International Journal of Management, IT & Engineering
Vol.14 Issue 9, September 2024,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

Enhancing API Scalability and Performance with CQRS and Event-Driven Architecture: ATheoretical Case Study on Ad Management Systems

Sanket Panhale

Abstract

In modern application design, ensuring high performance and reliability while handling large volumes of requests and data is a significant challenge. Traditional monolithic architecture often falls short in meeting these requirements. Command Query Responsibility Segregation (CQRS) is a design pattern that addresses these challenges by separating read and write operations into distinct components, allowing each to scale independently and optimize for specific tasks. This paper explores the application of CQRS for API scaling, particularly when combined with an event-driven architecture that leverages event streaming platforms to manage data changes across the system. Using a theoretical case study in an ad management platform, the paper examines how CQRS can be adapted to various scenarios and discussespotential performance improvements. The study relies on assumed metrics to illustrate the benefits and considerations of implementing CQRS in similar high-demand environments.

Keywords:

CQRS;
API;
API scaling;
Event-Driven Architecture;
Event Streaming Platforms;
Eventual Consistency.

Author correspondence:

Sanket H Panhale,
Team Lead – Staff Software Developer,
SoFi, San Francisco, California - USA
Email: hopegreen1000@gmail.com

1. Introduction

One of the key challenges in designing modern applications is ensuring that they can handle high volumes of requests and data without compromising performance and reliability. Traditional monolithic architectures, where the same component is responsible for both reading and writing data, often fail to meet these requirements due to their limited scalability and high coupling of components. As a result, applications face difficulties in handling concurrent requests, leading to increased latency and potential bottlenecks. Command Query Responsibility Segregation (CQRS) is a design pattern that addresses these challenges by splitting the read and write operations into separate components, allowing each to scale independently and optimize for their specific tasks. This paper explores how CQRS can be applied to scale APIs, especially when combined with an event-driven architecture that uses event streaming platforms to communicate changes in data across the system. By decoupling these operations, CQRS enables better resource utilization and enhances the system's ability to handle high-demand scenarios. The paper also discusses how the CQRS pattern can be modified or adapted to suit different scenarios and requirements of API scaling.

2. Understanding basic concepts

2.1 What is CQRS?

CQRS stands for Command Query Responsibility Segregation, a software design pattern that separates the read and write operations of a data store. In CQRS, the write model (command) and the read model (query) have different structures and can be updated and accessed asynchronously. This separation allows for better performance, scalability, and maintainability of complex applications.

2.2 What is an API?

An API, or Application Programming Interface, is a set of rules and specifications that define how different software components can communicate and interact. APIs expose the functionality and data of one system to another without revealing the internal details or implementation, providing a standardized way to access and manipulate data and resources.

2.3 Event Streaming platforms

An event streaming platform is essential in modern event-driven architectures, enabling real-time data processing and communication across distributed systems. These platforms support scalability, fault tolerance, and event retention, allowing applications to efficiently handle large volumes of data and respond to events as they occur. Popular platforms like Apache Kafka, Amazon Kinesis, and Azure Event Hubs offer features such as data replication, scalability, and integration with other systems, making them crucial for building responsive and resilient applications. By leveraging these platforms, developers can create scalable, fault-tolerant systems that process and analyze data in real-time.

2.4 Event-Driven Architecture in CQRS

One of the key enablers of CQRS is the event-driven architecture, where events are the primary means of communication between different components of a system. Events are generated in response to changes in the state of the system and are consumed by various services to update their states or perform other actions. This approach enhances the responsiveness and flexibility of the system, as it allows for real-time processing of data and immediate propagation of changes. By leveraging event streaming platforms like Kafka, Kinesis, or EventHub, CQRS ensures reliable and consistent data propagation across distributed systems. These platforms provide the necessary infrastructure for handling large volumes of events, enabling systems to scale horizontally and maintain high availability. In the context of CQRS, event-driven architecture allows the read model to be updated asynchronously, ensuring that clients always have access to the most recent data while maintaining eventual consistency.

2.5 Eventual Consistency

Eventual consistency in distributed systems refers to the principle that, after a period of time and in the absence of further updates, all replicas of the data will converge to a consistent state. This model does not guarantee immediate consistency but ensures that the system will eventually reach consistency across all nodes, provided that no new updates are introduced. In the context of CQRS and event-driven architecture, eventual consistency ensures that the read model will eventually reflect the latest state of the write model, even if there are temporary inconsistencies.

2.6 Comparing CQRS with Other Architectural Patterns

CQRS is often compared with traditional monolithic and microservices architectures, each of which has its strengths and weaknesses. In a monolithic architecture, all operations (both read and write) are tightly coupled within a single codebase, leading to challenges in scaling and maintaining the application as it grows. The microservices architecture, on the other hand, breaks down the application into smaller, independent services that can be developed and deployed separately. However, even within a microservices architecture, the read and write operations are often handled within the same service, potentially leading to inefficiencies in handling high volumes of requests. CQRS takes the principles of microservices a step further by decoupling the read and write models entirely, allowing each to be optimized for its specific task. This separation enables greater scalability and flexibility, making CQRS particularly suitable for applications that require high throughput and low latency.

3. CQRS Pattern for API Scaling

3.1 Tenets of CQRS Pattern for APIs

The CQRS pattern for APIs can be summarized by the following tenets:

- Separate the write and read models of the data store and use different APIs for each.
- Utilize asynchronous communication between the write and read models and between the APIs and the clients.
- Employ event-driven architecture to propagate changes from the write model to the read model and notify clients about the status of their requests.
- Ensure eventual consistency to reflect the latest state of the write model in the read model, delivering accurate data to clients.

3.2 Key Components

Key components of the CQRS pattern for API scaling include:

- **Write APIs:** Handle data modification requests.
- **Read APIs:** Handle data retrieval queries.
- **Event Sourcing Platforms:** Such as Kafka, Kinesis, or EventHub, which ensure at least once delivery and robust disaster recovery.
- **Datastore:**Databases for writing and reading;can be separateand optimized for each operation.

3.3 Implementation

Implementing CQRS for API scaling involves several critical steps, each designed to decouple the read and write operations and leverage event-driven architecture for maximum efficiency. The process begins with designing Write APIs that are dedicated solely to handling data modification requests. These APIs receive data from clients, perform necessary validations, and return a basic identification ID for subsequent read operations. The received data is then published on an event streaming platform for asynchronous processing, ensuring that write operations do not block or delay other system activities.

Simultaneously, Read APIs are developed to handle client data retrieval queries. These APIs are optimized for performance and can access data that is eventually consistent with the write model. Anevent streaming platform, such as Kafka, Kinesis, or EventHub, plays a crucial role in securely accepting data from the Write APIs and delivering it to consumers. These consumers process the data, perform asynchronous tasks (such as detailed validation), and store the final state in a dedicated datastore.

A reliable datastore is established for storing both the write model and read model data, with separate read replicas to enhance system performance. While CQRS offers significant scalability benefits, it introduces complexities in managing eventual consistency, ensuring fault tolerance, and handling potential bottlenecks in the event processing pipeline. These complexities require careful design and optimization to ensure that the CQRS implementation delivers the desired improvements in scalability and performance.
These are simplified high-level implementation steps. In a real-world production environment, additional components like load balancers, API gateways, caches, and CDN networks would be integrated.

4. Business Use Case: Ad Management System

4.1 Establishing the Business Case

To demonstrate the application of CQRS for API scaling, let’s consider a hypothetical platform that manages ad placements for global retailers such as Walmart, Amazon, and eBay on a large scale. These companies rely on highly scalable APIs to submit advertisements, which must undergo various checks before being published on the platform. The API is responsible for performing a series of critical validations, including data sanity checks, regulatory compliance verification, fraud detection, and adherence to company ad publication policies.

In addition to these validations, the system must also ensure that all submitted ads are logged and stored for auditing purposes. This requirement is crucial for maintaining a transparent and compliant ad submission process, especially when dealing with sensitive or regulated content. The platform must be capable of handling a substantial volume of requests while ensuring that it meets all legal compliance and business requirements, including the ability to audit past submissions effectively.

Traditional synchronous processing approaches, where all these checks and logging operations are performed in a single, linear process, often struggle to meet the demands of high-traffic scenarios. CQRS offers a scalable solution by decoupling the validation, storage, and auditing processes, allowing the system to handle more requests efficiently without compromising performance.

4.2 Establishing Baseline

Typically, the synchronous ad intake process proceeds as follows: a retailer client sends an ad -> the ad is received -> a basic validation occurs -> the ad is stored in a datastore -> another API is called for detailed validation -> the record is updated with detailed validation results -> a response is sent back to the client including the unique Ad_Id -> the client submits another ad, and the cycle repeats.A consistent state of the advertisement can be defined as when the ad is stored in the data store together with detailed validation information. It is important to note that, at the conclusion of each cycle, the ad achieves a consistent state.Dependencies on external APIs and datastore latency can lead to delays and decreased throughput in this conventional approach. Here, CQRS provides a scalable solution by separating validation and storage processes, enabling the system to process more requests efficiently without compromising performance.

4.3 Baseline Performance Calculations

Before implementing CQRS, let's establish a baseline for the current system's performance using a traditional synchronous approach. Assume the following configuration:

- Maximum number ofads a system can support per day: 1,000,000 requests/day.
- Number of servers: 5 servers
- Concurrent requests per server: 5 requests/server
- Total concurrent requests: 25 requests

The total amount of time in a day expressed in milliseconds:

$$Tday = 24 \times 60 \times 60 \times 1000 = 86{,}400{,}000\ milliseconds$$

Given that there are 86,400,000 milliseconds in a day, we calculate the time available to process each request as follows:

$$trequest = \frac{86{,}400{,}000 \text{ milliseconds}}{1{,}000{,}000 \text{ requests/day}} \times 25 = 2160 \text{ milliseconds/request}$$

The time allocation for different API operations is shown in the table below.

Table 1. The time distribution for various operations in the synchronous(write and read)API.

Action	**Percent time spend on action**	**Time (milliseconds)**
Basic Validation	15%	324
Store Incoming Record in Data Store	25%	540
Call Another API for Detailed Validation	40%	864
Update and Store Validation Results Data	20%	432
Total	100%	2160

This leads to an overall processing duration of 2160 milliseconds for each request. Dependencies on external APIs and datastore latency contribute to this processing time, which limits the system's ability to scale beyond 1M requests per day. This baseline serves as the reference point for evaluating the improvements introduced by implementing CQRS.

4.4 Implementing CQRS for the Ad Management System

To explore the potential benefits of CQRS, let's consider a hypothetical implementation of the pattern in the ad management system. In this scenario, the platform is designed to handle high volumes of ad submissions, requiring a scalable and efficient processing architecture.This exploration is based on a theoretical scenario and assumed metrics, designed to illustrate how CQRS could enhance system performance and scalability.

Write Flow

Upon receiving an ad, the API initially conducts basic validation, a process that takes roughly 324 milliseconds as per previous calculations. Once validated, the ad is subsequently stored in the database, requiring an extra 540 milliseconds. The ad data is transmitted to an event streaming platform such as Kafka, Kinesis, or EventHub, adding about 100 milliseconds to the process. This estimate is on the higher side, considering network delays and a typical (non-optimized) setup. Finally, the system responds to the client with the Ad_Id, ensuring that the client receives a response within 964 milliseconds.This response time is significantly shorter than the total time required for the full synchronous processing of the ad, which includes detailed validation and updating the datastore.In this CQRS implementation, further processing tasks—such as detailed validation, updating the datastore with validation results, and ensuring audit logs are created—are offloaded to asynchronous consumers. These consumers process the ad data from the event streaming platform, eventually bringing the ad to a consistent state after all validations and audits are completed.

This CQRS approach demonstrates how the system could achieve higher scalability and efficiency by decoupling time-consuming tasks from the initial client-facing response. While the final state of the ad, including all validations and audit records, is eventually consistent, the immediate response to the client is optimized for speed, allowing the platform to process a much larger volume of requests.

Read Flow

The Read flow in the CQRS implementation is equally important, as it ensures that clients can retrieve the latest state of the ad, including the results of all validations and audits. The Read API is designed to be optimized for querying data, allowing it to handle high volumes of read requests efficiently. When a client makes a read request using the Ad_Id, the Read API accesses the read-optimized datastore, which is eventually consistent with the write model. This means that while there may be a slight delay in reflecting the most recent changes made by the Write API, the system ensures that all data is eventually consistent across both the read and write models.

4.5 Performance Calculation After CQRS Implementation

These calculations focus on determining how many numbers of Ads per day the system can accept with the exact same server configurations.

Table 2. The time distribution for various operations in the CQRS writeAPI.

Action	Time (milliseconds)
Basic Validation	324
Store Incoming Record in Data Store	540
Sending Data to Event Streaming Platform	100
Total	964

This approach reduces the total response time to 964 milliseconds per request.Given the same server configuration, we can calculate the new request handling capacity. We can first calculate the capacity per thread per day:

$$Requests\ per\ thread\ per\ day = \frac{86{,}400{,}000\ \text{milliseconds/day}}{964\ \text{milliseconds/request}} \approx 89{,}628\ requests/thread/day$$

For 5 threads per server across 5 servers:

$$Total\ requests\ per\ day = 89{,}628\ requests/thread/day \times 5\ threads \times 5\ servers$$
$$= 2{,}240{,}700\ requests/day \approx 2.2\ million\ requests/day$$

With CQRS implemented, we can handle ~2.2M requests per day. This represents more than 120% improvement over the initial baseline of 1M requests per day.

4.6 Further Optimizations with CQRS

To further enhance performance, we can modify the CQRS implementation.Once basic validation is completed, we send the advertising data straight to the event streaming platform instead of first saving it in the database. We need to update the event consumers to save the ad data in the datastore upon arrival before further processing.

It's crucial to carefully choose an event streaming platform that ensures *at least once* delivery, such as Kafka, Kinesis, or EventHub, to prevent any loss of ad data. Keep in mind that the *at least once* delivery model guarantees the event will always be delivered but may result in consumers receiving the same event multiple times. This model is efficient for handling large volumes of data, but it requires consumers to be idempotent, meaning they must effectively manage duplicate messages without adverse effects.

Since we are not giving the client an Ad_Id, we can offer an additional API to retrieve all Ads by the client that didn't pass the validation. Additionally, we can provide a daily CSV report to the client, or if there is a UI, they can view the failed Ads there.

4.7 Performance Calculation with Optimized CQRS

These calculations focus on determining how many numbers of Ads per day the system can accept with the exact same server configurations.

Table 3. The time distribution for various operations in the optimized CQRS writeAPI.

Action	Time (milliseconds)
Basic Validation	324
Sending Data to Event Streaming Platform	100
Total	424

This approach reduces the total response time to 424 milliseconds per request.Given the same server configuration, we can calculate the new request handling capacity. We can first calculate the capacity per thread per day:

$$Requests\ per\ thread\ per\ day = \frac{86{,}400{,}000\ \text{milliseconds/day}}{424\ \text{milliseconds/request}} \approx 203{,}773\ requests/thread/day$$

For 5 threads per server across 5 servers:

$$\begin{aligned} Total\ requests\ per\ day &= 203{,}773\ requests/thread/day \times 5\ threads \times 5\ servers \\ &= 5{,}094{,}325\ requests/day \approx 5.1\ million\ requests/day \end{aligned}$$

With this optimized CQRS implemented, we can handle ~5.1M requests per day. This represents more than400% improvement over the initial baseline of 1M requests per day.

5. Pros and Cons of Using CQRS in API Scaling

In this subsection, we will examine the different advantages and disadvantages of implementing CQRS for API scaling.

5.1 Pros

1. Scalability: By separating the read and write operations, each can be scaled independently, allowing for better handling of high volumes of requests.
2. Performance: CQRS can improve performance by optimizing the read and write models for their specific tasks.
3. Flexibility: The separation of concerns allows for more flexible and maintainable code.
4. Event-Driven Architecture: Utilizing event-driven architecture with platforms like Kafka, Kinesis, or EventHub ensures reliable and consistent data propagation.

5.2 Cons

1. Complexity: Implementing CQRS introduces additional complexity in managing separate models and ensuring consistency across distributed systems.
2. Eventual Consistency: Ensuring eventual consistency can be challenging, as it requires careful design and handling of asynchronous communication.
3. Potential Bottlenecks: Handling potential bottlenecks in the event processing pipeline requires careful consideration and optimization.
4. Idempotency: Consumers must be idempotent to effectively manage duplicate messages without adverse effects, adding to the complexity.

6. Conclusion

In conclusion, the implementation of the CQRS pattern for API scaling using event-driven architecture offers significant benefits in terms of performance, scalability, and maintainability. By separating read and write operations, utilizing asynchronous communication, and leveraging event-driven architecture, organizations can achieve substantial improvements in request handling capacity while maintaining system reliability and integrity. The practical use case discussed in this paper is hypothetical and based on assumed metrics, aiming to demonstrate the potential effectiveness of CQRS in handling high volumes of requests and data. However, it is essential to consider the complexities and challenges associated with implementing CQRS, such as managing separate models, ensuring consistency across distributed systems, and handling potential bottlenecks in the event processing pipeline. Future research could explore the integration of machine learning models to optimize the read and write models or investigate the application of CQRS in other domains that require high scalability and performance. Overall, CQRS represents a powerful architectural pattern for modern applications that require high scalability and performance.

References

[1] A. Debski and B. Szczepanik, "A Scalable, Reactive Architecture for Cloud Applications," in Proceedings of the 2017 IEEE 2nd International Workshops on Foundations and Applications of Self Systems (FAS W), Tucson, AZ, USA, 2017, pp. 231-236, doi: 10.1109/FAS-W.2017.80.

[2] Z. Long, "Improvement and Implementation of a High Performance CQRS Architecture," in *Proceedings of the 2019 IEEE 5th International Conference on Computer and Communications (ICCC),* Chengdu, China, 2019, pp. 1023-1027, doi: 10.1109/ICCC47050.2019.9064291.

[3] D. Betts and J. Dominguez, *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft Corporation, 2012.

[4] U. Yadav, *Microservice Patterns and Best Practices: Explore patterns like CQRS and Event Sourcing to create scalable, maintainable, and testable microservices*. Birmingham, UK: Packt Publishing, 2019.

[5] GeeksforGeeks, "CQRS Design Pattern in Microservices," *GeeksforGeeks*, May 23, 2023. [Online]. Available: https://www.geeksforgeeks.org/cqrs-design-pattern-in-microservices/.

[6] IBM Developer, "Building an application using microservices and CQRS," *IBM Developer*, Oct. 17, 2018. [Online]. Available: https://developer.ibm.com/articles/cl-build-app-using-microservices-and-cqrs-trs/.

[7] Microsoft Advertising, "Advertising Guides and Code Examples," *Microsoft Learn*, 2024. [Online]. Available: https://learn.microsoft.com/en-us/advertising/guides/?view=bingads-13.

[8] Google Ads, "Get Started with the Google Ads API," *Google Developers*, 2024. [Online]. Available: https://developers.google.com/google-ads/api/docs/start.