Database Connection Pool in Microservice Architecture

Nur Ayuni Nor Sobri, Mohamad Aqib Haqmi Abas, Ahmad Ihsan Mohd Yassin, Megat Syahirul Amin Megat Ali*, Nooritawati Md Tahir and Azlee Zabidi

Abstract—**The increase and growing number of users in the internet gives a higher requirement to backend application systems nowadays to be designed to handle thousands of users traffic concurrently. Microservice architecture is also in a rising trend which they allow for each service to scale horizontally by their throughput and load helps to scale the system efficiently without waste of resources like in the traditional monolithic application system. Database connection pool helps for backend systems to access databases efficiently. The present issue is determining the optimal number of database connections to use in a microservice based backend system. This paper aims to find the most suitable amount of database connections in a microservice setting, where multiple instances of the service are used for scalability and high availability purposes of the system. The experiment was conducted by varying the number of database connections from one to five to ten in both single instance and three instance services, where the service being examined is the backend system's roles and permissions service. The results of this experiment indicate that five database connections provide the best performance latency result in a microservice architecture with three service instances. With 2000 requests per second, the maximum latency was 53ms, while the 99th percentile latency was 19ms. The study contributes by determining the optimal size of a database connection pool for use in a microservice architecture with several instances of the service are operating concurrently.**

Index Terms—**Backend application, database connection pool, high availability, microservice, scalability.**

I. Introduction

IN the past few years in the cloud services domain, we have seen a trend of companies moving from monolithic architecture applications to microservices architecture. The idea of breaking a complex monolithic application that serves the whole functionality in a single application to multiple loosely-coupled and single-purpose started with big tech companies like Apple, Google, Netflix [1–3] due to numerous advantages that it brings compared to the traditional monolithic architecture.

Scalability and flexibility are some of the most important advantages of microservice architecture [4–6]. The traditional approach on handling scalability is to increase the number of instances or the size of the whole monolithic application. Although increasing the number of instances of the application running can help to achieve high availability and fault tolerance, the default way to increase scalability is by increasing the size of the application as it is less complex. However, in the context of monolithic architecture, this is very much inefficient because in most cases, there are only a few particular domains of services that are expected to be used by a large number of users and require high throughput.

In microservice architecture, each of the services are loosely-coupled, serving a single-purpose and independent from other services [2, 4]. Hence, this allows for the ability to deploy and scale each service independently and using different policies from the other services [3, 4].

Most current backend application systems require an interaction between the application and database to store all users data. Most legacy backend systems use a direct method to invoke a call to the database where first the application will create a database connection in the program, execute the SQL query to the database, lastly, close the database connection [7, 8]. However, as the application gets bigger and more complex, this way of making connections to the database is not efficient as it will greatly increase the system overhead to create and close the connection frequently [8].

Newer backend application system uses a database connection pool, where the application will help to create and maintain the connections so that it can be reused in the future as required. The general idea is, connections will be in either two states, whether it is being used or idle [9]. Each time there is a need to make a database request, the application will check if there is any idle connection for it to use, else it will create a new connection for as long as it has not reached the maximum amount of connection threshold. Postgres databases by default have 100 'max_connections' limit and if this limit is being hit under heavy load, the backend application will return an error to end users [9, 10].

This manuscript is submitted on 16th December 2021 and accepted on 23th February 2022. The work is funded by the Ministry of Higher Education, Malaysia through the Long-term Research Grant Scheme (600-RMC/LRGS 5/3 (001/2020)).

Nur Ayuni Nor Sobri and Mohamad Aqib Haqmi Abas are postgraduate students with the School of Electrical Engineering, College of Engineering, Universiti Teknologi MARA, 40450 Shah Alam, Malaysia.

Ahmad Ihsan Mohd Yassin and Megat Syahirul Amin Megat Ali are with the Microwave Research Institute, Universiti Teknologi MARA, 40450 Shah Alam, Malaysia. Nooritawati Md Tahir is with the Institute for Big Data Analytics and Artificial Intelligence, Universiti Teknologi MARA, 40450 Shah Alam, Malaysia (Email: megatsyahirul@uitm.edu.my)

Azlee Zabidi is with the Faculty of Computing, College of Computing & Applied Sciences, Universiti Malaysia Pahang, 26600 Pekan, Malaysia.

*Corresponding author
Email address: megatsyahirul@uitm.edu.my

Some backend applications give the developer the flexibility to choose their own configurations for managing the pool. This is often important as each application will have their own different requirements. In general, a medium sized monolithic application will usually opt with the default amount of maximum database connections, which is 100 database connections if using Postgres database. Having too high amount of maximum connections can also cause problems as it can overwhelm the database and application system, requiring larger amount of memory (RAM) for maintaining the connections and high overhead in terms of CPU cycle and RAM for setting up and closing the connection.

We have found relevant articles related to our work in using database connection pool. In [11], the authors use a database connection pool when developing their Student Information System (SIS), a three-tier web application that allows registrars to perform tasks that involve system setup, admission, registration, graduation grades processing and report. The SIS system was developed using Java and the authors set up a JDBC connection pool to solve the possible issue of scalability of the system. A study of database connection pool done in [7], shows comparison between traditional connection pool with tomcat, hibernate and the new proposed connection pool. The result shown from the study shows how the differences of methods used in managing the connection pool directly affects the performance of the system.

In both [12, 13], the authors study the security aspect of database connection pool in three-tier web systems. In [12], the authors use a formal model of three-tier web system and few security problems faced in the web system were found from the model. Few methods on solving the security issue were introduced and proposed such as securing application, terminal user tracing and modifying the previous standard on securing the database connection pool. Database connection pool audit system (DCPAS) is proposed [13] to trace identity of the end user and bind the operations done by the user to the execution of the SQL statements to the database. The proposed DCPAS allows for a better security audit, as the admin will be able to trace the detailed SQL statements if an illegal user makes an SQL injection to the system.

Generally, choosing the configuration for database connection pool, such as maximum amount of connections and maximum idle time of connection requires performance testing to ensure that the most suitable configuration is chosen for the backend application where it would not cause a bottleneck due to having too low amount of connections and not waste the system's resource by having too high amount of connections. Therefore, this study aims to find the most suitable maximum amount of database connections in a microservice setting, where multiple instances of the service are used for scalability and high availability purposes of the system.

II. Proposed Architecture and Methodology

To tackle the issue of scalability and to achieve high availability of our services, we propose running multiple instances of each of our services in production, especially for services that we anticipate will be hit the most during runtime.

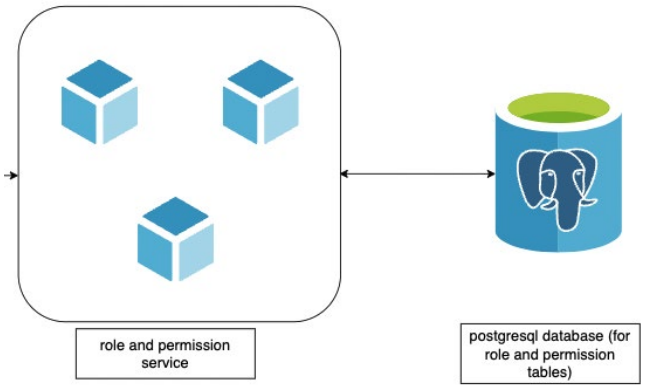
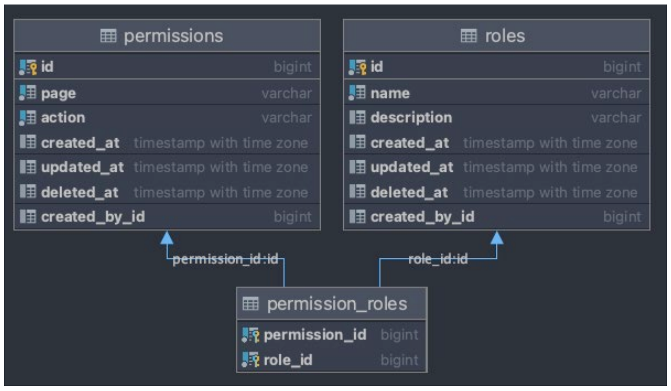

role and permission service
postgresql database (for role and permission tables)

Fig. 1. Multiple instances for role and permission service.

This is to allow load balancing of request load between multiple instances and having backup instances to serve HTTP requests when one of the instances is down.

An example of a single service with the proposed architecture to run in the production server, where there are a total of three instances that are running for the role and permission service is shown in Fig. 1. This is only one small service out of multiple other services that we will run in the production server. The service handles only the roles and permissions information for the system.

Any request that requires the roles and permissions logic from the API gateway will be delegated to this service. As shown, each instance of the service will connect to the Postgres database that has the roles table and permissions table. However, the microservice architecture is flexible and does not set any hard requirements for database setup. In the production system we have the option to set up the database on the same server, set up the database on a different server, or opt with managed database services which most cloud providers are offering. However, accessing a database on a different server or managed service will have an increased network latency due to the request calls needing to be made to an external server instead of accessing a database in a different port on the same server.



permissions	
id	bigint
page	varchar
action	varchar
created_at	timestamp with time zone
updated_at	timestamp with time zone
deleted_at	timestamp with time zone
created_by_id	bigint

roles	
id	bigint
name	varchar
description	varchar
created_at	timestamp with time zone
updated_at	timestamp with time zone
deleted_at	timestamp with time zone
created_by_id	bigint

permission_roles	
permission_id	bigint
role_id	bigint

Fig. 2. Roles and permission table.

Fig. 2 illustrates the roles and permissions table with its intermediary many-to-many table. We are using a role based

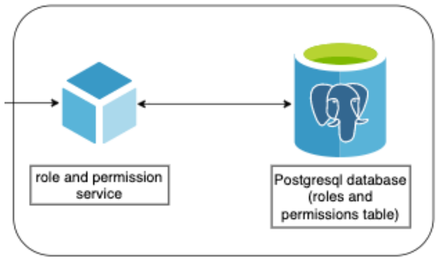

role and permission service
Postgresql database (roles and permissions table)

Fig. 3. Single instance for role and permission service.

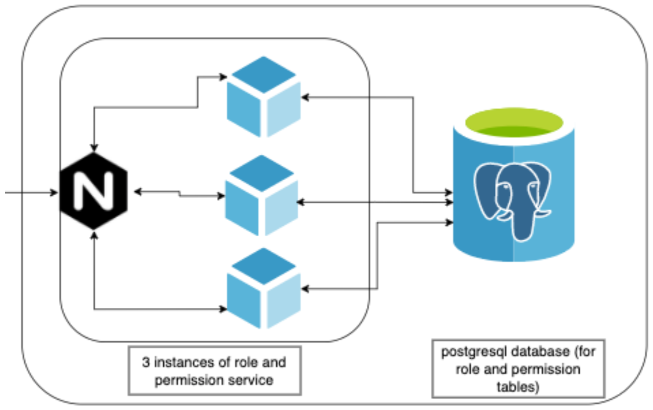

3 instances of role and permission service
postgresql database (for role and permission tables)

Fig. 4. Three instances for role and permission service with Nginx as load balancer.

access control (RBAC) authorization model for our system. In RBAC, users have access to an object, page and module based on their respective assigned role in the system [14]. Roles are commonly assigned based on job function and permissions are defined based on job authority and responsibility of the job.

To find the most suitable maximum amount of database connections we run the performance testing on this service with two different scenarios; first, with a single instance as shown in Fig. 3 and second, with three instances and an Nginx load balancer as shown in Fig. 4, where the load balancer will route the client requests traffic to the three application instances.

The load testing is done with Arm64v8 CPU architecture. The limitation of the platform applies to this project. We also limit the go runtime (for each instance) to use a single CPU (with GOMAXPROCS = 1) and 128 MB of memory (ulimit), however we found neither limiting the CPU and RAM gives any effect in our experiment as none of the tests would even hit the limit. However, the situation will be different in production servers when we deploy the services where we have a more limited amount of CPU cycles and RAM configuration for our machine. The benchmark performance testing will be done using Vegeta load testing tool which is written in Go. In this test, we are using the default setting of Postgres database as will be in production without tuning any configuration. We also did not change for any optimization being done by Postgres for similar SQL request calls either by its shared buffer cache or operating system cache method. The only manipulated variable for this experiment is the maximum number of connections and the number of instances (for the two different scenario test), everything else will be similar throughout the test.

We tested on four different amounts of connections for the database connection pool which are one, five, and ten. The load tester makes 500, 1,000 and 2,000 requests per second (rps) to the service. The load test will be done for 5 seconds for each test. Only one API endpoint will be tested for this experiment, which is the "/roles" endpoint that will give all the roles in the database table, including its permissions relation. The reason that role/permission service are chosen for this experiment is due to this service being one of the most used in the system. Multiple endpoints in the system require authorization checks on whether a specific user has the necessary role and permission needed to access the endpoint.

III. Results and Discussion

Table I shows the result of maximum latency for different number of requests made to different number of connections in a single instance service The max latency for 500 requests per second (rps) made for one, five and ten connections declines as the number of connections increase. For a single connection the latency is at 439 ms, then drops to 65 ms when having five connections and lastly, 24 ms for ten connections. For 1,000 and 2,000 HTTP rps, we can see that having a single database connection becomes a bottleneck to the service as it requires 9,387 ms and 19,433 ms, respectively. Note that this is without tuning any shared buffer cache or operating system level cache for Postgres database default setting, which shows the latency struggle of having a single connection made to the database. Meanwhile, for five connections, the service starts to bottleneck when having 2,000 rps where it recorded 1,658 ms latency. For 1,000 rps the service is still able to tolerate the throughput at 156 ms latency. For ten connections, the latency increases as the number of requests increase to 1,000 and 2,000 at 120 ms and 499 ms, respectively, but it is still bearable compared to having one and five connections.

TABLE I
Maximum Latency (in ms) for Different Number of HTTP Requests Per Second Made to Different Number of Database Connections with Single Instance Service

Number of Connections	Requests Per Second		
	500	1,000	2,000
1	439 ms	9,387 ms	19,433 ms
5	65 ms	156 ms	1,658 ms
10	24 ms	120 ms	499 ms

Table II shows the result of 99th percentile latency for different number of requests made to different number of connections in a single instance service. In some benchmark situations, this number is often used as a realistic measure of latency where 99 percent of end users will receive this latency, while maximum latency can show if there has been a sudden hiccup to a system (that might happen for a single request). The latency shows the same pattern as in the maximum latency result, where the latency decreases as the number of

TABLE II
99TH PERCENTILE LATENCY (IN MS) FOR DIFFERENT NUMBER OF HTTP REQUESTS PER SECOND MADE TO DIFFERENT NUMBER OF DATABASE CONNECTIONS WITH SINGLE INSTANCE SERVICE

Number of Connections	Requests Per Second		
	500	1,000	2,000
1	204 ms	7,442 ms	17,749 ms
5	19 ms	56 ms	427 ms
10	5 ms	35 ms	205 ms

TABLE IV
99TH PERCENTILE LATENCY (IN MS) FOR DIFFERENT NUMBER OF HTTP REQUESTS PER SECOND MADE TO DIFFERENT NUMBER OF DATABASE CONNECTIONS WITH THREE INSTANCES SERVICE

Number of Connections	Requests Per Second		
	500	1,000	2,000
1	7 ms	15 ms	55 ms
5	6 ms	13 ms	19 ms
10	7 ms	18 ms	56 ms

connections increases and single connection shows a bottleneck in performance in both 1,000 and 2,000 rps tests. For 500 rps, single connection gives 204 ms latency, followed by five connections at 19 ms and lastly, ten connections at 5 ms. For 1,000 rps, single connection still shows a bottleneck result at 7,442 ms, followed by 56 ms for five connections and 33 ms for ten connections. For 2,000 rps, we can see that five connections start to show the bottleneck in performance as well at 427 ms, but this is far lower than 17,749 ms which is recorded by single connection. Ten connections shows a good performance at 205 ms.

Table III shows the result of maximum latency for different number of requests made to different numbers of connections in a three instance service. We can see that even with single connection, the service does not suffer the same performance impact as when having only a single instance of service. This shows that having multiple instances helps to balance the throughput load. For 500 rps, single connection gives the best latency at 33 ms, followed by ten connections at 35 ms and lastly, five connections at 36 ms. For 1,000 and 2,000 rps, five connections shows far better performance latency compared to single and ten connections. In 1,000 rps result, five connections only recorded 30 ms, better than its performance at 500 rps, followed by single connection at 53 ms, and ten connections at 59 ms. For 2,000 rps, five connections recorded a low 53 ms, followed by single connection at 165 ms, and lastly ten connections at 227 ms.

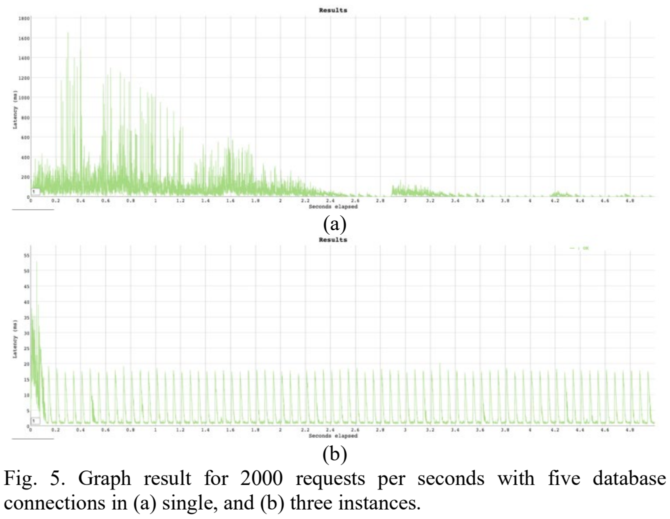
TABLE III
MAXIMUM LATENCY (IN MS) FOR DIFFERENT NUMBER OF HTTP REQUESTS PER SECOND MADE TO DIFFERENT NUMBER OF DATABASE CONNECTIONS WITH THREE INSTANCES SERVICE

Number of Connections	Requests Per Second		
	500	1,000	2,000
1	33 ms	53 ms	165 ms
5	36 ms	30 ms	53 ms
10	35 ms	59 ms	227 ms

Table IV illustrates the result of 99th percentile latency for different number of requests made to different numbers of connections in a three instance service. As seen, five connections shows the best recorded performance for all 500, 1000 and 2000 rps. In 500 rps, five connections records the lowest latency with 6 ms, followed by 7 ms by ten connections and lastly single connection with 7 ms. For 1000 rps, five connections gives 13 ms latency, followed by 15 ms for single connection and 18 ms for ten connections. Lastly, for 2000 rps, five connections only gives 19 ms compared to a single connection with 55 ms and ten connections with 56 ms.

Both Table III and Table IV, which analyse three service instances, produce an unexpected result in which the latency for ten database connection pools is greater than the latency for five database connection pools, which contradicts the result obtained when examining a single service instance in Table I and Table II. One possible explanation for why this occurred during the experiment is that the connection pools were already sufficient for the database query, but the additional time was due to the latency associated with creating new connection pools rather than reusing the current.

Based on all results shown in this section, we can see that low number of database connections will start to become a bottleneck when being hit with a larger load especially with single instance service, however, the performance gets better when multiple instances are involved as load balancing the requests throughput helps to distribute the load instead of hitting only single instance to serve the requests. Having a larger amount of connections is not guaranteed to have a better performance in terms of latency as we can see from the result in the experiment ran with multiple instances, the diminishing return effect for this could be caused by multiple factors such as the algorithm used to assign connection pool to request and how the performance from the database side when handling numerous concurrent connections.



(b)
Fig. 5. Graph result for 2000 requests per seconds with five database connections in (a) single, and (b) three instances.

In Fig. 5, another noticeable difference we see between serving load with single instance and multiple instances is we notice there is a constant spike for every few milliseconds recorded, which could be because of how the load balancer works when distributing the load between instances. However, even with the spike in latency, the overall result of distributing load with multiple services is far better compared to serving all the requests with only a single instance.

IV. Conclusion

We have presented the load testing done to our service to obtain the suitable number of database connections for our database connection pool (DCP). We tested for a single instance of our role and permission service as it is one of the most used services in our system, mostly due to authorization middleware checks for our users to access endpoints. From the result of our experiment and our proposed architecture for production environment, we choose five connections configuration as it gives the best performance for multiple instances service setup as shown in Table III and Table IV result.

As the microservice design for cloud computing gains traction in comparison to the traditional monolithic architecture. The study makes a contribution by establishing the appropriate size of a database connection pool for usage in a microservice architecture with several concurrent instances of the service.

Acknowledgment

The work is funded by the Ministry of Higher Education, Malaysia through the Long-term Research Grant Scheme (600-RMC/LRGS 5/3 (001/2020)).

References

[1] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, "Supporting microservice evolution," in *2017 IEEE Int. Conf. Softw. Maint. Evol.*, Shanghai, China, 2017.

[2] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for clound & edge systems," in *Proc. 2019 Archit. Support Program. Lang. Oper. Syst.*, Providence, RI, 2019.

[3] V. Singh and S. K. Peddoju, " Container-based microservice architecture for cloud applications," in *2017 Int. Conf. Comput. Commun. Autom.*, Greater Noida, India, 2017.

[4] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How to make your application scale," in *Perspectives of System Informatics*. *PSI 2017*. *Lecture Notes in Computer Science*, A. Petrenko and A. Voronkov, Eds, Moscow, Russia: Springer, 2017, pp. 95–104.

[5] W. Hasselbring and G. Steinacker, "Microservice architecttures for scalability, agility and reliability in e-commerce," in *2017 IEEE Int. Conf. Softw. Archit. Workshop*, Gothenburg, Sweden, 2017.

[6] I. Asrowardi, S. D. Putra, and E. Subyantoro, "Designing microservice architectures for scalability and reliability in e-commerce," *J. Phys.: Conf. Ser.*, vol. 1450, no. 1, 2020.

[7] X. D. Xu, B. Li, Q. M. Lu, X. Y. Yan, and J. L. Li, "A study of database connection pool," *Appl. Mech. Mater.*, vol. 556-562, pp. 5267–5270, 2014.

[8] F. Liu, "A method of design and optimization of database connection pool," in *2012 4th Int. Conf. Intell. Hum.-Mach. Syst. Cybern.*, Nanchang, China, 2012.

[9] A. Edwards, *Let's Go Further*, pp. 116–122.

[10] A. Trzop, Estimate database connections pool size for Rails application, Apr. 2021. Accessed on: Nov. 1, 2021. [Online]. Available: https://docs.knapsackpro.com/2021/estimate-database-connections-pool-size-for-rails-application

[11] F. Al-Hawari, A. Alufeishat, M. Alshawabkeh, H. Barham, and M. Habahbeh, "The software engineering of a three-tier web-based student information system (MyGJU)," *Comput. Appl. Eng. Educ.*, vol. 25, no. 2, pp. 242–263, 2017.

[12] X. D. Yu, M. Y. Zhang, M. Q. Zhu, K. H. Xu, and Q. C. Xiang, "Security problem modeling of database connection pool," *Appl. Mech. Mater.*, vol. 543-547, pp. 3276–3279, 2014.

[13] X. D. Yu, M. Y. Zhang, M. Q. Zhu, K. H. Xu, and Q. C. Xiang, "Research on the Security Audit of Database Connection Pool," *Appl. Mech. Mater.*, vol. 543-547, pp. 3286–3289, 2014.

[14] N. Meghanathan, "Review of access control models for cloud computing," in *Comput. Sci. Inf. Technol.*, Delhi, India, 2013, pp. 77–85.