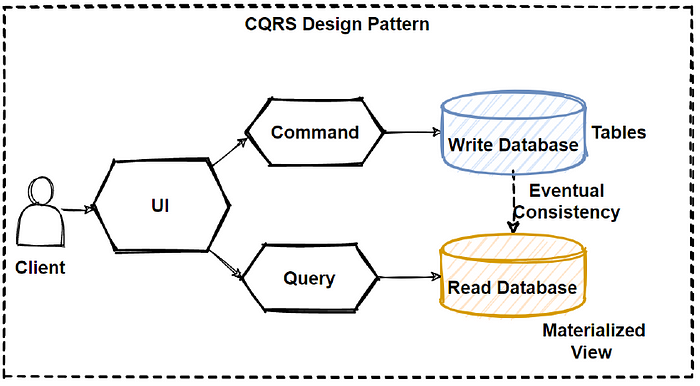
**CQRS**

CQRS stands for Command and Query Responsibility Segregation, a pattern that isolates data store read and update processes. CQRS implementation in your application can improve its performance, scalability, and security. The flexibility gained by moving to CQRS enables a system to evolve more effectively over time and prevents update instructions from triggering merge conflicts at the domain level.



Separate query and update models make design and implementation easier. although, CQRS code cannot be automatically generated from a database schema using scaffolding techniques such as O/RM tools (although, you can add your customised on top of the generated code).

You can physically split the read and write data for more isolation. In that instance, the read database can utilise its own query-optimized data schema. It can, for example, store a materialised view of the data to avoid complex joins or O/RM mappings. It may even employ a different sort of data storage. For example, the write database could be relational, and the read database could be a document database.

CQRS stands for Command Query Responsibility Segregation, which is a design pattern used in software architecture to separate the concerns of reading data (queries) and writing data (commands) into two distinct components. The CQRS pattern is often used in conjunction with event sourcing and domain-driven design (DDD) to build scalable, maintainable, and high-performance applications, especially in complex domains.

In a traditional monolithic architecture, a single model is used to handle both reads (queries) and writes (commands) to the data store. However, as an application grows in complexity and scale, this approach can lead to several challenges, such as performance bottlenecks, contention for resources, and difficulties in scaling reads and writes independently.

The CQRS pattern addresses these challenges by introducing two separate models:

Command Model: This is responsible for handling write operations and modifying the state of the application. Commands represent actions that change the data, such as creating, updating, or deleting records. The command model encapsulates the business logic and validation for these operations.

Query Model: This is responsible for handling read operations and providing data for queries. The query model is specifically optimized for efficient and fast retrieval of data needed for various read operations. It represents a denormalized view of the data, tailored to meet the specific needs of different queries.

Key characteristics and benefits of CQRS include:

Performance: By separating read and write concerns, each model can be optimized independently for its specific use case, leading to better performance and scalability.

Scalability: Read and write operations can be scaled independently based on their respective requirements. For example, read-heavy applications can have multiple replicas of the query model for better read performance.

Separation of Concerns: CQRS promotes a clear separation of concerns between commands and queries, making the codebase more maintainable and easier to reason about.

Event Sourcing: CQRS is often used in conjunction with event sourcing, where each state-changing operation is represented as an event, providing a complete history of state changes.

Domain-Driven Design (DDD) alignment: CQRS aligns well with DDD principles, where the domain model is designed to reflect the business domain closely.

While CQRS offers several advantages, it is essential to carefully consider the complexity and requirements of the application before adopting this pattern. It is best suited for applications with complex domains, high concurrency, and read-heavy workloads, where the benefits of separating reads and writes outweigh the additional complexity introduced by maintaining two separate models.