UDC 004-042

S.O. Diakov, T. E. Zubrei, A.S. Samoidiuk

APPLICATION OF EVENT SOURCING AND CQRS PATTERNS IN DISTRIBUTED SYSTEMS

Annotation: The purpose of this report is finding suitable approaches for dealing with the issue, particularly ability to recreate system state in modern high load distributed systems. In order to achieve the goal, the report will overview existing problems, compare conventional design to proposed architecture solutions. A combination of command query responsibility segregation (CQRS) and event sourcing is suggested to solve performance and design issues that often arise in conventional information systems development.

Keywords: CRUD, CQRS, event sourcing, software architecture, design patterns, data modeling.

Problem Statement

The common template for any data-oriented application is multi-level architecture [1]. The main idea is to use the division of responsibility to keep the presentation, data storage and business logic separated from each other. The persistence layer should not know about the mechanisms used to store and retrieve data, it's only responsible for data operation with storage. A data layer has to deal with different relations between entities and often works as a bridge from application domain model to its normalized view in data storage. Data changes are generally expressed as C, U, and D of CRUD (create, update and delete).

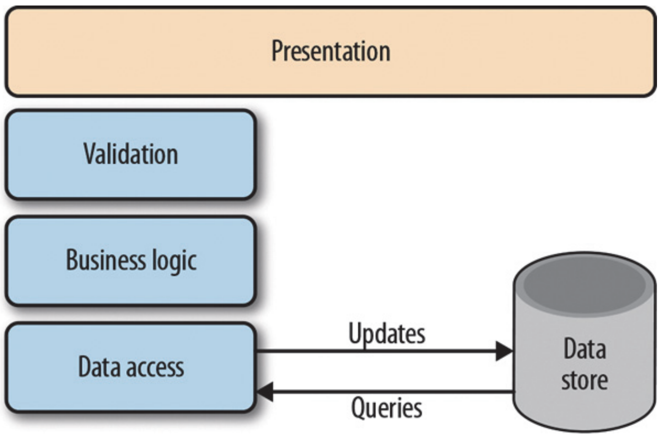

Fig. 1 - A traditional CRUD architecture

So, what's wrong with this approach? This model is so popular that most people are not even thinking about an alternative, and for simple applications it may not cause any problems. However, there are a few shortcomings in this conventional architecture.

The common problem of CRUD applications is that they are receive all data models and views from its primary data storage on which they depend. It enforces two different requirement of data structure: fast writes and fast reads. These parameters are hard to balance using only one solution and in most cases this problem is lessen by adding caches. However, with caches comes additional complexity which requires tremendous knowledge to handle properly.

Another issue with CRUD-like systems is violation of single responsibility principle, since update operation may not only do the update but also read newly changed data. The User object may have an id or update datetime, or other generated data that is present while reading the object, however the persistence layer will forbid you from updating them yourself. Therefore, code becomes more unmaintainable as the scope of application grows.

In fact, writing and reading can be differentiated by its priorities:

Table 1

Comparison of areas of concern of two operations

Writing	Reading
Assuring data integrity	Perform efficient queries and lookups
Enabling atomic updates and transactions	Calculate derived and aggregated values (sum, average, etc.)
Optimistic concurrency or locking	Provide a number of data views
Enforcing write permissions	Enforce row and column level permissions

An overview of existing solutions

There are a lot of way to deal with some of the issue above. Most application use caches [8] as fast and denormalized way to access heavily requested data. However introducing them adds new layer of complexity since caches need to be synchronized for all instances of application, its size and objects it contains is a very debatable subject considering different applications may use it for various amount of reasons. But the most difficult task regarding caches is keeping them up-to-date. Caches aren’t the persistent data storage meaning they have to be rebuilt on each application launch from some source. This creates a gap between this data storage - source of truth - and caches.

Transactions [9] are mostly considered as silver bullet of CRUD applications. While they are effective at keeping data integrity is SQL databases it causes a lot of overhead and business logic put on the data storage further violating single responsibility pattern. Transactions that are held open for quite a long time make the data storage track changed rows of frequently-modified tables that could be cleared. Moreover, it is really costly to roll back transactions. For some databases to roll back transactions takes more time than committing it.

Another way to increase application performance is vertical scaling [10]. It’s a concept of adding more resource to single instance allowing for faster computation of larger amount of data. In most scenarios the servers are already at full capacity physically. To see an actual impact of scaling typically would involve purchasing an entire new server to replace the old one. This is still vertical scaling from the database/application point of view. Having one high performance server is generally more expensive than buying a few less powerful.

Proposed solution

Main idea of CQRS (Command Query Responsibility Segregation) is separation of models for updating and reading information. Collaboration and staleness are two driving forces of CQRS. Collaboration means set of rules on how many participants will use / change the same shared data. Often there are rules that indicate which actor can execute which modifications and modifications that can be accepted in one case may be forbidden in another. Actors can be people like ordinary users or automated as software.

Staleness is demonstrated by the fact that in shared use, when data is shown to one actor, it may be changed by another one. Almost any system that uses caches serves stale data - often due to performance reasons. This means that we can not take into account the decisions of their actors, because they can be made on the out-of-date data. Standard layered architectures do not address any of these issues. Although all in one database could be a single step in the direction of collaboration, the staleness is usually more unpleasant in these architectures because of the caches usage as a performance improvement after data modeling is already done.

These issues are addressed in CQRS via read and write models segregation, where queries are for reading while commands deal with data updates.

- Commands have to tell what needs to be done rather that tell how it should be done. ("Switch the lights on, "not" set LightStatus to ON."). In most scenarios they are put in some queue for later asynchronous processing.
- Queries aren’t allowed to change anything in data storage. They should return DTO which is a container for data. You may think of it as a struct.

Comparing data flow of CRUD-based applications to CQRS ones, implementation and development process is far simpler with separation of responsibility. Both models can be designed via one data storage or be completely separated having different structure. Distributed systems greatly benefit from having numerous read optimized model closely located to request them applications. The separation of reading and writing storages also allows each to scale according to the corresponding load. For instance, reading stores are generally faced with a much greater load than store recordings.

Denormalization of data allows creation of read optimized views which may make enormous impact on a performance by reducing number of data transformations necessary to form the result.

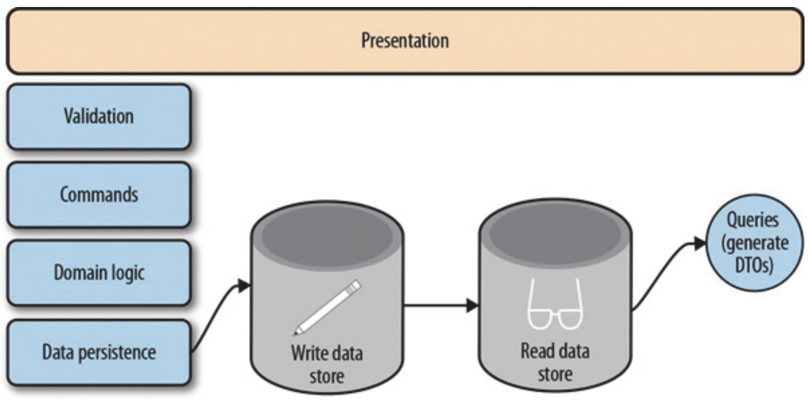

Fig. 2. - A CQRS architecture with different read and write stores

Let’s analyze benefits and drawbacks of this approach before taking it to another level by adding event sourcing on top.

Advantages

- **Optimized data schemas**. Data models can be designed for read intensive purposes while write model should assure fast write operations.
- **Independent scaling**. Segregation of models allows independent scaling of different services based on their load.
- **Simpler queries**. Read optimized views make this operation that much faster and easier.
- **Security**. Permission enforcement is handled better for actors to perform write operations.
- **Concerns separation**. Maintainability and flexibility are gained as a result of the data models segregation. Writing model should be responsible for complex business logic, while reading model is kept relatively simple.

Things to consider

- **Complexity**. While core of CQRS is fairly straightforward an actual implementation may lead to some complexity.
- **Messaging**. In most cases messaging goes hand in hand with CQRS, though it’s not a requirement. This enforces asynchronicity in application design.
- **Eventual consistency**. Separating read and write models may result in read data being stale. However, data loss or inconsistency are impossible by design.

Event sourcing

Event sourcing neatly supplements CQRS further enhancing availability and eventual consistency of application. Main idea of event sourcing lies in having ordered set of changes rather than aggregated state. This means each event doesn’t replace shared data but rather works as delta. Generally, it is presented as append-only log where actor can send commands and when we need to recreate entity state we can replay each command.

Dealing with events implies introducing asynchronicity to application. Therefore, this model is eventually consistent – every event will be processed at some point in future. Using log as a transport and storage of events prevents any concurrent issues since every command for one entity is processed one by one. Event sourcing and log model are heavily used in databases as a tool to synchronize different nodes. Every write to database is stored in its log and when replication happens new node just needs to replay all commands from the log. There’s a lexical difference between command and event. Event is a fact, it happened at some time in the past, while command is more of an intent to do something, an application which may be satisfied or rejected.

Multiple data services may consume events and update its model accordingly without any repercussions for not thinking about permissions or data integrity. It allows to store data in whichever model suits the cause, duplicating and precomputing any required data. Usually application that use event sourcing also have denormalized data. Having one log of all commands allows easy synchronization of multiple data storages which can be tuned to be either read intensive or write intensive. This is very different from normalization standards of typical CRUD application. Having different read optimized data representations will increase performance tremendously and data integrity issue is dealt via each data service separately updating its model.

There are quite a few benefits of event sourcing:

- Asynchronicity.
- Source of truth.
- Concurrency.
- Tracing.
- Scalability.
- Reusability.

Event replay

Let’s take a closer look at example of accounting system with event sourcing. Here is a log of events concerning bank account:

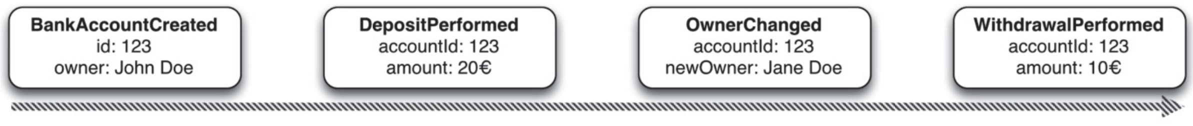

Fig. 3. - Append-only log of accounting events

Each of these events are persisted and are processed one by one. It allows to recreate account’s state at any given time by replaying all previous events. This process creates an aggregate - data model which represents state of account at a moment. This aggregate is similar to domain object in CRUD-based application. In the example below each event is applied to previous aggregate having empty BankAccount as the initial state.

In order to query this, aggregate we can either build it from scratch on each request or have pre build version stored and continuously update it. The second approach is more preferred considering performance implications on aggregating history of events every time. It also allows to tune this presentation for read intensive purposes since it can be requested often. It showcases distinct segregation of commands and queries.

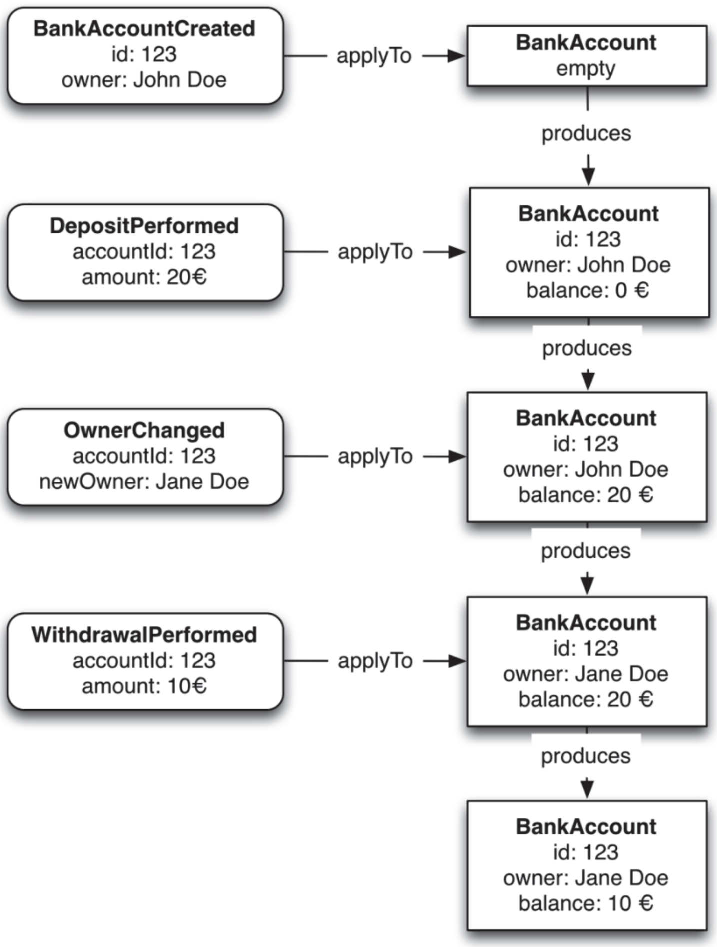

Fig. 4. - Creation of a BankAccount aggregate

One of the biggest pitfalls of CRUD architecture is inability to proper trace changes to domain entity. The only approach used for that is logging, however most of the times it's not enough for debugging purposes and it doesn't allow to properly recreate state of the system. This is solved in event sourcing architecture by enforcing the log onto commands which leads to ordered sequence of events - traces of each actor's actions. This approach allows to work with domain objects (aggregates) as well as keep the changelog.

Conclusion

Event sourcing enables straightforward and reliable way to log state changes via zero loss protocol. With ordered log or a journal as a backbone such systems can easily and efficiently recover. CQRS goes a step further, making queryable view of raw events which can be used by other data services.

Moreover loosely-coupled design is a result of building stateful applications in this way. Simple troubleshooting and upgrading, resilience and scalability are the most glaring benefits.

Having different read optimized data representations will increase performance tremendously and data integrity issue is dealt via each data service separately updating its model.

REFERENCES

1. Fowler M. Patterns of Enterprise Application Architecture / M. Fowler. — Boston: Addison-Wesley Longman Publishing Co., Inc., 2002. — 576 c.

2. Udi D. Clarified CQRS [Електронний ресурс] // http://udidahan.com. 2005. URL: http://udidahan.com/2009/12/09/clarified-cqrs/.

3. Fowler M. CQRS [Електронний ресурс] // https://martinfowler.com. 2008. URL: https://martinfowler.com/bliki/CQRS.html.

4. Richards M. Software Architecture Patterns / M. Richards. — Sebastopol: O'Reilly Media, 2015. — 47 c.

5. Korkmaz N. Practitioners’ view on command query responsibility segregation / N. Korkmaz, M. Nilsson // School of Economics and Management Department of Informatics Lund University. — 2014. — C.33-37.

6. Barnkob M. Event Sourcing and Command Query Responsibility Segregation Reliability Properties / M. Barnkob, J. Krukow // Computer Science University of Aarhus. — 2018. — C.21-39.

7. Kleppmann M. Making Sense of Stream Processing / M. Kleppmann. — Sebastopol: O'Reilly Media, Inc., 2016. — 172 c.

8. Handy J. Cache Memory Book, The (The Morgan Kaufmann Series in Computer Architecture and Design) 2nd Edition / J. Handy. — Burlington: Morgan Kaufmann, 1998. — 229 c.

9. Bernstein P. Principles of Transaction Processing (The Morgan Kaufmann Series in Data Management Systems) / P. Bernstein, E. Newcomer. — Burlington: Morgan Kaufmann, 2009. — 400 c.

10. Atchison L. Architecting for Scale / L. Atchison. — Sebastopol: O'Reilly Media, 2016. — 154 c.