Database Optimization Techniques with Logic Execution Optimization on Microservices Architecture

Teknik Optimasi Database dengan *Logic Execution Optimization* pada Arsitektur *Microservices*

Isnen Hadi Al Ghozali *[1], Mohammad Shiddiq Antarressa[2], Samidi[3]
[1,2,3] Magister Ilmu Komputer, Fakultas Teknologi Informasi, Universitas Budi Luhur;
Jalan Ciledug Raya, Petukangan Utara, Jakarta Selatan, DKI Jakarta
e-mail: ***[1] 2111601163@student.budiluhur.ac.id** , [2]2111601197@student.budiluhur.ac.id ,
[3]samidi@ budiluhur.ac.id

Abstrak

Microservices architecture, arsitektur kerangka kerja terdistribusi yang memungkinkan perubahan pada satu modul tanpa mengganggu modul lainnya. Penerapan arsitektur ini memiliki tantangan tersendiri. API get-list-attachment yang berjalan pada arsitektur ini membutuhkan waktu rata-rata 12,5 detik untuk menyajikan data. Hal ini perlu diperhatikan karena proses bisnis memerlukan waktu akses yang lebih singkat untuk mendukung pengambilan keputusan. Tujuan penelitian adalah untuk mengefisiensikan waktu respon query untuk aplikasi akuntansi. Untuk mencapai hal tersebut, penelitian ini menggunakan teknik optimasi database dengan logic execution optimization microservices architecture. Penelitian memperoleh sumber informasi dari Modul Accounting Harmony Accounting yang memiliki API (get-list-attachment) dengan sumber data dari Service Accounting (581253 record) dan Service Users (2182 record). Berdasarkan pengujian yang dilakukan, beberapa layanan perlu ditambahkan dengan API untuk meningkatkan microservices architecture untuk menerima bulk parameters yang menghasilkan daftar objek sehingga penyajian data lebih optimal. Setelah melakukan serangkaian perekayasaan pada microservices architecture dan indexing application, kinerja query response time meningkat sebesar 49,22% untuk modul Service Accounting.

Kata kunci— *Optimisation Techniques, Indexing, Microservices Architecture*

Abstract

Microservices architecture, a distributed framework architecture that allows changes to one module without interfering with other modules. The implementation of this architecture has its own challenges. The get-list-attachment API running on this architecture takes an average of 12.5 seconds to serve data. This needs to be considered because business processes require shorter access times to support decision making. The research objective is to obtain query response time efficiency for accounting applications. To achieve this, the research uses database optimization techniques with logic execution optimization microservices architecture. This study obtained the source of information from the Accounting Harmony Accounting Module, which has an API (get-list-attachment) with data sourced from Service Accounting (581253 records) and Service Users (2182 records). Based on a series of tests carried out, several services need to be added with APIs to improve the microservices architecture to accept bulk parameters that generate a list of objects so that data presentation is more optimal. After doing a series of engineering on microservices architecture and indexing application, query response time performance increased by 49.22% for Service Accounting module.

Keywords— Optimisation Techniques, Indexing, Microservices Architecture

1. INTRODUCTION

In the last decade, digital transformation has changed work patterns and business processes in various industrial sectors. It is characterized by an increasing need for data storage and management. Modern business organizations demand flexibility in accessing data without the limitations of space and time. One of the implications is the development of accounting applications with the concept of a Software as a Service (SaaS) platform. The platform that will be used massively requires excellent and fast performance support.

Harmony Accounting is a company that has built a product with the concept of a SaaS platform using the microservices architecture, a distributed framework architecture that allows changes to one module not to interfere with other modules. This architecture has its own challenges in executing logic code and queries to run optimally. For example, in the accounting module there is an Application Programming Interface (API) get-list-attachment, to be able to present paginated journal attachment data. The data presented by the API comes from two combined services, namely Service Accounting and Service User. In Service Accounting, which houses the get-list-attachment API, it executes two SQL queries, merging the two tables (journal_attachment and journal). First, execution to get the count of the total data. Second, execution to get a list of data whose amount is in accordance with the requested pagination limit. After that, the list of data obtained will be mapped (combined) with the user data obtained in the API (get-user-byid) on the Service User. In the requested Service User, execute SQL query to get single user data from user table. The current condition, when using parameters that are specifically for testing with 1000 items paginated data returns, the get-list-attachment API takes an average of 12.5 seconds to present the data. Under these conditions, efficient use of queries, logic execution (in calling other services and mapping data) and the availability of filter parameters on each API need to be considered.

The most common and well-known conception for storing data is through the relational model [1]. Relational databases store and organize data in linked tables based on related data. The concept of a relational database is the basis of a DBMS, which is a program used to create and maintain databases. DBMS simplifies the process of defining, manipulating, and sharing databases with multiple users and applications [2]. According to Elmasrti and Navathe, query optimization is an activity carried out by the query optimizer in the DBMS to select the best available strategy for executing queries [3]. Structured Query Language (SQL) extracts relevant data from a collection of databases. In addition, query optimization can use the Index method,

Microservice is the process of implementing a software-oriented architecture by dividing a complete application into interconnected services and each service will serve a specific business need [4]. Theoretically, Fowler [5] states that with a microservice architecture, an application can be easily scaled vertically or horizontally, developer speed and productivity can increase drastically. This microservice architecture follows the Monolithic architecture, where all request handling logic runs in one process, the application is divided into classes, functions, and namespaces using the basic features of the programming language used [4].

In the context of query optimization, there are several methods that can be applied, one of which is using the index method as implemented in [1], [2], [6], [7], [3], and [8]. Research [2] concludes that implementing indexes on spatio-temporal data increases query execution time, while taking into account the overall effect of query optimization. Although testing on relatively small data (less than 100,000 records) did not show a significant difference in execution time. Using 2,248,590 records, study [8] states that the index method is superior in terms of query response time compared to the other three tables in the study. While [3] concluded that table indexing and query optimization improve database performance, in this case it reduces response time. It is interesting to note, experiment [1] proposes intelligent indexing, an algorithm that can

perform automatic operations (defrag, recreation, or modification) that can improve the overall performance of the system. Several other studies have proposed alternative query optimization besides indexing such as MapReduce framework and a semantic-based clustering method [9], query batching optimization [10], Nesterov Accelerated Gradient [11], genetic algorithm [12], and Neo [13].

For microservice architecture, [5] has proven Service-oriented Process for Reengineering and DevOps (SPReaD), as a result of the process of re-engineering the old system architecture into a microservice architecture. After the implementation of these microservices, there is an increase in performance and scalability. Meanwhile, from a theoretical point of view, study [4] concludes that microservice architecture offers extraordinary agility and efficiency. This microservice architecture further increases productivity to DevOps, which is a set of activities for integrating application creation. However, implementing a microservices architecture on a system is not an easy matter. Research [14] details six challenges in engineering microservices architecture in webshops. This challenge is also felt by web-based application developers. This microservice architecture has its own complexity in terms of initial setup when the query optimizer attempts to perform query optimization. This is because when optimizing, you must consider synchronization between services. One of the things that sometimes becomes an obstacle between services is using a different relational database management system (Firebird). These matters will be studied in more depth in this study.

Previous studies mostly did not mention in which industry the methodology was applied. Whereas at the implementation level, the public needs to know which research is in accordance with the characteristics of the industry. This is what makes us specifically examine the field of the online accounting service provider industry. In previous studies, which focused more on query optimization, they did not consider the overall application architecture, especially the microservice architecture. The research related to microservice architecture that we found led to a survey paper. Research that examines the application of microservice architecture in industry is still limited. Therefore, this study will examine database optimization techniques with logic execution optimization microservices architecture for accounting applications.

2. METHODOLOGY

This study uses an experimental method using the Harmony Accounting dataset module, a Software as a Service (SaaS) product that focuses on accounting, invoicing, and financial systems. The Harmony Accounting module is built using a microservice architecture, which is divided into many microservices including Inventory, Accounting, User and Payroll. The trial was carried out on the Accounting Module, which has an API (get-list-attachment) to be able to present paginated journal attachment data. The data presented by the API comes from two combined services, namely Service Accounting and Service User. The database in Service Accounting has two tables, namely journal_attachment with 312038 records and journal with 269215 records. In Service User, there is a single user data from the user table with 2182 records.

We conducted experiments by replicating the related services in this trial on a server with Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz specifications, 8GB RAM, and 8GB/s SSD. The operating system used is Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-188-generic x86_64) and the database system uses MySQL 5.7.38-0ubuntu0.18.04.1. The programming language used is Javascript which is running using the Node JS engine v12.17.0 with the Express JS framework. In this trial, the server used is dedicated specifically for this research and the optimizer only replicates some services and code related to the test object but does not change the functional, logic code and flow so that the test can run like the original system.

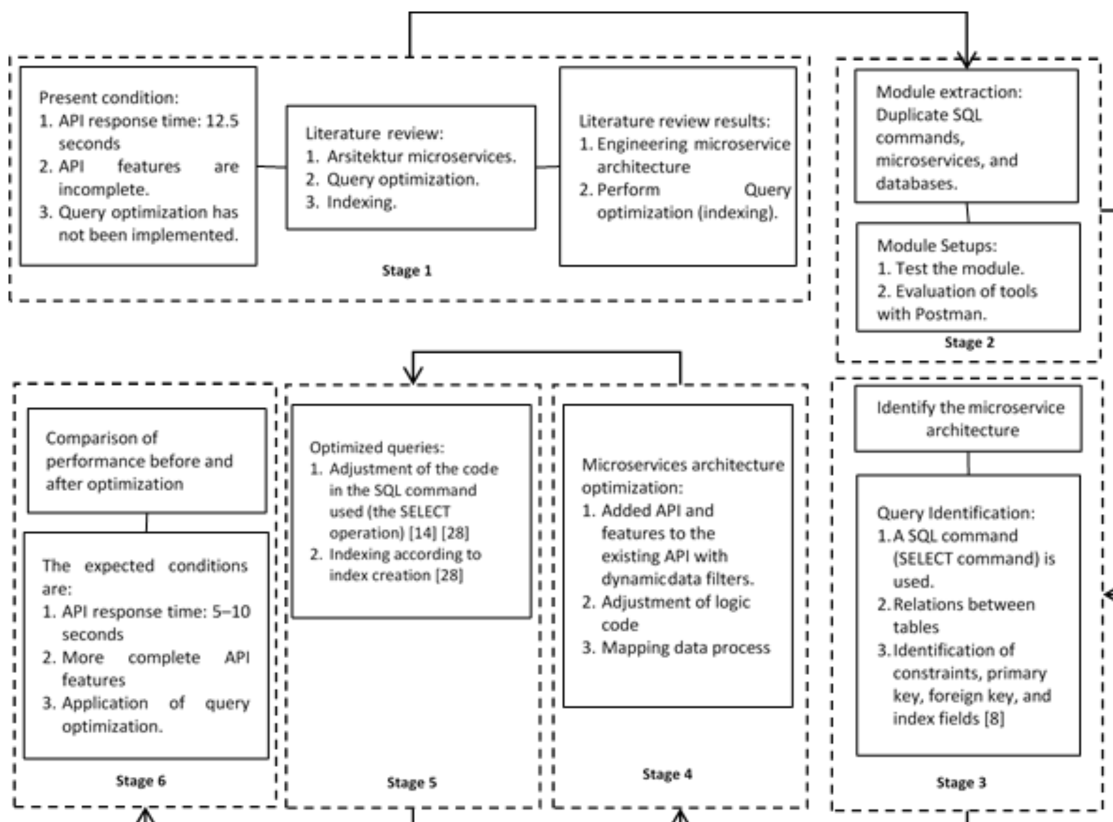

Figure 1. Research Framework

This study conducted a series of tests to compare the performance before and after tuning. The test was carried out on the API in which there was an ineffective SQL statement and the research also carried out the process of engineering microservices architecture by rewriting logic code on the mapping of data found that it was necessary to make requests to other services as much as the total data you wanted to map. The study carried out six stages to obtain conclusions, namely (1) identification of problems that occurred in the research sample and literature review related to microservices architecture, query optimization, and indexing strategies; (2) Perform module extraction and setup environment; (3) Identification of microservice and query architectures that perform less than optimally; (4) Optimization of microservices architecture (5) Query optimization by reengineering the SQL command (SELECT statement) and implementing Table Indexing; (6) Performance comparison before and after optimization of microservices architecture and query optimization.

3. RESULTS AND DISCUSSION

Stage 1: The problem we can identify consists of three major parts. First, the get-list-attachment API response time is up to 12.5 seconds. this condition by using parameters that are specifically for testing with the return of data that has been paginated as much as 1000 items. Second, the currently available API does not yet have a feature to be able to set the fields/attributes needed by the sender. Third, query optimization has not been implemented thoroughly in Service Accounting. To get a solution to this problem, first a literature study related to microservices architecture, query optimization, and indexing was conducted. After conducting an in-depth literature study, there are several proposed steps that can be taken to overcome these problems. First, engineering microservices architecture can improve performance and scalability [5]. There are several methodsquery optimization methods that can be applied, one of which uses the index method [1] [2] [6] [7] [3] [8].

Stage 2: Extracting the module in the sample is done by copying the SQL command from the server using Visual Studio Code. After all the SQL commands are copied, then a test is carried out to run the module in the test environment. the next step is to evaluate the tools needed so that the module runs normally in the test environment with the help of the Postman application.

Stage 3: The application of the microservices architecture in Harmony Accounting allows services with varying levels of complexity to be divided into several microservices according to business needs [4]. When a service requires data from a module, it will be handled by the microservice module. To obtain data between these services ideally using the API Gateway. For communication between services using HTTP requests, and the response data returned is in JSON format. The implementation is to support DevOps, a set of methods to minimize time to changes in systems and changes in production [5].

The discussion of this research is limited to the microservices architecture in Service Accounting, including the transaction datasets associated with it. Although the internet connection between the sender and the recipient of the service affects the speed of the service's response time, this is not included in the discussion in this study. When setting up the module in the environment, we found some services that do not have an API to return multiple object data/bulk request data. The currently available APIs can only accept and return a single object. The implication is, if there is a data list that requires detailed data from the service, it is necessary to make HTTP requests as much as the number of the data list. The currently available API does not yet have a feature to be able to set the fields/attributes required by the sender.

Stage 4: For services that do not have an API, it is necessary to create an API that can accept bulk parameters that generate a list of objects. Therefore, the service needs to add filter parameters to the existing API and the new API, which will affect the operation of the SELECT statement on the query. Thus, the presentation of data is more optimal, it can reduce the size of the presentation of data related to communication between services. The addition of the API will have implications for the need to change the logic code when requesting a service. For existing API services that generate/require single object data in other services, they can send data attribute filters so that the data returned is data that is really needed.

In the Accounting service architecture, if a user or other service requests data, the attachment by company_id module activates the mappingAttachJournal function so that the database will return 100 items attachments. The mappingAttachJournal function sends 100 items attachments for mapping. The data mapping process asks for user data 100 times. Service User activates getUserById to get userdata from the database. The result of the process returns single user data 100 times, the data will be assigned to each item in the data mapping process. Then, the data mapping process returns 100 attachment items that have been mapped for each attachment item with detailed user information. Finally, the Accounting service returns 100 attachment items with complete data to the user or another service requesting data.

Service which produces a list of data and each item requires a data object in another service, to get it by requesting as much as the number of list data. This will be changed by first collecting the IDs of each item in the data list and adding a data attribute filter to be sent with a single request, after which it can be mapped again. The application of this change is carried out with the uniq user IDs parameter obtained from all item attachments, as well as using the required data filter feature. This applies to Service User when enabling getUserById. As a result, Service User returns multiple user data with total data less than the number of items, but will not exceed the total items. The data will be assigned to each item in the data mapping process.

Table 1. Modification of Code Service Accounting

	Existing Logic Code	Rengineering Logic Code
SQL Statement	SELECT count(*) AS total FROM (SELECT j.journ_id AS journ_id,	SELECT count(*) AS total FROM (SELECT j.journ_id AS journ_id FROM journal j INNER JOIN journal_attachment att ON j.journ_id = att.journ_id

	Existing Logic Code	Rengineering Logic Code
	j.journ_name AS journ_name, j.journ_date AS journ_date, att.jattch_id AS jattch_id, att.jattch_name AS jattch_name, att.jattch_url AS jattch_url, att.jattch_src AS jattch_src, att.created_date AS created_date, att.created_by AS created_by FROM journal j INNER JOIN journal_attachment att ON j.journ_id = att.journ_id WHERE ((j.company_id = 1 AND att.deleted_date is NULL)) GROUP BY att.jattch_id ORDER BY att.jattch_name DESC, att.jattch_id DESC) AS journal	WHERE ((j.company_id = 1 AND att.deleted_date is NULL))) AS journal
Service User	const newData = [] for(let x = 0, len = data.length; x < len;x++) { const rowData = data[x] newData.push(new Promise((resolve,reject)=>{ request({ method: 'get', url: `${baseUrlUser}/getUser/?use rId=` + rowData.created_by, json: true }, function (err, res, body) { if (res.statusCode == 200) { resolve({ id: rowData.journ_id, file_name: rowData.jattch_name, file_url: rowData.jattch_url, file_src: rowData.jattch_src, module: 'journal', description: rowData.journ_name,	let userIds = {} for(let x = 0, len = data.length; x < len;x++) { const rowData = data[x] userIds[rowData.created_by] = rowData.created_by; } const usersData = await new Promise((resolve,reject)=>{ request({ method: 'post', url: `${baseUrlUser}/getUserByIds`, body:{"userIds":Object.values(use rIds),"select":["user_id","user_fullna me"]}, json: true }, function (err, res, body) { if (res.statusCode == 200) { resolve(body) } else { reject() }})})

	Existing Logic Code	Rengineering Logic Code
	user_name: body.user_fullname, upload_date: rowData.created_date, transaction_date: rowData.journ_date, }) } else { reject() }})}))}	
SQL Statement pada API	SELECT `user_id`, `user_fullname`, `user_email`, `user_phone`, `user_status` FROM `user` AS `user` WHERE `user`.`user_id` = ${request.userId(String)}	SELECT ${request.filter(ex: `user_id`, `user_fullname`)} FROM `user` WHERE `user`.`user_id` IN (${request.userIds(ex:1, 65, 251, 266, 994)})

As an implication of adding the API, the SQL command for the SELECT operation also needs to be adjusted. Table 1 shows the modification of code Service Accounting. Previously the SELECT operation involved many columns, this was less efficient because not all columns were needed. After optimization, the SELECT operation only involves the required columns. SQL query on get-list-attachment API aims to get the total data as information on pagination. Changes to the SELECT operation are performed in line with the simple SELECT operation [10]. In addition, the GROUP BY and ORDER BY operations are removed because they are not needed in the process. When the GROUP BY and ORDER BY operations occur, there is a significant decrease in data presentation time. With a total of 310,984 data, the query that previously took 4.5 seconds can be optimized to 1.5 seconds. It is also intended to minimize items in the SELECT statement, whose end goal is to get the total value of the data with the COUNT() operation. With this engineering, the SELECT operation is more efficient because it does not have to first read many columns from the database, the operation can focus on only the columns that are needed.

Stage 5: The next engineering process is carried out on the User service, which previously had to request as much as the amount of data to refer to unique data (userIds). The process of mapping data on the get-list-attachment API (service accounting) uses a list of data that has been obtained by executing the query in Table 2. The process needs to be assigned to user data obtained from another service, namely the User service. Prior to optimization, to get user data, it is necessary to access the getUser API available on the User service, but the API can only accept a single userID parameter and can only return single user data. we made changes to the command const newData = [] to let userIds = {}. Using this command is followed by changing method: 'get' to method:'post'. The change from get to post changes the paradigm of requesting data from an unspecified number of data into a mechanism for sending a number of unique data to the database to request updates to resources. The process of merging item attachment data with user data obtained by the getUser API is carried out one by one in parallel requests so that the number of requests for service users is 1000 times (in this study 1000 times). Based on the selection process from 50 items of data that are combined in user_id, by eliminating data duplication, it produces data with IDs 1, 65, 251, 266, 994. Previously, SQL Query on the Service Account API focused on getting the desired user data, after doing so engineering into multiple user data.

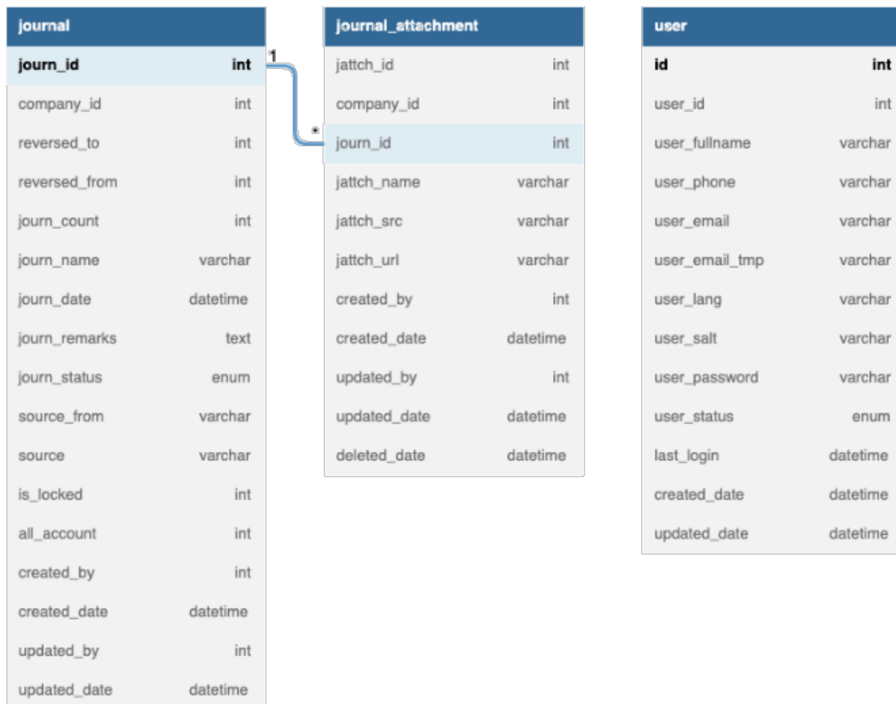

Figure 2. Tabel Database
(journal and journal_attachment pada service “accounting”, user in service “user”)

The resulting index table is shown in Table 2. The index table is applied to three tables, namely the journal table, the journal attachment table, and the user table. Each table is added a primary key for data that is often accessed during transactions. Index creation based on the clustered index pattern. This pattern uses a unique index per table and uses a primary key to organize the data available in the table. This clustered index improves performance for data manipulation such as SELECT, INSERT, UPDATE and DELETE. This pattern is automatic in MySQL.

Table 2. Table Index

Information	Query
SQL query to divide data into pages based on what was asked for in the API request.	SELECT journ_id, journ_name, journ_date, jattch_id, jattch_name, jattch_url, jattch_src, created_date, created_by FROM (SELECT j.journ_id AS journ_id, j.journ_name AS journ_name, j.journ_date AS journ_date, att.jattch_id AS jattch_id, att.jattch_name AS jattch_name, att.jattch_url AS jattch_url, att.jattch_src AS jattch_src, att.created_date AS created_date, att.created_by AS created_by FROM journal j INNER JOIN journal_attachment att ON j.journ_id = att.journ_id WHERE ((j.company_id = 1 AND att.deleted_date is NULL))

Information	Query
	GROUP BY att.jattch_id ORDER BY att.jattch_name DESC, att.jattch_id DESC) AS journal limit 0, 1000
Query indexing table journal	ALTER TABLE `journal` ADD PRIMARY KEY (`journ_id`), ADD KEY `journal_company_id` (`company_id`), ADD KEY `journal_reversed_to` (`reversed_to`), ADD KEY `journ_id` (`journ_id`), ADD KEY `company_id` (`company_id`);
Query indexing table journal attachment	ALTER TABLE `journal_attachment` ADD PRIMARY KEY (`jattch_id`), ADD KEY `journal_attachment_company_id` (`company_id`), ADD KEY `journal_attachment_journ_id` (`journ_id`), ADD KEY `jattch_id` (`jattch_id`), ADD KEY `company_id` (`company_id`), ADD KEY `journ_id` (`journ_id`);
Query indexing table user	ALTER TABLE `user` ADD PRIMARY KEY (`user_id`), ADD KEY `user_id` (`user_id`), ADD KEY `user_fullname` (`user_fullname`);

Stage 6: After the index table is successfully created, testing is carried out on the SQL query response time with the help of the Postman application to generate data as shown in Table 3. Before optimization, Stage 1 requires a response time of up to 2757 ms, after optimization the response time is significantly reduced to 99.35% to just 18 milliseconds (ms). For Stage 2, the optimization process was able to reduce the response time from 4507 ms to 1580 ms, meaning that there was a decrease in response time of 64.94%. For Stage 3, the optimization results were only able to reduce the response time by 5.81%, from 4817 ms to 4537 ms. At Stage 3, the optimization process carried out did not have a significant impact. This is because at Stage 3 there are still GROUP BY and ORDER BY operations. These two operations cannot be deleted because they are related to the application of limits. Overall, the optimization process carried out was able to reduce the response time very satisfactorily, which was 49.22%. If the previous response time required 12,081 ms, after optimization is done, the response time decreases to 6135 ms. This means that the optimization process carried out can significantly reduce the response time.

Table 3. Response Time Before and After Indexing

Stage	Data Row (Record)	Before (ms)	After (ms)	Difference	
				Time (ms)	Percentage (%)
request get-userdata-by id (Service -> user) as many as 10 data attachments without specific select (Stage 1)	2182	2757	18	2739	99.35%
SQL query to return the total number of items pagination attachment_journal and journal, without limit, select statement (Stage 2)	581253	4507	1580	2927	64.94%
SQL query to get attachment_journal and journal pagination items, with limit (Stage 3)	581253	4817	4537	280	5.81%
		12081	6135	5946	49.22%

Based on the identification and testing that we carried out on the microservice architecture in the research sample, the challenges faced to implement the architecture are in line with research [14]. This research has optimized the mapping process to add APIs to several services as a step in engineering microservice architecture to improve performance. The results of this study support research [5], [4], and [16] for the application of microservices architecture in business organizations to obtain flexibility and efficiency. Although with a note, this architecture requires prerequisites such as the use of Docker to support synchronization between services.

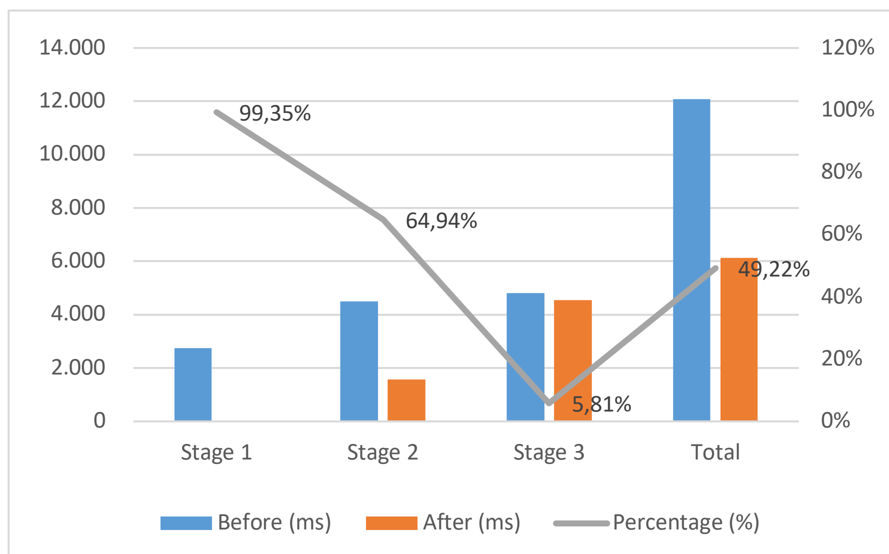

Figure 2. Comparison of Response Time Before and After Indexing

Adding an API to the service has implications for changing the SQL command for the SELECT operation. The results of research testing the changes made have a significant effect on reducing response time, this result is contrary to research [17] which states that simple SELECT has no significant effect on performance. The engineering of SELECT operations supports research [18] and [3]. This research is in line with the thinking of the two studies that the SELECT operation should only involve the required columns so that the process of requesting data is more efficient.

The results of the indexing process carried out in this study support research [1], [19], [6], [7], [8], [20], and [15]. This study performs indexing on journal tables, journal attachment tables, and user tables using the ALTER operation as research [6]. Index creation based on the clustered index pattern. This method is commonly used in related research. Based on the test results, indexing in this study succeeded in increasing the speed of SQL response times [6]. Indexing optimizes SQL in the search or scan process so that it can refer to the target data faster, without having to scan the entire table's rows. The response time after indexing based on the test results in this study was better than the response time in previous studies [21] [22] [23] [8] [24]. The difference in the results of this test can be caused by differences in the complexity of the relational table used in the related research sample. In addition, changes to the microservices architecture have become an inseparable factor in the database optimization process. MySQL testing in previous studies generally focused on the database engine only so that the results obtained were still not optimal. As a database engine, MySQL has many shortcomings compared to the latest database engines, but excels in terms of stability and ease of application in a variety of application architectures. The test results are still not optimal when compared to the NoSQL-based database engine [25] [22] [23] [24]. This is something that needs to be studied further considering that implementing NoSQL in the research sample requires radical changes and has the potential to not be implemented in the near future [26]. This is because the sample uses a microservice architecture so synchronization between services can be a separate issue if you want to implement NoSQL.

4. CONCLUSION

Based on a series of tests carried out, several services need to be added with APIs to improve the microservices architecture to accept bulk parameters that generate a list of objects so that data presentation is more optimal. Meanwhile, query optimization using the indexing method has been proven to improve the performance of data response times for the Accounting service module. The results of this study support the results of previous studies which state that the indexing method can improve database performance [1][18][6][7][8].

This study recommends adding APIs to several services so that the performance of data presentation is more optimal. We also recommend engineering code to be applied to all modules. Implementing indexing on the service can also be considered to improve SQL response times.

For further research, we suggest to examine the effect of internet connection between service sender and receiver on microservices architecture. We also suggest further research to continue comparative studies related to the use of various database engines, especially the implementation of NoSQL. This study can add insight regarding the implementation of microservices architecture with NoSQL, especially in the Industry 4.0 era [27].

ACKNOWLEDGMENT

This research is supported by PT Harmoni Solusi Bisnis (Harmony). We would like to thank fellow Developers of PT Harmoni Solusi Bisnis (Harmony) who have provided insight and expertise that greatly assisted this research.

REFERENCES

[1] Arteta Albert, N. Gómez Blas, and L. F. de Mingo López, “Intelligent Indexing—Boosting Performance in Database Applications by Recognizing Index Patterns,” *Electronics*, vol. 9, no. 9, p. 1348, Aug. 2020, doi: 10.3390/electronics9091348.

[2] E. Inersjö, *Comparing database optimisation techniques in PostgreSQL : Indexes, query writing and the query optimiser*. 2021. Accessed: Jun. 27, 2022. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-306703

[3] Samidi, D. Iskandar, M. Fachruroji, W. Adi Septyo Wibowo, and A. Khaerani A, “Database Tuning in Hospital Applications Using Table Indexing and Query Optimization,” *J. Pendidik. Tambusai*, vol. 6, no. 1, pp. 1960–1967, 2022.

[4] C. V. Dave, “Microservices Software Architecture: A Review,” *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 9, no. 11, pp. 1494–1496, Nov. 2021, doi: 10.22214/ijraset.2021.39036.

[5] C. E. da Silva, Y. de L. Justino, and E. Adachi, “SPReaD: service-oriented process for reengineering and DevOps: Developing microservices for a Brazilian state department of taxation,” *Serv. Oriented Comput. Appl.*, vol. 16, no. 1, pp. 1–16, Mar. 2022, doi: 10.1007/s11761-021-00329-x.

[6] S. Maesaroh, H. Gunawan, A. Lestari, M. SufyanAts Tsaurie, and M. Fauji, “Query Optimization in MySQL Database Using Index,” *Nternational J. Cyber IT Serv.*, vol. 2, no. 2, pp. 104–110, 2022.

[7] M. V. Praveena and A. A. Chikkamannur, “IndexingStrategies for Performance Optimization of Relational Databases,” *Int. Res. J. Eng. AndTechnologyIRJET*, vol. 8, no. 5, pp. 3801–3805, 2021.

[8] S. Samidi, F. Fadly, Y. Virmansyah, R. Y. Suladi, and A. B. Lesmana, “Optimasi Database dengan Metode Index dan Partisi Tabel Database Postgresql pada Aplikasi E-Commerce. Studi pada Aplikasi Tokopintar,” *J. Pendidik. Tambusai*, vol. 6, no. 1, pp. 2094–2102, 2022.

[9] E. Azhir, N. Jafari Navimipour, M. Hosseinzadeh, A. Sharifi, and A. Darwesh, “A technique for parallel query optimization using MapReduce framework and a semantic-based clustering method,” *PeerJ Comput. Sci.*, vol. 7, p. e580, Jun. 2021, doi: 10.7717/peerj-cs.580.

[10] M. Eslami, V. Mahmoodian, I. Dayarian, H. Charkhgard, and Y. Tu, “Query batching optimization in database systems,” *Comput. Oper. Res.*, vol. 121, p. 104983, Sep. 2020, doi: 10.1016/j.cor.2020.104983.

[11] A. Rahmanto, A. Budi, and R. Primananda, “Implementasi Self-Tuning Pada Database Dengan Menggunakan Metode Nesterov Accelerated Gradient,” *J. Pengemb. Teknol. Inf. Dan Ilmu Komput.*, vol. 5, no. 5, pp. 1907–1913, 2021.

[12] K. T. Hidayat, R. Arifudin, and A. Alamsyah, “Genetic Algorithm for Relational Database Optimization in Reducing Query Execution Time,” *Sci. J. Inform.*, vol. 5, no. 1, p. 27, May 2018, doi: 10.15294/sji.v5i1.12720.

[13] R. Marcus *et al.*, “Neo: A Learned Query Optimizer,” 2019, doi: 10.48550/ARXIV.1904.03711.

[14] W. K. G. Assunção, J. Krüger, and W. D. F. Mendonça, “Variability management meets microservices: six challenges of re-engineering microservice-based webshops,” in *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, Montreal Quebec Canada, Oct. 2020, pp. 1–6. doi: 10.1145/3382025.3414942.

[15] Q. Xie, W. Yang, and L. Yao, “A Database Optimization Strategy for Massive Data Based Information System,” in *Proceedings of the 2019 2nd International Conference on Mathematics, Modeling and Simulation Technologies and Applications (MMSTA 2019)*, Xiamen, China, 2019. doi: 10.2991/mmsta-19.2019.47.

[16] R. V. O’Connor, P. Elger, and P. M. Clarke, “Continuous software engineering-A microservices architecture perspective,” *J. Softw. Evol. Process*, vol. 29, no. 11, p. e1866, Nov. 2017, doi: 10.1002/smr.1866.

[17] C. A. Győrödi, D. V. Dumşe-Burescu, R. Ş. Győrödi, D. R. Zmaranda, L. Bandici, and D. E. Popescu, “Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases,” *Appl. Sci.*, vol. 11, no. 15, p. 6794, Jan. 2021, doi: 10.3390/app11156794.

[18] S. J. Kamatkar, A. Kamble, A. Viloria, L. Hernández-Fernandez, and E. G. Cali, “Database Performance Tuning and Query Optimization,” in *Data Mining and Big Data*, vol. 10943, Y. Tan, Y. Shi, and Q. Tang, Eds. Cham: Springer International Publishing, 2018, pp. 3–11. doi: 10.1007/978-3-319-93803-5_1.

[19] G. Feng, "The Design and Optimization of Database," *J. Phys. Conf. Ser.*, vol. 1087, p. 032006, Sep. 2018, doi: 10.1088/1742-6596/1087/3/032006.

[20] J. Kossmann, T. Papenbrock, and F. Naumann, "Data dependencies for query optimization: a survey," *VLDB J.*, vol. 31, no. 1, pp. 1–22, Jan. 2022, doi: 10.1007/s00778-021-00676-3.

[21] S.-V. KHOLOD, "Performance comparison for differenttypes of databases," *Fac. Appl. Sci. Ukr. Cathol. Univ.*, pp. 1–25, 2021.

[22] M. S. Kumar and P. .J, "Comparison of NoSQL Database and Traditional Database-An emphatic analysis," *JOIV Int. J. Inform. Vis.*, vol. 2, no. 2, p. 51, Mar. 2018, doi: 10.30630/joiv.2.2.58.

[23] Y. Y. Putra, O. Purwaningrum, and R. H. Winata, "PERBANDINGAN PERFORMA RESPON WAKTU KUERI MySQL, PostgreSQL, dan MongoDB," *J. Sist. Inf. Dan Bisnis Cerdas*, vol. 15, no. 1, pp. 39–48, Mar. 2022, doi: 10.33005/sibc.v15i1.2749.

[24] R. Wodyk and M. Skublewska-Paszkowska, "Performance comparison of relational databases SQL Server, MySQL and PostgreSQL using a web application and the Laravel framework," *J. Comput. Sci. Inst.*, vol. 17, pp. 358–364, Dec. 2020, doi: 10.35784/jcsi.2279.

[25] D. Ilin and E. Nikulchev, "Performance Analysis of Software with a Variant NoSQL Data Schemes," in *2020 13th International Conference "Management of large-scale system development" (MLSD)*, Moscow, Russia, Sep. 2020, pp. 1–5. doi: 10.1109/MLSD49919.2020.9247656.

[26] P. Martins, M. Abbasi, and F. Sá, "A Study over NoSQL Performance," in *New Knowledge in Information Systems and Technologies*, vol. 930, Á. Rocha, H. Adeli, L. P. Reis, and S. Costanzo, Eds. Cham: Springer International Publishing, 2019, pp. 603–611. doi: 10.1007/978-3-030-16181-1_57.

[27] V. F. de Oliveira, M. A. de O. Pessoa, F. Junqueira, and P. E. Miyagi, "SQL and NoSQL Databases in the Context of Industry 4.0," *Machines*, vol. 10, no. 1, p. 20, Dec. 2021, doi: 10.3390/machines10010020.