STUDIA INFORMATICA 2019
Volume ?? Number ? (???)

Mateusz NIEDZIELA
Rzeszow University of Technology, The Faculty of
Electrical and Computer Engineering
Maciej PENAR
Vulcan Sp. z o.o.

PERFORMANCE EVALUATION OF CQRS ARCHITECTURE – INTRODUCTORY RESEARCH

Summary. This article addresses the performance evaluation of the Command And Query Responsibility Separation Software Architecture. The evaluation was carried out with TPCE benchmark. In this paper we present our results and our reflections upon this architecture.

Keywords: CQRS, performance analysis, TPCE, streaming OLTP

EWALUACJA WYDAJNOŚCI ARCHITEKTURY CQRS – BADANIA WSTĘPNE

Streszczenie. Artykuł dotyczy analizy wydajności architektury Separacji Odpowiedzialności Komend I Zapytań. Testy wydajności zostały przygotowane w oparciu o benchmark TPCE. W tym artykule prezentujemy wyniki oraz wnioski dotyczące tego rozwiązania architektonicznego.

Słowa kluczowe: CQRS, analiza wydajności, TPCE, strumieniowy OLTP

1. Introduction

In recent time some trends have emerged in Software Engineering that aim to encourage the developers and architects to experiment with the Software Architectures that go beyond the classic Three-Tier Architecture. This happens because trends like Cloud Computing, Containerization/Virtualization, Big Data and NoSQL/NewSQL Databases have emerged and are no longer out-of-reach for developers [1]. Within this article, We ponder upon one of the modern architectures – named Command And Query Responsibility Separation Architecture – CQRS. This Architectural Pattern is not leveraged by any existing framework and little scientific materials exist that motivates or even states the advantages of such solution (one example of such work can be found in [2]) - We consider our work as preparatory, as We carried out some experiments that aimed to estimate the scale of benefits/drawbacks of using such architecture.

In this article, We present our methodology and findings. The article is organized as follows: next section contains short overview of CQRS architecture with a note about how it differs from standard Three-Tier Architecture (3-TA). Third section contains description and evaluation of the carried our tests. Last section contains conclusions and note about further directions.

2. CQRS

2.1.General Concept

The building blocks of CQRS are Command Query Separation (CQS) and Single Responsibility Principle (SRP). SRP was proposed around 1980 and was popularized by Bertrand Meyer [3] aiming in separation of read and write operations. It was one of Martin SOLID principles [4]. CQRS pattern (figure 1) integrates those concepts and defines how it should be implemented. There was also endeavors to find more efficient ways to address staleness and collaboration problems in traditional N-tier architectures (N-TA) as it was observed that some data was supposed to be read only. CQRS moves read-only data to read-optimized store which should allow horizontal scaling[5]. Effectively read optimization techniques include i.e. dedicated indexes, materialized views, partitioning and denormalization. The write operations should be handled in a write-optimized database. Such approach may result in inconsistencies between read-only/read-write databases.

As for a decade since CQRS was introduced it has remained in practitioners interest and there are implementations which utilizes the pattern [6] [7]. Nonetheless, the traditional 3-TA is still present in software industry field as it gives predictive results during project development and is flexible enough for majority of use cases. The split between separate presentation, business logic and database layers allow to work in parallel on every of those. Data flows between each of the layers through single channel, what is challenged in CQRS pattern – and is still problematic to debug in development [8]. In overall, CQRS systems tend to be much more complex than N-TA [9].

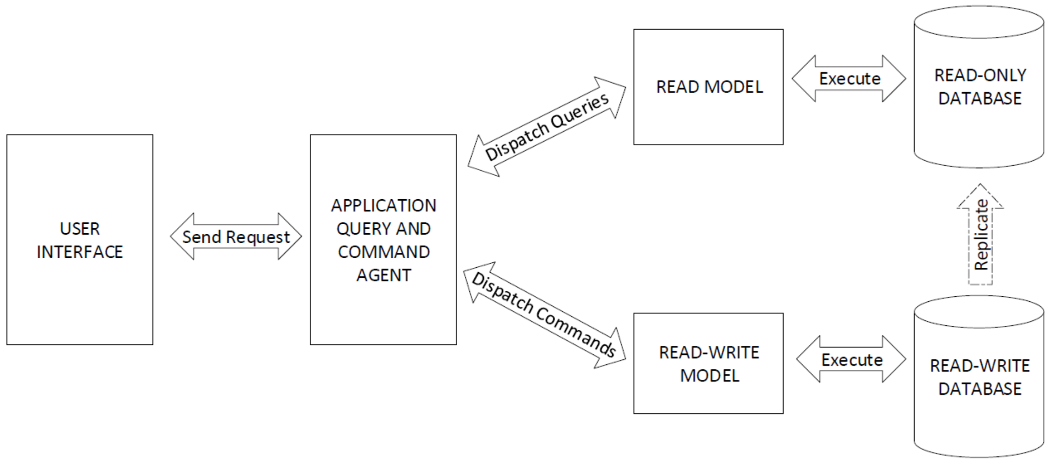

Fig. 1. Scheme of a CQRS architecture
Rys. 1. Schemat architektury CQRS

2.2. Advantages and disadvantages

Separation of queries and commands requires additional layer which distinguishes and routes them to proper endpoint. Also duplication of data and necessity to synchronize the data between two different stores adds up to the solution cost. On the bright side CQRS pattern allow us to optimize read-only and read-write endpoints separately. Propagation of data from read-write DB to read-only DB could benefit from dedicated software solutions or hardware support. Lack of solutions which leverage automation of building separated models may lead to manual implementation of replication mechanisms which may result in using metadata tribble antipattern [10].

2.3. Role of Message Brokers

Message Brokers (i.e. RabbitMQ [11], Kafka [12]) could be implemented in CQRS as agent responsible for recognizing and dispatching Queries and Commands. That seems to be preferable as there is no dedicated software which implements or supports CQRS pattern and gives the architects single entry point for system. Moving step further CQRS is often connected with Event Sourcing [13]. Paring those solutions give possibility to use division between command and query databases of CQRS pattern, that command DB is event store and query DB is denormalized read store.

2.4. Implementing CQRS

In the light of CAP theorem CQRS pattern trades consistency to gain higher availability and lesser response times. Message-driven environment eased the implementation of Sagas

[14] which are not supported in any traditional RDBMS. The extra resources are said to be compensated by enhanced scalability, the simplification of a complex aspect of your domain, and greater adaptability to change business requirements.

There is also an issue where separation of query and command segregation should take place. In general we have two options;

- Database Replication mechanism can be used – from development perspective this solution is cheap, but the control over the synchronization is an obvious trade-off. In many cases we are not able to control the latency in any way.
- Application layer synchronization – which is often paired up with Message Brokers, given they are a single-point-of-entry. This solution is often custom made, as no frameworks exist that are dedicated to in-fly data remapping. This approach was used in [2] where the layer was named Synchronization/De-normalization Component.

Considering segregation of data stores CQRS can be paired with Function-As-A-Service (FaaS) or Serverless model when direct access to the read-optimized/write-optimized model is provided.

3. Experiments

In this section we describe two experiments that were carried out in order to estimate whether the CQRS approach to build Software is beneficial. First experiment is simple and tries to capture the fundamental benefits of CQRS architecture. Second experiment is based on TPC-E benchmark. The tests were executed on Windows 10 Education 64-bit with Intel® Core™ i7-6500U CPU and 16 GB RAM. The database used to simulate the CQRS-like data access was SQL Server 2017 Developer Edition with Database Replication feature. The database buffer pool size was set to 400 MB and no custom trace flags were set on.

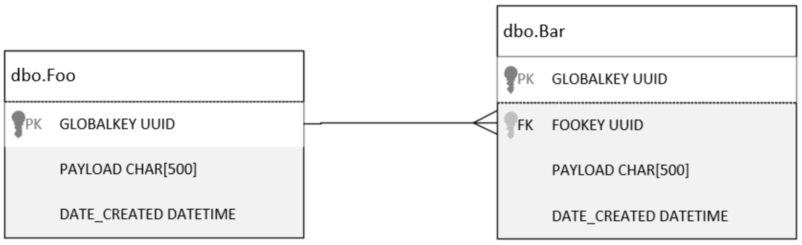

Fig. 2. Database schema for the CQRS Simple Test
Rys. 2. Schemat bazy danych dla uproszczonego testu architektury CQRS

3.1.Simple test

The aim of the simple test was to assess whether data access does benefit when using CQRS architecture. The database schema for the test consisted of two tables: Foo and Bar as in the figure 2. The test was split into two phases: write and read, both were measured separately. In the write phase the Java program inserted the records into the tables Foo and Bar. In the second phase the database was queried as follows:

```
SELECT
    F.GLOBALKEY AS FOOKEY,
    COUNT_BIG(*) AS GROUP_COUNT
FROM
    dbo.FOO F
    INNER JOIN dbo.BAR B ON F.GLOBALKEY = B.FOOKEY
GROUP BY
    F.GLOBALKEY
```

Three configurations of database were checked:

- Configuration 1: which simulated CQRS-like access. Using Heap Files as storage with PRIMARY KEY constraint on GLOBALKEY columns in the write database. Write database was replicated asynchronously to the read database which had Indexed View (Materialized View) for query presented previously.
- Configuration 2: which simulated classic access. Using B+ trees clustered on Foo.GLOBALKEY and Bar.FOOKEY. This setup had single database.
- Configuration 3: also classic approach but had different set of indexes. Using Heap Files with indexes on Foo.GLOBALKEY and Bar.GLOBALKEY. This setup also had single database.

In the first phase, the test program had inserted records with latency $l = 100\ ms$ between every insert transaction. Every 100 transactions the l was lowered by $1\ ms$ – so that initial stream of transaction was operating at 10 TPSE ($l = 100\ ms$), through 1000 TPSE ($l = 1\ ms$) and finally up to As-Fast-As-Possible stream ($l = 0\ ms$). Two clear disadvantages of CQRS has been highlighted:

1. The Eventual Consistency latency as presented in figure 3. Till $TPSE < 30$ ($l = 33\ ms$), the replication latency is lower than 1 second. When $TPSE$ goes over 30 the latency starts to rise as the new transactions queue. When the transaction stream becomes unbounded (Transaction group 100), the latency begins to rise exponentially. **This is crucial disadvantage – as the Read Database is inconsistent during the latency period and should not be queried**. In our test the latency goes as high as 12 seconds. Interestingly enough, execution time seems not to be influenced by the replication mechanism and value of TPSE.

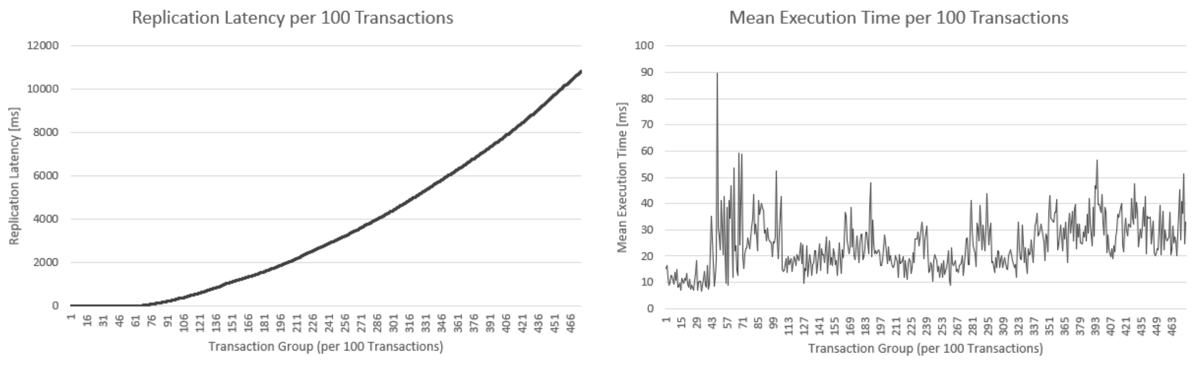

Fig. 3 Mean replication latency/execution time of write transaction per group of 100 transactions
Rys. 3 Średni czas opóźnienia replikacji/wykonania transakcji zapisu w grupach po 100 transakcji

2. The data needs to be replicated. Vanilla, single database instance in Configuration 2 of our test required 411.27 MB for Bar table and 77.38 MB for Foo table which sums up to 488.65 MB of space required. The CQRS setup required 382.08 MB for Bar and 77.45 MB for Foo tables in the Write-Optimized Database. The Read-Optimized Database required 382.08 MB for Bar and 81.88 MB for Foo tables. This sums up to 923.37 MB of space required. In this case this means 188% greater space requirement.

However, as we expected the efficiency of write operation are visible as presented in table 1 which contains summary for the obtained results without outliers (range $q_{.95}$-Max of original dataset was discarded).

Table 1
Summary of execution times of write transactions (in milliseconds)

Configuration	Min	$q_{.25}$	$q_{.5}$	Mean	$q_{.75}$	Max
1	0.0	5.0	9.0	21.89	23.0	122.0
2	0.0	8.0	26.0	58.6	101.0	218.0
3	0.0	31.0.	33.0	73.59	113.0	192.0

The efficiency results of the second phase of the experiment (read phase) are presented in table 2 which presents summary for the obtained results without outliers (range $q_{.95}$-Max of original dataset was discarded).

Table 2
Summary of execution times of read transactions (in milliseconds)

Configuration	Min	$q_{.25}$	$q_{.5}$	Mean	$q_{.75}$	Max
1	0.0	0.0	1.0	2.614	6.0	8.0
2	0.0	0.0	2.0	7.481	16.0	32.0
3	0.0	1.0.	1.0	4.589	6.0	714.0

We used the Wilcoxon test on $\alpha = 0.05$ to establish that in every case (Read/Write) execution times in Configuration 1 were significantly lower than ones in any other Configuration (also when full-set was considered). The normality of distributions cannot be assumed due to the complex nature of data retrieval from DBMS (i.e. hard to predict if access happens from buffer pool or disk).

3.2.TPC-E

The second test aims to compare Non-CQRS and CQRS setup in conditions similar to real-world application using TPC-E benchmark. It is has 10 operational transactions with different complexity and size, 2 maintenance transactions and 33 tables. To perform tests, We implemented benchmark according to specification available on the benchmark web page [15]. However, one should remember that our test harness is not 100% compliant.

Generally, TPC-E is read oriented because 77% of workload is made from Read-Only transactions (6 of 10 TPC-E transactions are Read-Only). Such composition agree with assumptions about typical CQRS use cases. The technology stack that we choose for realization of driver was composed of pure Java in version 10 and JDBC driver for connecting with DB. To test the efficiency of CQRS architecture, we defined two configurations:

1. Configuration 1 (CQRS) – which was a combo of Read-Only and Read-Write DB. The synchronization between databases was implemented using Database Replication feature. The write schema was stripped down from Clustered Indexes in favor of Heap Files with auxiliary PRIMARY KEY indexes. To leverage UPDATE and DELETE statements the indexes were build up. The read schema was optimized with dedicated Materialized Views for every SELECT statement.
2. Configuration 2 (Non-CQRS) – which was a single database instance which had TPCE database with original schema. All tables were implemented as B+ trees clustered on PRIMARY KEY/FOREIGN KEY almost in every case. Additional indexes were build up based on SELECT, UPDATE and DELETE statements.

The experiment for each of the tested environments contained 250,000 transactions, mixed in proportions given in TPCE documentation. Transactions were executed from a single thread sequentially. During the experiment there was increase in size of database size although this could be ignored as it stands only for a small part of whole DB size. For every transaction we measured transaction start t_s and transaction end t_e. Difference of these timestamps is execution time $e = t_e - t_s$, which is our main efficiency measure. Figure 4 presents the histograms of e for Read-Only transactions in both setups.

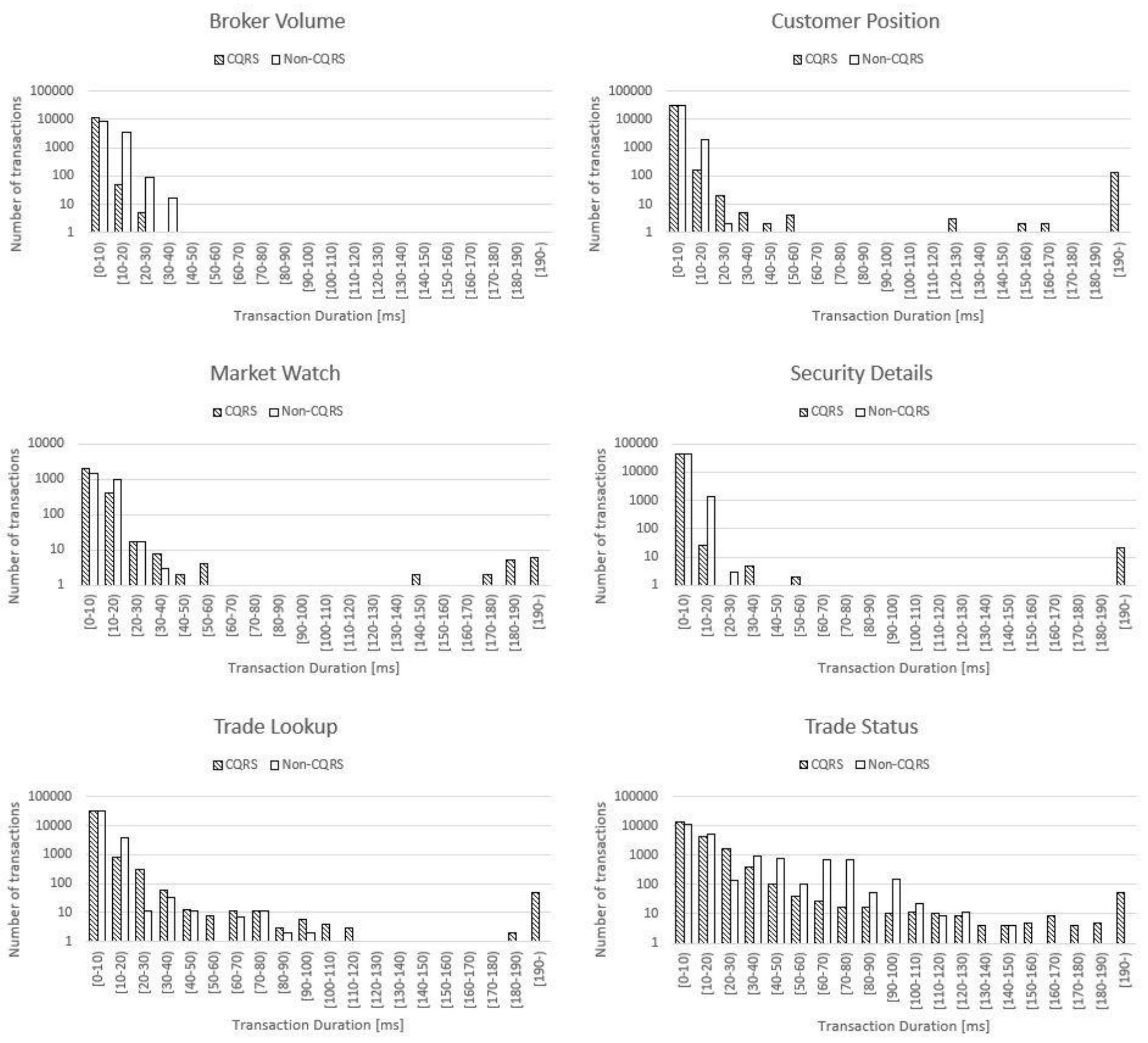

Fig. 4 Execution times histogram of Read-Only transactions of Non-CQRS/CQRS setup
Rys. 4 Histogram czasów wykonania transakcji tylko-do-odczytu konfiguracji Non-CQRS/CQRS

The efficiency varies between different cases and high spread of time values could be observed especially in those transactions performed on CQRS setup. Interestingly, CQRS has much more high-value outliers – as database often locks during the synchronization process. In both environment for Read-Only operations for 5 transactions more than 95% of cases falls into first two buckets, in case of Trade Status transaction achieved results are slightly lower. The results are quite similar for both of the tested setups. Nonetheless, We are careful about general assumption about CQRS efficiency as some *e* are very similar in both setups. One should note that in our test We did not verify the consistency of the queries – and basing on the results of previously executed Simple Test, we should take additional few seconds for Read-Only DB to "catch up" with the changes.

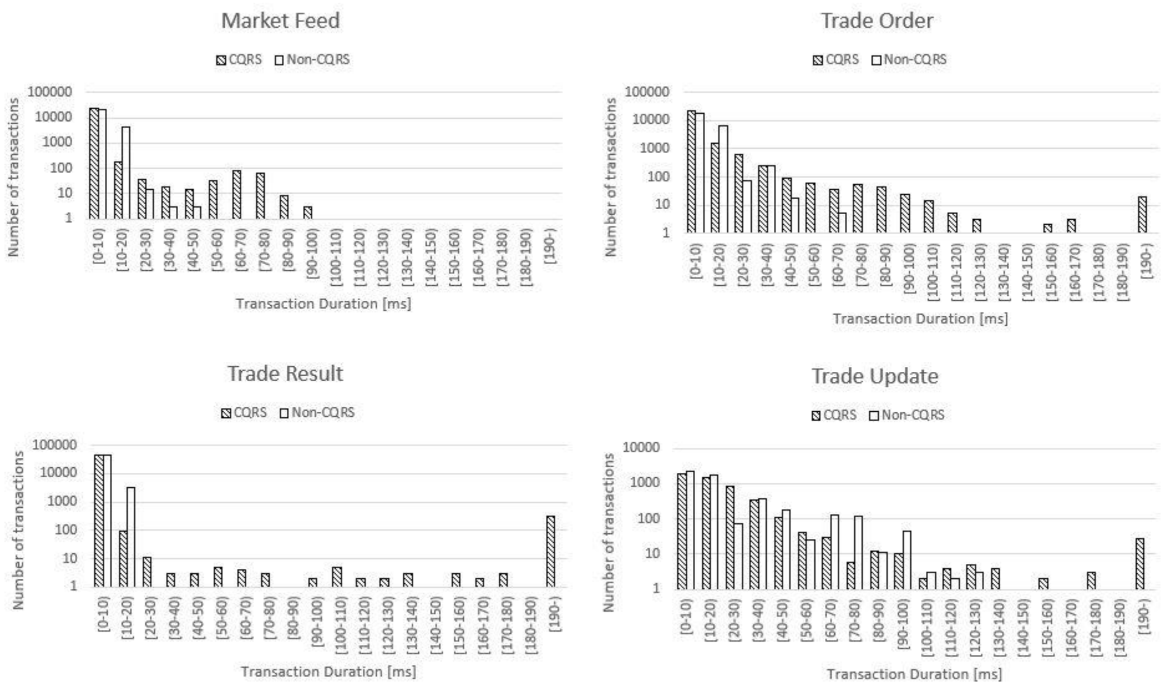

Fig. 5 Execution times histogram of Read-Write transactions of Non-CQRS/CQRS setup
Rys. 5 Histogram czasów wykonania transakcji zapisu-odczytu konfiguracji Non-CQRS/CQRS

The Read-Write transactions characterized overall longer times for both tested setups. In the context of comparing CQRS and Non-CQRS the time quotas were lower for second one. Wide distribution across the horizontal axis is especially true for CQRS transactions. Lower complexity of the Non-CQRS setup gives much more predictable results as outliers with high transaction time are almost non-existent.

Comparing conducted test most of the transactions 95th percentile composed of transactions from the first two buckets. In order to have more intuition about size of the discrepancy in individual transactions We took summary times. Equation calculate difference between total execution time of CQRS T_C and Non-CQRS transaction T_{NC} divided by summed time of Non-CQRS transactions T_{NCS}. This can be written as: $\Delta T = \frac{(T_C - T_{NC})}{T_{NCS}} \cdot 100\%$.

Some of transactions have bigger share in number of all transactions what allowed them to grow the difference more significantly. However, when focusing on total timing it is in favor of Non-CQRS system which had better performance in the majority of transactions.

There are two reasons which could be also taken into consideration when assessing test results. Main test assumptions about OLTP character of transactions comply with suggested use cases. Next one is about test seriality which differs from conditions found in the real world systems. We are precautions to name the better solution but some tangible results were shown that allows to expose real costs and benefits of using CQRS pattern.

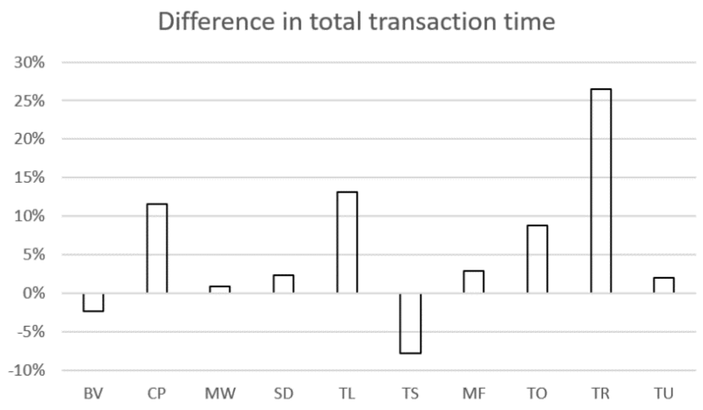

Fig. 6 Percentage difference between the total transaction duration times in CQRS and Non-CQRS setup
Rys. 6 Procentowa różnica pomiędzy całkowitym czasem trwania transakcji dla konfiguracji CQRS oraz Non-CQRS

4. Conclusion and Further Research

As for now Our concerns about problems connected with CQRS architectural pattern and one of proposed ways to implement it in production were confirmed. The performance was affected by the necessity of data distribution between two data stores introducing new degradation factor. But eventual consistency is the property which could be taken for granted when deciding on CQRS. There were seen also theoretically great advantages of CQRS setup when looking at Simple test results. Even a few times shorter transaction execution times was impossible to achieve when working with more complex transactions. When looking on TPC-E test results CQRS Architecture was outperformed by almost 60% by Non-CQRS Architecture – even when the consistency was not taken into account.

We hold the perception that CQRS Architecture is not a good alternative for more traditional solutions such as Three Tier Architecture in the general case. Nevertheless, it might be useful when applied to bounded context that want to take advantage of its properties. Also lack of dedicated software solutions could be the field in which further improvements can be made. There is no doubt that in order to put those systems to production the measurable benefits should outperform other solutions or be more approachable when it comes to the complexity of solution.

Further study might involve proposing the formal efficiency model for software build upon CQRS and performance evaluation in the distributed environment. The efficiency might be hard to examine as our present methodology of implementing CQRS resulted in many database locks in Read-Only Database which decreased the overall performance.

BIBLIOGRAPHY

1. Woźniak A.: Application of Big Data in Poland and in the world, Studia Informatica Vol 37, No 1, 2016.
2. Rajkovic P. , Jankovic D., Milenkovic A.: Using CQRS Pattern for Improving Performances in Medical Information Systems in BCI (Local), volume 1036 of CEUR Workshop Proceeding, 2013.
3. Meyer B.: Object-oriented software construction, second edition, 1988, pp. 748-764.
4. Robert M. C., Micah M.: Agile Principles, Patterns, and Practices in C#, Prentice Hall, pp. Chapter 8: The Single-Responsibility Principle (SRP).
5. Dahan U.: Udi Dahan – The Software Simplist, Enterprise Development Expert & SOA Specialist, 2009-12-09. [Online]. Available: http://udidahan.com/wp-content/uploads/Clarified_CQRS.pdf. [Accessed 24 February 2019].
6. Betts D., Dominguez J., Melnik G., Simonazzi F., Subramanian M.: Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure, 1st edition ed., Microsoft patterns & practices, 2013, pp. 223-242, 248-249.
7. Fowler M.,: CQRS," 14 July 2011. [Online]. Available: https://martinfowler.com/bliki/CQRS.html. [Accessed 24 February 2019].
8. Dahan U.: When to avoid CQRS, 22 April 2011. [Online]. Available: http://udidahan.com/2011/04/22/when-to-avoid-cqrs/. [Accessed 22 February 2019].
9. Nilsson M., Korkmaz N.: Practitioners' view on command query responsibility segregation, 2014.
10. Karwin B.: SQL Antipatterns: Avoiding the Pitfalls of Database Programming, 1st edition ed., Pragmatic Bookshelf, 2010, p. Chapter 9: Metadata Tribbles.
11. RabbitMQ, "RabbitMQ," Pivotal, [Online]. Available: https://www.rabbitmq.com/. [Accessed 24 February 2019].
12. Apache Kafka, [Online]. Available: https://kafka.apache.org/. [Accessed 24 Feburary 2019].
13. Erb B. and Kargl F., Combining Discrete Event Simulations and Event Sourcing, in SIMUTools 2014 - 7th International Conference on Simulation Tools and Techniques
14. Garcia-Molina H., Salem K.: Sagas, in Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, 1987.

15. TPC.: Transaction Processing Council TPC-E Benchmark, [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-e_v1.14.0.pdf. [Accessed 24 February 2019].

Omówienie

Artykuł ten porusza temat wydajności architektury Separacji Odpowiedzialności Komend i Zapytań (ang. CQRS). Testy zostały przeprowadzone w sposób umożliwiający porównanie z systemami opartymi o pojedynczą bazę danych. Zwracamy uwagę na zalety stosowania tej architektury: zmniejszenie średniego czasu odpowiedzi i możliwość potencjalnego skalowania wszerz, oraz wady: zwiększona złożoność systemu, brak frameworków wspierających tworzenie aplikacji oraz pojawienie się zjawiska latencji pomiędzy zapytaniami, a komendami. Na przykładzie dwóch scenariuszy testowych: uproszczonym oraz złożonym (zbudowanym w oparciu o benchmark TPC-E) zaobserwowane zostały pewne zjawiska dot. tej architektury. Na mało złożonym przykładzie pokazujemy, że zastosowanie tej architektury znacząco zmniejsza czas odpowiedzi systemu zarówno na zapytania jak i komendy. Niestety im bardziej intensywny jest strumień transakcji oraz im bardziej rośnie złożoność transakcji, tym większe opóźnienie transferu danych pomiędzy bazami do-odczytu i do-zapisu, co można zaobserwować w bardziej złożonym scenariuszu testowym. Podczas wykorzystania architektury CQRS dochodzi koszt duplikacji danych, który się maksymalizuje, jeżeli rozmiary baz danych do odczytu oraz zapisu są podobne.

Addresses

Mateusz NIEDZIELA: Rzeszow University of Technology, The Faculty of Electrical and Computer Engineering, ul. W. Pola 2, 35-959 Rzeszów, Poland
Maciej PENAR: Vulcan Sp. z o.o., ul. Wołowska 6, 51-163 Wrocław, Poland, Maciej.Penar@vulcan.edu.pl