Evaluation of Data Storage Patterns in Microservices Archicture

Munonye K*
Budapest University of Technology and Economics/Computer Science/
Budapest, Hungary
kanymuno@gmail.com

Martinek P**
Budapest University of Technology and Economics/Computer Science/
Budapest, Hungary
martinek@ett.bme.hu

***Abstract*—A microservice is an architectural style that structures an application as group of loosely-coupled services that are independently deployable and centered around the business capabilities. Since microservices are a shift from the conventional monolithic application which normally uses a relational database backend, it becomes essential that attention be given to data storage structure for Microservices. In this research, we would examine five possible data storage patterns for microservices. The evaluation would include setting up prototypes and evaluating the performance of both RDBMS and document-store enabled data stores for microservice architecture. To carry out the evaluations, the five existing patterns for data persistence in microservice architecture were modeled: shared database, per-service pattern(3 variations) and the CQRS/EV pattern. Then a simplified hypothetical Electronic Medical Record(EMR) was designed and implemented using of the five existing patterns under consideration. The results of the evaluation in terms of performance and resource utilization are presented in this work.**

Keywords—Enterprise Application Integration(EAI), Microservices, REST API, Shared Databases, Database Per Service.

I. Introduction

Before we discuss, the database storage patterns, we first present an overview of the microservices architecture(MSA) as this would enable a clear understanding of the need for efficient data storage. Microservices provides a framework for developing enterprise applications as a suite of services which are independently deployable, implementable in different languages and platform and can be managed by different teams [1] [2].

Compared to the conventional monolithic architecture, microservices, among other benefits, provides a more flexible and scalable application with each service centered around a single function [3]. This enable the application to be scaled in a more granular level with little or no impact to the availability.

The microservice architectural pattern partitions and information system into a set of microservices [4] [5]. In the context, the complexity of each services is low enough to be understood by a relatively small team of developers or even a single developer. Being self-contained, microservices, contains all the components needed to be deployed and used in a production environment without having to depend on other services for its functionality. Therefore, strictly speaking, they do not share source codes or database schemas with each other although there are scenarios where interface with a common datasource as highlighted in this research in cas of relational and document store shared databases.

Each microservice can be implemented using a different programming language paradigm, middleware or data persistence framework. Finally, microservices have been succesfully applied in deveopment of large information systems. For example the OceanTea project discussed in [6] for exploring ocean-derived climate data.

Figure 1 show a typical microservices architecture made up of three basic sections:

The Access Layer which provides an interface for users to connect to the services, the API Gateway which allows for discovery of available services and routing of user requests to particular microservices and the data repository, which could be either abstracted as another layer shared by all the service(as in Figure 1) or collection of individual databases built into each microservice.

This research would be examining the three categories of data access patterns described above: the shared database pattern, the per-service pattern and the CQRS-ES pattern

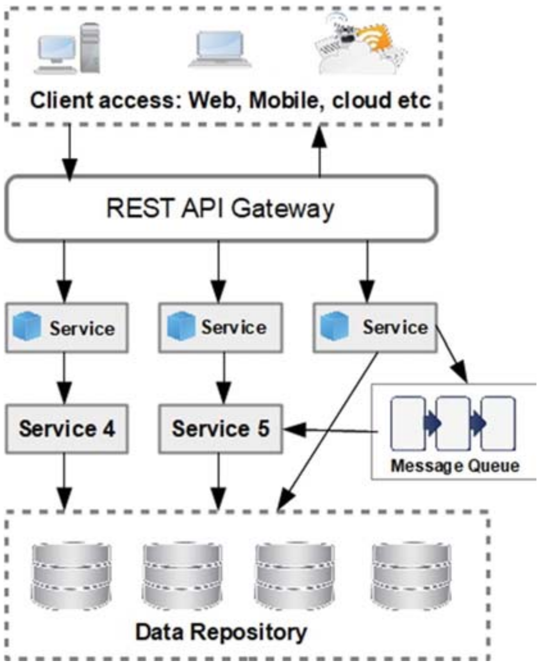

Figure 1: Typical Microservice Architecture

While microservices offer a wide range of benefits over the monolithic architecture, other challenges arise relating to complexity, inter-communication between services, data sharing and others which are outlined in [7]. However, we would focus on data-related challenges and their solutions via data persistence patterns.

The rest of this paper is arranged as follows: Section 2 provides an overview of the challenges of data storage for microservices. Some of the microservices data storages patterns are described in Section 3. In Section 4, the methods and setup of the evaluation environment is presented. Then in Section 5, a discussion of the results obtained as well as the criteria for reliability is offered. Finally, in Section 6, the conclusion and possibilities for future research are reported.

II. DATA STORAGE CHALLENGES FOR MSA

Unlike monolithic application which maintains performance and scalability on relational database model, performance microservices is impacted for serveral services having a single entry-point for data access.

In design of microservices-based information systesm, one of the key challenges is the partioning of the application into separate services, each having a dedicated data store. Each of these services should be light-weight and small compared to the whole and be simple enough to handle just on business function.

Maintaining efficient performace while keeping the services partitioned is another key challenge for MSA. One of the recommendations for handling this is the use of result cache[8]. This allows for services making network calls to remote databases to use a result cache on the client side to limit calls made to external data services.

Ontology-based Data Access (ODBA) paradigm described in can be used to provide the users with access to the data store via a conceptual layer between the services and the actual database[9].

As observed by [8], performance can be improved by adopting set of optimization strategies for data storage and having a formal framework for data-aware integration. This process optimization can be complicated by the fact detailed data access and communication processed are not always present due to data isolation. The complexity is further amplified by the fact that distributed transaction would have to span multiple services.

Data integrity concern is also a common challenge in microservcies adoption since each microservice being responsible for its own data. As a result maintaining data consistency and integrity has to be handled by a trade-off with performance as is proven by this research.

Data integration patterns for microservices always face the decision of how process-driven the integration solution would be as as opposed to data-driven. This is highlighted in [9] and suggests that computation can move from process-driven to data driven for particular services and vice versa based on identified recurring communication patterns.

Then the challenge of transaction management in distributed systems which applies to monolithic applications also applies to microservices. While a 2-phase commit process is recommended for centralized database, there need to be a way to optimized data sychronization for inter-site data traffic [10]. The significance of the issue of transaction management is further highlighted section IV of this research.

III. MICROSERVICES DATA STORAGE PATTERNS

We now outline existing databases storage patterns and also possible combinations of this pattern to yeild improved performance. We would examine the following five patterns, namely: Database Per Service Pattern, Schema Per Service Pattern, Private Tables Per Service Pattern, Shared Database Pattern andd CQRS/Event Sourcing

These five patterns could be grouped into categories. :

A. the per-service pattern,

B. the shared database and

C. the CQRS and Event Sourcing.

The architectural framework of three main categories are now discussed. Then we also consider a few enhancements to these patterns.

A. Shared Database

In this pattern, a single database is shared by multiple services and as such each services accesses the same global pool of data. As discussed in chapter 4, this pattern provides the best peformance in terms of response time but does not align with the principle of loose-coupling which is a fundamentals aspect of microservices design. Besides, it significantly lead to increase in complexity in scenario where the services are based on disperate database platform such as document store and other NoSQL technologies.

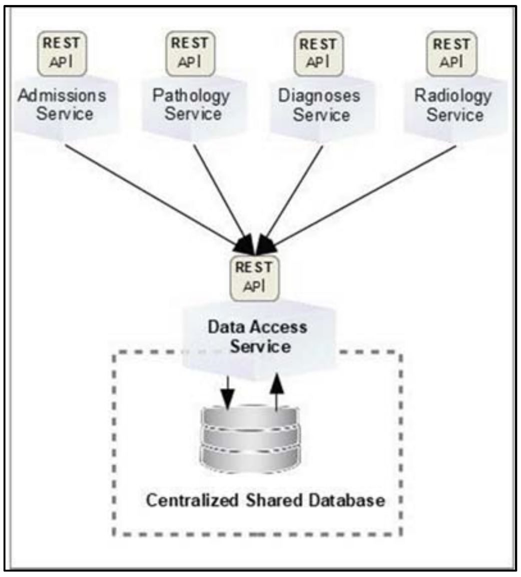

Figure 2: Simplified Architecture of the SDB pattern.

B. The Per-Service Pattern

In this approach, some data storage container is provisioned for the specific service such that the data within this container is accessible only by the service that own it. The Per Service pattern could be either of the three implementations: *(1) Private Tables per* service where each service owns as set of talbes that must be accessed only by that service *(2) Schema Per Service*(not available for some database platforms) where each service has a database schema that's fully privated to that service *and (3) Database Server per service* where each service owns it's own database server.

The per service pattern in illustrated in Figure 3 with four different services each having it's distint data storage. Figure 3 shows additional components of the microservices architecture such as the load balancer and the service discovery layer. These are not discussed here as it's not the subject of this research. However as shown in the Figure 3, a complete isolation is maintained as each service does not have direct access to the data store of other services.

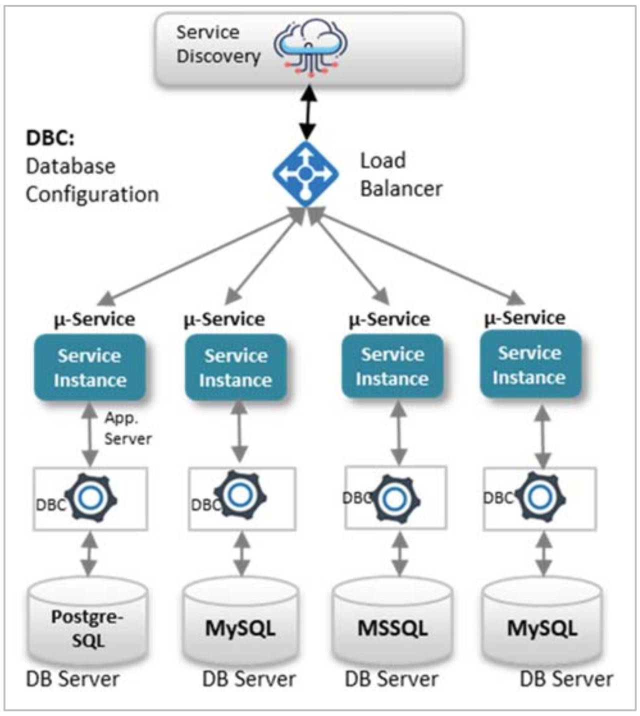

Figure 3: The Per Service Pattern for (DBPS)

Of the three per-service patterns, the private tables per service has the lowest overhead in terms of resource consumption but is not supported by a number o database platforms including MySQL, PostGreSQL and HyperSQL.

C. *CQRS and Event Sourcing Approach*

CQRS which is the most recent data storage pattern for MSA stands for Command Query Reponsibility Segregation. This pattern represents a way to split data access into two distinct parts: the commands which represend operations that modify the state of the data and queries which represents a read-only request for data.

Figure 4 shows a basic outline of the CQRS and ES architecture. The presentation layer comprises of Views, Controller and Materialized views. The materialized views component represents trhe read-only view of the data while the controller/view component represent entrypoint for both commands and queries.

Commands are routed through the command gateway to the command handlers which execute the operation to modify the data via the repository interface. A modification in the repository emits an event which updates the read database. Similarly, queries are passed through the query gateway to the read model. The entities representing the data model are known as aggregates.

The operations that read the data are segregated from the operations that write or update the data by separate interfaces. This implies that the data models used for querying the entities and updating the entitites may also be different, although this is not a requirement. Therefore, both a read store and a write store are implemented and would requires some kind of synchronization. This is achieved via events. An Event is a notification that a change have occurred in the write store and such events contain the attributes that have been modified. So each time a change is made, the and event is emitted. Through event handling components as shown in Figure 4, this a corresponding change is effected int the Read store and so at any point in time both the read and write store contains the same data. The principles of CQRS can be summarized as follows:

- Data storage is split into two parts: the read store and the write store
- Entities in both stores have to be in sync
- Changes made in te read store are published as events
- Event handlers receive events and updates the read store based on the information contained in the event.

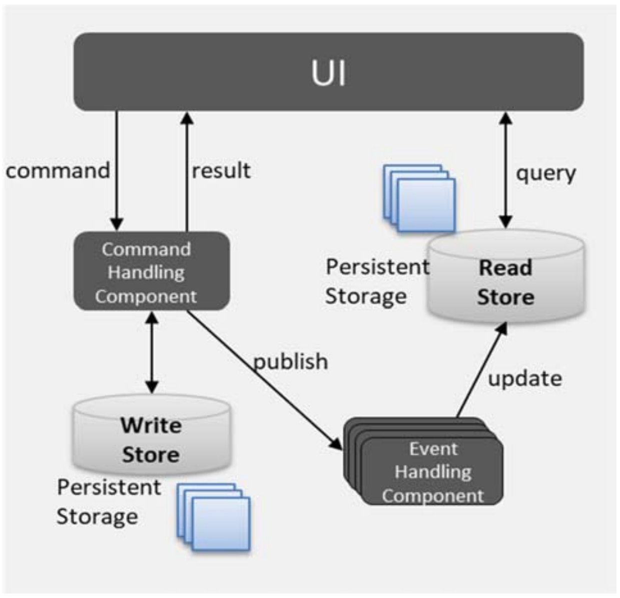

Figure 4: CQRS and ES Basic Architecture

D. *Enhancements to Data Storage*

Additional approaches exist which to augment the patterns described above and enhance the performance of data storages in microservices. They included method to improve scalability, performance and data consistency. Some of them are outlined in this section.

Due to the need to support a number of divers services and also support concurrent users, architecture of microservices require a well-defined approach to data modelling for significant data-processing throughput to support scalability and performance [11].

One of the recommendatations for microservice data storage pattern is the result cache [12]. This allows for services making network calls to remote databases to use a result cache on the client side to limit calls made to external data services.

Ontology-based Data Access (ODBA) paradigm described in can be used to provide the users with access to the data store via a conceptual layer between the services and the actual database [13].

With respect to maintaining a consistence of data a stateful approach was adopted by [14] using transacted resource-bound patterns. This method ensures eventual consistency as transactions are replicated and commited in other databases based on event triggered by other event.

IV. EXPERIMENTAL SETUP

In this section, an overview of the methods and test environment setup is provided. Then further details on the key concepts of the models being evaluated are discussed.

A. *Evaluation Criteria*

A number of criteria exists against which MSA can be evaluated. This includes: data access performance, data integrity, transaction management, scalability and query execution time. In this research, we would focus on two of these criterias:

- Query Execution Time(QET) which is the time taken to executed a distributed query accessing data across service boundaries
- Resource Utilization which is the impact of transactions on system resources. Since a test environment was used, we would focus on memory, CPU and swap space used

B. *Experimental Design*

The first step was to design a prototype using the five architectures described in this research. For each prototype, a relational data model(using MySQL) and a document store NoSQL model(using MongoDB) was implemented in turns. A prototype REST API was developed based on the architecture in Figure 5 to test the query-response time of the databases. The API acts as a single service and maintains a driver for each of the databases under consideration.

To simplify the design and implementation of our hypothetical ERP system, we limit the number of module to four namely:

- Admissions module (Microservice 1)
- Diagnosis module (Microservice 2)
- Pathology module (Microservice 3)
- Radiology module (Microservice 4)

The simplified Database-Per-Service(DBPS) architecture of the ERP system is shown in Figure 2. In this pattern, the client could access the microservices via a service discovery interface exposed through REST APIs. The service discovery interface routes the incoming request to the appropriate microservice. Each microservice maintains a light-weight database which is independent of each other.

A Model-View-Controller(MVC) architecture was adopted for this design by injecting the Spring-MVC dependency to the application classpath.

For this research, the architecture of Figure 2 represents both the DBPS architecture for both Document store DB(H2) and Relational DB(MySQL).

For simplification, orchestration framework used in this implementation (Kubernetes) as well as deployment-related details was not included in the diagram. Also, related details of service discovery tools has intentionally been omitted to allow focus on the data model under consideration.

Each of which corresponds to a microservice. Moreover, each of the modules/microservice was designed as a full-fledged standalone application.

The application was developed in using Spring Framework using the Spring Tool Suite (STS) IDE, an Eclipse-based IDE by Apache Software Foundation.

The dependencies were managed using Maven through the project object module (pom) file in the application directory.

For the shared database and database-per-service pattern, Java Persistence API was used to connect to a MySQL database.

Object-Relational Mapping (ORM) was achieved using Hibernate which is an ORM tool for Java programming language and provides a framework for mapping object-oriented domain model to a relational database.

For the document store database pattern, H2 document store database was used. The H2 relational database management system (RDBMS) was chosen since consistency can be maintained with the Spring Framework and it can be embedded in a java application.

Communication between microservices was achieved using REST API requests and responses. This was implemented using REST Template available in Sprint Framework which allows for synchronous communication between the microservices. REST template was chosen over REST client to allow for inspection of time it takes for requested data to be returned to the client.

A Simplified architecture for the Shared Database pattern used for this research is presented in Figure 3 (the API gateway and Client section has been removed for simplicity). This represents the implementation for both relational and document store design.

While a database cluster could be used as applied in enterprise environment, for this research, a single database was designed with provides several stored procedure callable from a data context in the application.

For the test environment, we deploy the four services to Apache Tomcat server in the local machine. To simulate an enterprise environment, each of the service were made to run on four different HTTP ports (8081-5). We could also see from Figure for that the Shared Database pattern incurs an additional microservice, which serves an entry-point between the database and existing microservices.

C. *Database Structure*

The structure of the database is presented in Figure 4 for both the SDB pattern and the per-service pattern. From Figure 4, we can easily see that adopting DBPS (the microservice architectural style) clearly leads to duplication of data in multiple data stores.

Figure 4 illustrates the challenge of data access with database-per-service architecture. The relational nature of the databases and the requirement for isolation for each services' data means that transactions would have to span multiple services.

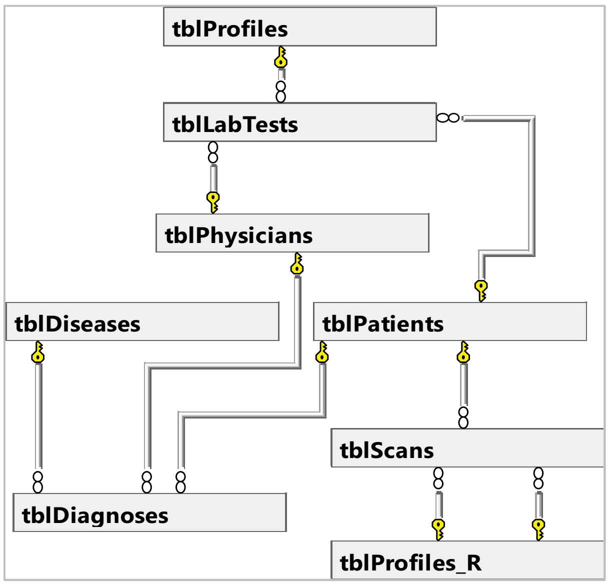

Figure 5: Database Design for our hypothetical ERP

Let's assume an update query, U_0 executed against the *tblProfiles* table take QET = T_0 ms.

For the best case scenario, only the tblProfiles table is updated and that gives a total QET T_0 ms.

If however U_0 triggers a update $U_{0,1}$ in the related table *tblLabTests* with time $T_{0,1}$, then total QET would be given by:

$QET_{total} = T_0 + T_{0,1}$ ------ Equation 1

Therefore for the, the total QET for any query operation would for a database with *n* tables would then be given by:

$QET_{total} = T_0 + T_{0,1} + \ldots + T_{0,n}$ ------ Equation 2

Where T represents QET for a query executed on a single database table while the subscripts represents the level of the particular table in the query execution profile.

From Equation 1 and Equation 2 it can therefore be deduced that the QET for any given query if of the order of the number of tables contained in the database (O(n)).

To verify this, we would execute a number of pre-defined test-cases against the ERP database based on the convention listed in the following sub-section.

D. Convensions

To ensure an accurate and precise measure of the query execution time as well as to simplify the analysis process for the patterns considered, the following conventions were adopted:

- It is assumed that the query issued to the database provides a response for the request without consideration to the client-side result caching. The use of result cache is considered as a different implementation pattern [12]
- Latency arising from network environment was considered constant for all the patterns considered and therefore is not included in the metrics.
- Database cluster load balancing was not included in the analysis and therefore no performance effects results from load-balancing.

V. RESULTS AND DISCUSSSION

In this section, the results of the performance of the three data persistence patterns are presented. For brevity, we present the most-significant results obtains. In the first three sub-sections, we examine how the patterns compare and contrast while in the last two sub-sections, we outline the performance matrix for all the patterns considered.

The query execution performance (QEP) of the five data persistence patterns were compared in a pairwise form as follows:

A. RDBMS vs Document Store (MongoDB)

B. CQRS vs the Per Service(PS) Pattern

C. Shared DB vs Per Service(PS) Pattern

D. Performance Matrix for all Patterns

E. Resource Consumption Matrix

The Query Execution Time (QET) recorded for each database pattern for each operation represents the averages of a series of test cases executed against the database.

A. QEP for RDBMS vs Document Store(MongoDB)

The result presented in Table shows the performance outputs for the relational databases patterns. Several test cases were executed against the deployed application. Each test case consists of a HTTP request which includes a server side markup for data access.

Taking an average of all the tests carried out indicates that for a given pattern, the use of a document-store database(MongoDB) performed better than RDBMS with all related factors kept constant. Table 1 shows an improvement of 28.96%. However, it's also observed that the DBPS pattern works show much better performance for Batch Select, and Parameterized Search operations with an improvement of 45.54% and 25.10% respectively.

We also see from Table 1 an improvement of the Document Store over RDBMS for batch update, insert and delete operations with an improvement of 13.08%, 23.74% and 37.34% respectively. These could be considered a relative decrease in performance for DBPS pattern. These measures are significant looking at the execution trace of these operations. A high execution time is incurred for these operations since update, insert or delete of data in another database could be triggered by the insertion in a specific database. So in production scenarios, the document-store database is preferred since it aligns more with the principle of loose coupling and modularity with microservices is based on.

Table 1. Response Time RDBMS vs Document Store(DS)

Operations	**Avg. Query Execution Time (milliseconds)**		**Diff-erence**	**Improved Pattern**
	RDBMS	**DS**		
Select	729.05	500.93	45.54%	DBPS
Update	690.25	780.55	13.08%	DS
Insert	792.05	980.09	23.74%	DS
Delete	350.00	480.69	37.34%	DS
Parameterize d Search	400.50	320.14	25.10%	DS
Avg. QET Performance	**592.37**	**612.48**	**28.96%**	**DS**

B. QEP for CQRS vs Per Service

The result presented in Table 2 shows query execution times for relational DBPS pattern compared to Shared Document store database pattern. In this case, 100,000 test cases were executed with the CQRS pattern as well as the Per Service pattern with. This was carried out in turns, first with the RDBMS and then with DS. The average value of the QET was computer.

The comparison between these two was chosen based on the fact the both the CQRS pattern and the Per Service pattern aligns with the principle of loose coupling. For the set of operations, for the test cases executed, it could be observed that the CQRS pattern showed an overall improvement over the relational DBPS with an improvement of 44.21%.

Table 2. Response Time for CQRS vs Per Service

HTTP Operation	Query Execution Time (milliseconds)		Diff-erence	Improved Pattern
	Per Service	CQRS		
Select	729.05	500.93	45.54%	PS
Update	690.25	780.55	13.08%	CQRS
Insert	792.05	980.09	23.74%	CQRS
Delete	350.00	480.69	37.34%	CQRS
Parameterized Search	400.50	320.14	25.10%	CQRS
Avg. QET Peformance	**592.37**	**612.48**	**28.96%**	**CQRS**

C. QEP for SDB vs the Per-Service Pattern

The result presented in Table 3 shows performance for the two Document store databases considered. The results of Table 3 indicates an average improved performance of Shared-Document store database pattern over DBPS pattern with an improvement of 32.94%.

The data in Table 3 clearly shows that the DBPS pattern performed better for batch update, insert and delete operations with an improvement of 20.9%, 11.06% and 59.12% respectively.

Table 3. QET for SDB vs the Per Service Pattern

Operations	Query Execution Time (milliseconds)		Diff-erence	Improved Pattern
	Shared-DB	DBPS		
Batch Select	729.05	500.93	45.54%	DBPS
Batch Update	690.25	780.55	13.08%	SDB
Batch Insert	792.05	980.09	23.74%	SDB
Batch Delete	350.00	480.69	37.34%	SDB
Parameterized Search	400.50	320.14	25.10%	SDB
Avg. QET Peformance	**592.37**	**612.48**	**28.96%**	**SDB**

D. Performance Matrix for all Patterns

An overall view of the relative performance of five of the patterns discussed is presented in Table 4. For the test scenario, 100,000 test cases (HTTP request) were executed with an average payload of 5,303 bytes (constant for all the requests. Table 4 indicates the response times range for all the five patterns considered.

It could be observed in this case, that SPS with document store pattern exhibits the best performance on average with a total an average time of 476.12ms for all the test cases. This figure represents a 52.5% improvement in performance over the SDB pattern, a 36.8% improvement in performance over the DBPS relational model and a 25.04% improvement in performance over the DBPS Document store model.

It could also be observed from Table 4 the QEP for transaction operation improves for Share DB pattern both for relational and document store databases. For the shared relational database, we have an a QET of 804.2ms for shared against 835.47 for DBPS. This represents improvement of 3.74%. This is significant since this research was carried out in a simulated environment. It could be deduced that for larger payloads of data access, we would have larger improvement values.

Table 4. Response Time (Range) Matrix

	RDBMS(ms)	Document Store(ms)
DBPS	650 – 9800	410 – 5100
SPS	260 – 6860	250 – 5800
PTPS	240 – 6840	250 – 7800
SDB	310 – 6410	320 – 6700
CQRS/ES	350 – 5480	350 – 5900

E. Resource Utilization Matrix for all Patterns

We now discuss the resource utilization performance for the four event-sourcing approaches. The graphical output has been generated Performance Metric Collector which is a plugin available for JMeter and supports application running on the Tomcat Server. We would focus on processor(CPU) usage, memory, disk and Network/IO
As can be seen in Table 5, a combination of CQRS/ES pattern with an RDBMS yielded the highest resource consumption in terms of storage and memory. Conversely a combination of SDB pattern with a document store provided the least overhead in resource consumption.

Table 5. Resource Utilization Matrix

	RDBMS(x1G)	Document Store(x1G)
DBPS	14.5-17	13.5-16.0
SPS	14-16.1	14-15.6
PTPS	13.9-17	13.9-17
SDB	14.2-16	13.9-16
CQRS/ES	13.9-17	14-15.8

It is important to note that from the test cases executed, the difference in the resource utilization is not significant as values remain within the same range. This is shown in Figure 6a and 6b which indicates the general trends for the resource utilization for all the database patterns evaluated in this research.
We can see in Figure 6 the processor usage at approximately 40% with an anomalous dip of at between

00:26 and 00:27. Then we see the memory consumption at a stable value of 60MB throughout the execution of the workload. The disk usage remained at 0 value with a few intermittent spikes.
We can also see from Figure 6(a) to Figure 6(b) shows that the other three approaches exhibit approximately similar resource consumption performance with a relatively high overhead on CPU, disk and network. The summary of the resource utilization values is given in Table 5.
The test cases executed here focuses on CPU, memory and swap. For brevity, only the memory utilization values are presented since other metrics shows similar trend. The memory utilization metric is defined as the heap memory used the host container during the test runs. Table 5 shows a summary of the performance metrics collected for all the test cases.

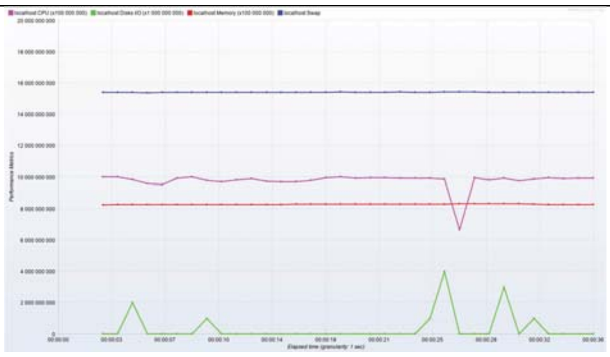

Figure 6a: Resource Utilization Output 1 (DS)

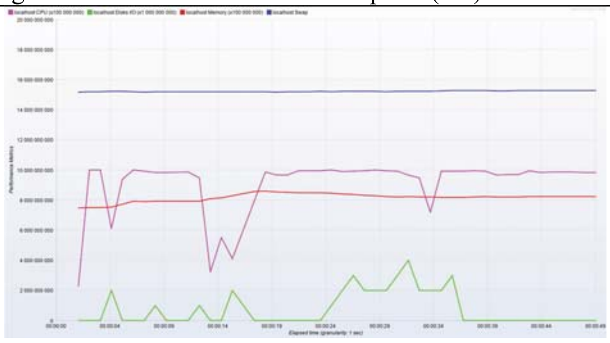

Figure 6b: Resource Utilization Output 2(RDBMS)_

From the outputs above, we could be deduced that for a resource utilization would not serve as a critical performance criteria. However, further tests would be carried out based on enterprise production-level applications as this may provide additional insight.

VI. CONCLUSION AND FUTURE WORK

In this research the Database patterns for microservices have been comprehensively examined the various database patterns for microservice. Moreover, an evaluation of Per-Service pattern and the Shared-Database patterns as demonstrated in this research highlights one of the core challenges of microservices: that is, how to we achieve a loose coupling between the service data while maintaining a very efficient data access performance. While future could find a way to mitigate this challenge, it still represents a significant factor to consider in design of data-driven microservices.

- We can summarize our finding in the following four items as it applies to MSA:
- Document Store bases microservices exhibits better performance than RDBMS
- The Shared Database pattern exhibited better performance than the per-service approach
- The CQRS approach exhibits better performance relative to SDB
- Each pattern shows insignificant impact on system resources relative to other patterns

From the test cases examined in this research, it could be deduced that the data persistence pattern of best performance which is the Document store Shared Database pattern does not align with the principle of modularity and loose-coupling. However, this research shows that the CQRS with ES pattern if standardized could actually mitigate the challenge mentioned. This is the subject on an ongoing research.

However, the decrease in performance created by the used of DBPS pattern could be address by adopting approach like client-side caching which reduces volume of requests to the database. Research on the methods to achieve this is currently been carried out.

This research has further proven that for some data-related operations such as simple CRUD operations the DBPS pattern give a better performance while for others, the SDB pattern gives a better performance. So another subject for further research would examine a possible hybrid pattern that combines both the SDB and the DBPS pattern in a given information system.

VII. REFERENCES

[1] 1. *A Simplified Database Pattern for the Microservice Architecture.* Messina, Anthonio and Rizzo, Riccardo. Palermo : The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications, 2016. 10.13140/RG.2.1.3529.3681.

[2]. *Continuous Software Engineering – A Microservices Architecture Perspective.* O'Connor, Rory V., Elger, Peter and Clarke, Paul M. 11, s.l. : Journal of Software: Evolution and Process, 2017, Vol. 29. 1866.

[3]. *On Micro-Services Architecture.* Namiot, Dmitry and Sneps-Sneppe, Manfred. 9, s.l. : International Journal of Open Information Technologies, 2041, Vol. 2. 2307-8162 .

[4]. Newman, Sam. *Building Microservices - Designing Fine-Grained System.* s.l. : O'Reilly Media, 2015.

[5]. Wolff, Eberhard. *Microservices: Flexible Software Architecture.* s.l. : Addison-Wesley Professional, 2016. 9780134650449.

[6]. *OCEANTEA: EXPLORING OCEAN-DERIVED CLIMATE DATA USING MICROSERVICES.* Johanson, Arne N., et al., et al. Colorado : Sixth International Workshop on Climatics, 2016.

[7]. *Open Issues in Scheduling Microservices in the Cloud.* Fazio, Maria, et al., et al. 5, s.l. : IEEE Cloud Computing, 2016, Vol. 3. 2325-6095.

[8]. *Optimization Strategies for Integration Patern Compositions.* Ritter, Daniel, May, Norman and Forsberg, Fredrik Nordvall. New York : 12th ACM International Conference on Distributed and Event-based Systems (DEBS), 2018.

[9]. *Data-Driven Workflows for Microservices: Genericity in Jolie.* Safina, Larisa, et al., et al. Switzerland : 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA), 2016. 1550-445X.

[10]. *Transaction management in the R* distributed database management system.* Mohan, C., Lindsay, B. and Obermack, R. New York : ACM Transactions on Database Systems (TODS), 1986.

[11]. *Architecture of an interoperable IoT platform based on microservices.* Vresk, Tomislav and Cavrak, Igor. Opatija : 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016. 978-953-233-086-1.

[12]. *Implementation Patterns for Microservices Architectures.* Brown, Kyle and Woolf, Bobby. Illinois : Proceedings of the 23rd Conference on Pattern Languages of Programs, 2016.

[13]. *Using an Ontology Network for Data Integration: A Case in the Public Security Domain.* Santos, Lucas A., Miranda, Gabriel M. and Campos, Silas. s.l. : ONTOBRAS 2018, 2018.

[14]. *Catalog of Formalized Application Integration Patterns.* Ritter, Daniel, Rinderle-Ma, Stefanie and Sinha, Aman. s.l. : Cornell University, arXiv:1807.03197v2, 2018, Vol. 2.

[15]. *Mathematical Model for Simulation of an Integration Solution for Application in the Academic Context of Unijui.* Karisig, Adriana, et al., et al. Unijui : s.n., 2017.

[16]. Linthicum, David. *Enterprise Application Integration.* s.l. : Addison-Wesley Professional, 2000.

[17]. *A Domain-Specific Languate to Design Enterprise Application Integration Solution.* Frantz, Rafael. 02, Unijui : Internation Journal of Corporative Information Systems, 2011, Vol. 20.

[18]. *Modelling EAI (Enterprise Application Integration) published in 12th International Conference of the Chilean Computer Science Society.* Losavio, F., Ortega, D. and Perez, M. Chile : IEEE, 2002. 1522-4902.