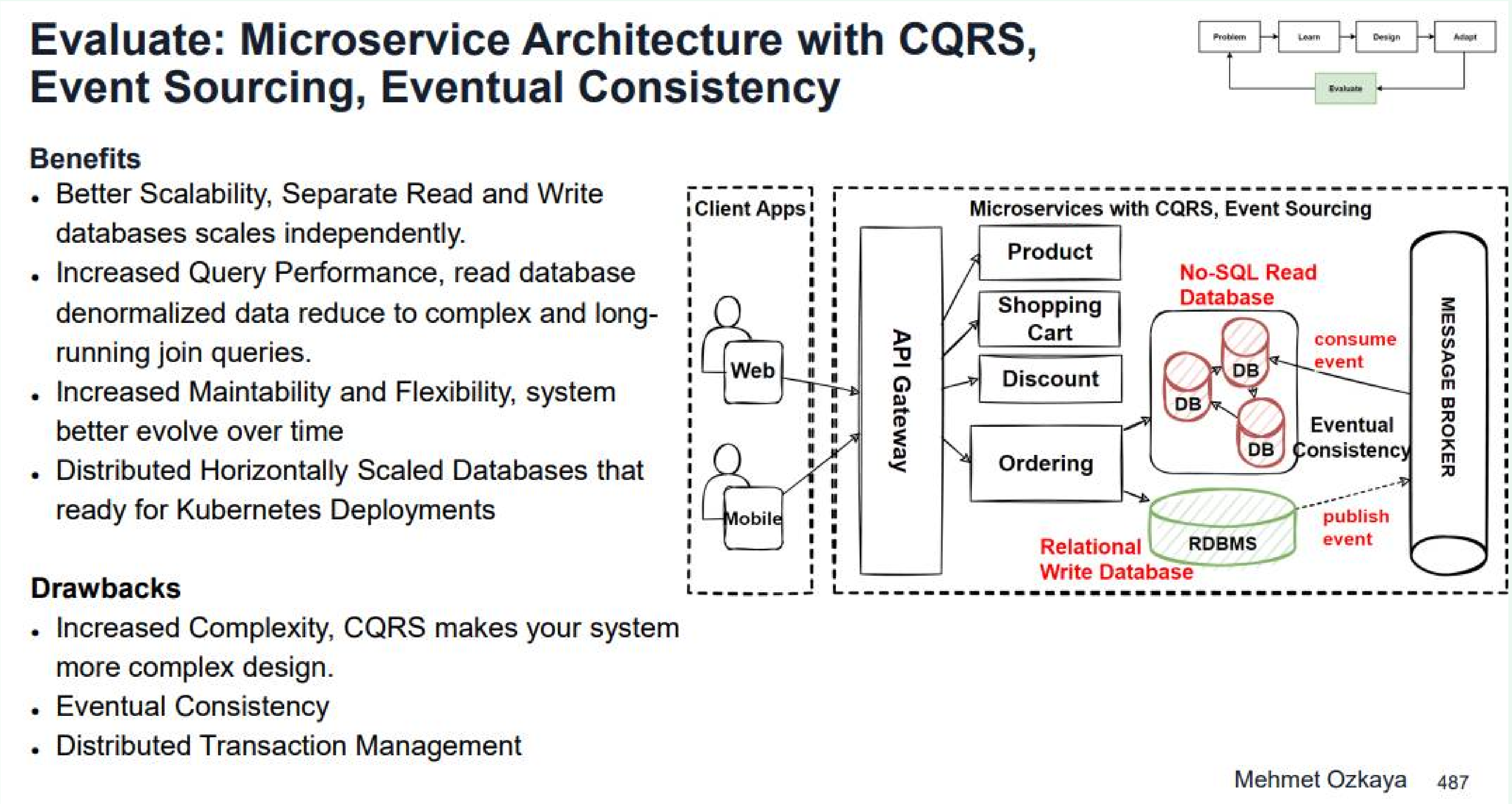
6. Write Databases (Relational)

- The relational databases used for the **write side** ensure strong consistency and transactional integrity:
  - **PostgreSQL**: Open-source relational database.
  - **SQL Server**: Microsoft's relational database platform.
  - **Oracle**: A widely-used relational database for enterprise-grade applications.

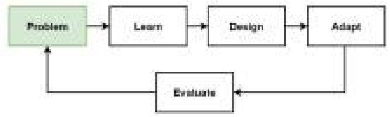
Summary:

- **Microservices Architecture**: Each service (Product, Shopping Cart, Discount, Ordering) follows the CQRS pattern, separating the write operations (handled by a relational database) from the read operations (handled by a NoSQL database).
- **Event Sourcing**: Events generated from changes in the system are stored and used to maintain the system's state. The events are published via a message broker and consumed by the read side to update the materialized views.
- **Eventual Consistency**: While the system is not immediately consistent across the read and write sides, it ensures that the data will eventually become consistent as the events propagate and are processed.

This architecture is highly scalable, distributed, and can handle a large number of users while maintaining performance and availability. Would you like more details on how to implement a specific part of this architecture?

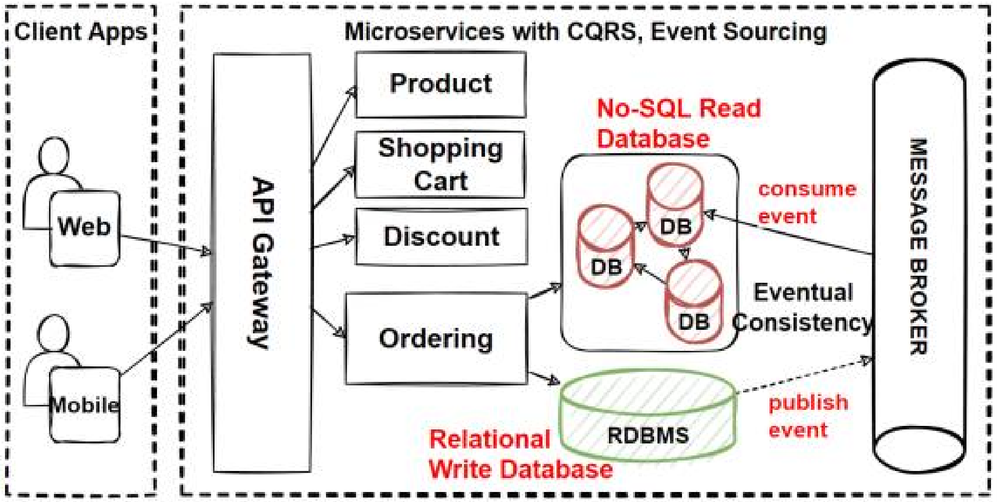


Problem: Manage Consistency Across Microservices in Distributed Transactions



Considerations

- Distributed Transactions that required to visit several microservices ?
- Consistency across multiple microservices ?
- Rollback transaction and run compensating steps ?



Problems

- Distributed Transaction Management
- Rollback Transaction on Distributed Environment
- Run Compensate Steps if one of service fail

Solutions

- Microservices Distributed Transaction Management Pattern and Best Practices
- Saga Pattern for Distributed Transactions
- Transactional Outbox Pattern
- Compensating Transaction pattern
- CDC - Change Data Capture

Mehmet Ozkaya 489

Kindly explain the above in details

The images you have shared provide an **evaluation of a microservice architecture** that leverages **CQRS (Command Query Responsibility Segregation)**, **Event Sourcing**, and **Eventual Consistency**, and also highlight the challenges of managing **distributed transactions** across microservices. Let’s break down each slide in detail.

Image 1: Evaluate: Microservice Architecture with CQRS, Event Sourcing, and Eventual Consistency

This slide summarizes the **benefits** and **drawbacks** of using CQRS, Event Sourcing, and Eventual Consistency in a microservices architecture.

Benefits:

1. **Better Scalability**:
   - The read and write databases are separated, allowing them to scale independently. For example, if your system experiences more read operations, you can scale the NoSQL read database without affecting the write database.
2. **Increased Query Performance**:
   - The read database is denormalized and optimized for queries, reducing the complexity and overhead associated with running complex joins and long-running queries. By using materialized views, queries can fetch data faster without needing to compute it on the fly.
3. **Increased Maintainability and Flexibility**:
   - This architecture improves **evolvability** because the separation of reads and writes allows for more flexible updates. The system can evolve independently in terms of handling read and write operations without having to worry about breaking changes.
4. **Distributed Horizontally Scaled Databases**:
   - The system can be easily deployed on **Kubernetes** or other container orchestration platforms due to the distributed nature of the architecture. Each microservice and its associated database can be horizontally scaled to handle increasing load.

Drawbacks:

1. **Increased Complexity**:
   - While CQRS and Event Sourcing improve scalability and flexibility, they also introduce **architectural complexity**. Managing separate databases for reads and writes, event-driven communication, and eventual consistency can be challenging.
2. **Eventual Consistency**:
   - The **eventual consistency** model means that the read side may not immediately reflect the latest updates from the write side. This can be problematic for systems requiring real-time consistency, and it must be handled carefully based on the business use case.
3. **Distributed Transaction Management**:
   - Managing **distributed transactions** in a system like this is challenging because different microservices may need to work together in a transactional process, but traditional database transactions (ACID) do not work well in a distributed environment.

Image 2: Problem: Manage Consistency Across Microservices in Distributed Transactions

This slide focuses on the **challenges of managing distributed transactions** across microservices and proposes potential **solutions**.

Considerations:

1. **Distributed Transactions**:
    - Some operations may span multiple microservices, requiring a distributed transaction to ensure data consistency. For example, when placing an order, multiple microservices like "Inventory," "Payment," and "Order Management" may need to coordinate.
2. **Consistency Across Multiple Microservices**:
    - In a microservices architecture, maintaining **consistency** across services is challenging because each microservice has its own database and operates independently. Achieving consistency without tightly coupling services is difficult.
3. **Rollback and Compensating Transactions**:
    - If one microservice fails in a series of operations, you may need to **rollback** the entire transaction or run **compensating actions** to undo the changes that other services have already made.

Problems:

1. **Distributed Transaction Management**:
    - Traditional transaction management (ACID) does not apply to distributed systems, where operations need to span multiple databases and services. Managing transactions across microservices without a clear strategy can lead to data inconsistencies.
2. **Rollback Transactions in a Distributed Environment**:
    - If one part of a distributed transaction fails, you need a strategy to rollback the entire operation or use a **compensating transaction** to reverse changes already made by other services.
3. **Running Compensating Steps**:
    - Compensating transactions are required when one microservice fails, but others have already executed their part of the transaction. For example, if payment succeeds but inventory update fails, the system should "compensate" by canceling the payment or refunding the amount.

Solutions:

1. **Microservices Distributed Transaction Management**:

    - Implementing patterns and best practices for managing transactions in a microservices architecture is crucial. These practices help ensure consistency while maintaining the loose coupling of services.

2. **Saga Pattern**:

    - The **Saga Pattern** is a widely used solution for managing distributed transactions in microservices. It breaks a transaction into a series of steps, where each step is handled by a different microservice. If one step fails, the previous steps are compensated (undone) to maintain consistency.

3. **Transactional Outbox Pattern**:
    - This pattern ensures that **events** generated by a service are reliably sent to the message broker (such as Kafka) even if there is a failure. The events are written to a local outbox table, which is then polled and sent to the message broker.

4. **Compensating Transaction Pattern**:
    - When a service fails during a distributed transaction, compensating transactions can be used to undo the effects of previously completed operations. This allows the system to revert back to a consistent state.

5. **CDC (Change Data Capture)**:
    - **Change Data Capture** (CDC) detects changes in a database (inserts, updates, deletes) and converts them into **events** that can be consumed by other microservices. This pattern is useful for keeping distributed databases synchronized in real-time.

Summary:

- **Scalability and Performance**: CQRS, Event Sourcing, and Eventual Consistency bring scalability and performance improvements but at the cost of increased complexity.
- **Distributed Transaction Management**: Managing transactions across microservices is a challenge due to the distributed nature of the architecture. The **Saga Pattern**, **Compensating Transactions**, and other solutions can help maintain consistency without requiring traditional ACID transactions.
- **Eventual Consistency**: In systems prioritizing availability and performance, eventual consistency allows for some flexibility in ensuring the data will eventually synchronize across services, though it may not happen immediately.

Would you like more details on how to implement any of the patterns or solve specific distributed transaction problems?