Using Server-side Processing Techniques to Optimize Data Presentation Responsiveness

Gat
Information System
STMIK Pontianak
Pontianak, Indonesia
gat@stmikpontianak.ac.id

Muh. Jamil
Computer Science
Universitas Widya Gama Mahakam
Samarinda, Indonesia
jamil@uwgm.ac.id

Irawan Wingdes
Information System
STMIK Pontianak
Pontianak, Indonesia
irawan_wingdes@stmikpontianak.ac.id

Tri Widayanti
Information System
STMIK Pontianak
Pontianak, Indonesia
tri.widayanti@stmikpontianak.ac.id

Tony Wijaya
Information System
STMIK Pontianak
Pontianak, Indonesia
tony_wijaya@stmikpontianak.ac.id

Kusrini
Magister of Informatics Engineering
Universitas AMIKOM Yogyakarta
Yogyakarta, Indonesia
kusrini@amikom.ac.id

Abstract*— One of the biggest challenges facing web developers in the rapidly changing digital world is how to convey massive volumes of data in an effective and adaptable manner. The use of server-side processing techniques to enhance the responsiveness of web applications' data presentation is covered in this article. Faster data serving is made possible by this strategy, which transfers the majority of the client's processing work to the server—especially in contexts with massive and complicated data. The goal of this project is to use the Gatling tool to simulate loads, evaluate response time, throughput, and latency in various big data scenarios, and analyze the performance of web applications with server-side processing approaches. The performance of web applications that make use of server-side processing approaches is measured and compared in this study using an experimental approach with load simulation. It is clear from the testing results of the program's versions 1 and 2 that the application with server-side processing techniques implemented for data display is considerably more optimal than the application without such techniques. In comparison to version 2, which took 0.86 seconds on average to reply to queries, version 1's server took an average of 0.31 seconds. According to the web page performance, version 1 loaded resources faster than version 2, taking 1.52 seconds as opposed to 105 seconds. Version 1 of the web page takes 0.082 seconds to render on the GPU, while Version 2 of the program takes 0.86 seconds.

Keywords—SPP, responsiveness, application, gatling, DevTools

I. INTRODUCTION

The amount of data that must be processed and presented in many commercial applications can grow significantly, particularly in the context of big data. Performance can suffer when the complete dataset is loaded on the client side, particularly with web apps that are used on low-spec devices. [1]. Customers dealing with the company's digital platforms and employees using internal systems both need rapid access to pertinent and easily comprehensible information [2]. Delays in data display can result in a bad user experience in digital services, including online or mobile applications, which could eventually hurt the business. The ability to make decisions quickly might mean the difference between success and failure in a cutthroat business environment [3]. Because the business environment is dynamic, businesses must be able to deliver data in a timely and relevant manner. Businesses want a system that can handle massive volumes of data while displaying it in real-time with the least amount of response time [4]. This is crucial to enable prompt, informed decision-making based on precise facts.

As responsive data display has an impact on user experience, performance, and operational efficiency, it is crucial to optimize it [5]. Users access data from a range of devices with varied screen sizes and specs, especially in the modern era of multi-device usage. Data should be presented as best as possible on computers, tablets, and mobile devices when it is responsive [6]. For users to efficiently access data in a variety of scenarios, the system needs to be able to adjust to diverse devices, including desktop, tablet, and mobile phones. Through the optimization of data presentation responsiveness, firms can enhance user retention, deliver better services, and foster future growth and innovation. Server-side processing is one method that has attracted a lot of interest since it uses the server's power to handle and process data more effectively than client-side processing methods [7].

It's critical to enhance data presentation responsiveness using server-side processing approaches, particularly when working with huge datasets or applications that demand extensive data processing [8]. Large datasets can be managed using server-side processing because it only sends the client the necessary subset of the data [9]. Servers with more powerful resources can conduct complex computations, which leads to more efficient big data processing on the server side. Depending on what the user requires, the server can optimize database queries [10]. As a result, the server produces responses more quickly than when all data processing is done on the client end. Reducing network burden by transmitting only the data that is required improves application responsiveness. In order to maintain safe data, optimal performance and provide a high-quality user experience, server-side processing is essential to the creation of modern web applications [11].

This study focuses on the ways that server-side data processing can improve the responsiveness, efficiency, and speed at which users receive data. Using server-side processing techniques should hopefully lessen the amount of work that needs to be done on the client side. This can enhance response time optimization, the speed at which web pages render, and the general responsiveness of applications. The findings of this study should have a major impact on the creation of quicker and more effective apps to meet modern business requirements.

II. RESEARCH METHOD

In order to test the performance of web applications by simulating user traffic, this study employs an experimental approach based on load simulation with Gatling, an open-source tool [12]. Frequently, gatling is employed to detect possible problems with performance and enhance apps to

manage heavy traffic [13]. In order to produce performance metrics such as reaction time, throughput, latency, and error rate, Gatling simulates user loads using various scenarios. A comparison is made between client-side and server-side processing methods based on test results. To evaluate the benefits and drawbacks of each strategy, performance in the two scenarios is examined. Data from multiple dimensions, including response speed, scalability, and client and server resource utilization efficiency, are compared. In the context of load testing using Gatling, two versions of the application are used: version 1 uses server-side processing, and version 2 does not. The goal of this scenario's testing is to gather performance metrics including throughput, reaction time, and resource consumption. Web page speed is measured using PageSpeed Insights, which measures performance indicators such as First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT), and Cumulative Layout Shift (CLS). The web application to be measured has a data volume of 36,612 records.

III. Result And Discussion

PHP is used to show the tested data in a browser's user interface, which is accessible through a MySQL database. Server-side processing approaches are highly successful in optimizing responsiveness when presenting enormous volumes of data. This is because the server-side technique uses pagination, which restricts the amount of data that can be sent to the client at once. As a result, the client's rendering time and network burden are decreased. The limit function, which generates paging and limits the quantity of data, is seen below.

```
static function limit ( $request, $columns )
{
    $limit = '';
    if ( isset($request['start']) && $request['length'] != -1 ) {
        $limit = "LIMIT ".intval($request['start']).", ".intval($request['length']);
    }
    return $limit;
}
```

This SQL statement's LIMIT clause limits the number of rows returned from the query result. It is created using the static limit function. A parameter request is an array with two values: length, which represents the number of rows to be taken from the dataset beginning at the start value, and start, which indicates the first row of the dataset to be obtained. The limit function works well when used in conjunction with server-side processing, which is frequently used with interactive tables.

PHP server-side processing for DataTables is handled by the function basic in the SSP class (ssp.class.php). Based on the request given by DataTables, this function is in charge of getting data from the database and delivering it in a format that works with DataTables. Function Simple constructs a SQL query using the pagination, sorting, and filtering options given in the $request. After that, the produced query is run using the connection specified in $conn against the database. The information generated by the query will be obtained and arranged in the DataTables-required manner. In response to the Ajax call, the compiled data will be encoded into JSON format and returned to DataTables. This is a simple function line.

```
static function simple ($request, $conn, $table, $primaryKey, $columns){}
```

In order to establish how data will be obtained from the database depending on parameters supplied from DataTables, a function simple must first construct a SQL query string from a request. Creating LIMIT, ORDER BY, and WHERE clauses to manage pagination, sorting, and search filtering are steps in this procedure. An SQL query string is as follows:

```
$limit = self::limit($request, $columns);
$order = self::order($request, $columns);
$where = self::filter($request, $columns, $bindings);
```

The SQL LIMIT clause is generated by the static function limit() using the pagination settings obtained from DataTables. An SQL ORDER BY clause is generated by the static function order() using the sorting parameters from DataTables. An SQL WHERE clause is generated by the static method filter() using the user's search query in DataTables.

When constructing a SQL query string from DataTables requests, LIMIT, ORDER BY, and WHERE clauses are created in response to user inputs for sorting, searching, and pagination. The appropriate data is then retrieved from the database by combining these clauses to form a complete SQL query. These procedures guarantee that the user's demands and preferences are met by the data that is obtained from the database. As a result, information that satisfies the user's request is processed further and forwarded back to DataTables for presentation. This is the primary query to retrieve the data.

```
$data = self::sql_exec( $db, $bindings,
"SELECT SQL_CALC_FOUND_ROWS `".implode("`,
`", self::pluck($columns, 'db'))."`
FROM `$table`
              $where
              $order
              $limit"
        );
```

To obtain data that matches the previously created WHERE, ORDER, and LIMIT clauses, the main query is run. The total number of rows found is counted using SQL_CALC_FOUND_ROWS, which ignores the LIMIT.

Function simple then computes the length of the data set following filtering, following the execution of the primary query to get the data. This is significant because, independent of pagination or any constraints that may be imposed on the primary query, DataTables must know the entire number of rows that fit the specified filter. The Data set length after filtering phase is essential for providing DataTables with information about the amount of data that matches the user-applied search and filter requirements. After the primary query is run, this is accomplished by utilizing the MySQL FOUND_ROWS() method, which provides the number of matching rows without taking pagination into account. Next, using this outcome, the value of recordsFiltered is updated and sent back to DataTables as part of the JSON response. Data set length after filtering code is as follows..

```
$resFilterLength = self::sql_exec( $db,
   "SELECT FOUND_ROWS()"
);
```

Determining the overall length of the data set prior to applying any filters is the last step in the data processing of the simple function. It is necessary to provide DataTables with this

information in order for DataTables to accurately display the total amount of data that is available in the table. DataTables receives this information back in the form of a JSON response that shows all of the records that are available in the table, not just the filtered data. The code for "total data set length" is as follows.

```
$resTotalLength = self::sql_exec( $db,
    "SELECT COUNT(`{$primaryKey}`)
    FROM   `$table`"
);
$recordsTotal = $resTotalLength[0][0];
```

Setting up the output to be returned to DataTables as an array is the last stage in the simple function. All the information needed by DataTables to show data accurately is included in this output, including information on the total number of data entries, the number of data entries following filtering, and the data that corresponds to the page the user requested. The last phase, known as constructing the output, entails gathering and organizing all the data needed by DataTables into an array that will be returned as a response. The draw number, the total number of records (recordsTotal), the number of filtered records (recordsFiltered), and the actual data to be presented (data) are the important components in this array. In order for DataTables to properly handle and display data in response to user requests, all of these information is necessary. The output code is as follows:

```
return array(
  "draw"  => isset ($request['draw']) ?
             intval($request['draw']):0,
  "recordsTotal"  => intval( $recordsTotal
),
  "recordsFiltered" => intval(
$recordsFiltered),
   "data" => self::data_output( $columns,
$data )
);
```

When working with huge datasets, method simple facilitates the connection between DataTables and a database via server-side processing. It is possible to make modifications or improvements in accordance with the particular requirements of the application by comprehending how this strategy operates. This JavaScript code initializes an HTML table with server-side processing functionality using jQuery and DataTables.

```
    <script>
    $(document).ready(function() {
       $('#example').DataTable( {
            "processing": true,
            "serverSide": true,
            "ajax": "load-data.php",
        } );
    } );
    </script>
```

Because it loads data from the server dynamically when needed, pagination, searching, and sorting, this JavaScript code allows a table to accommodate immensely large amounts of data. DataTables sends an AJAX call to load-data.php once the table loads. After processing this request, the server retrieves pertinent data from the database and provides it in JSON format. After that, DataTables will manage pagination, sorting, and searching based on the returned results in addition to displaying the data in the HTML table. The server-side processing technique's data output is displayed in Figure 3.1 below.

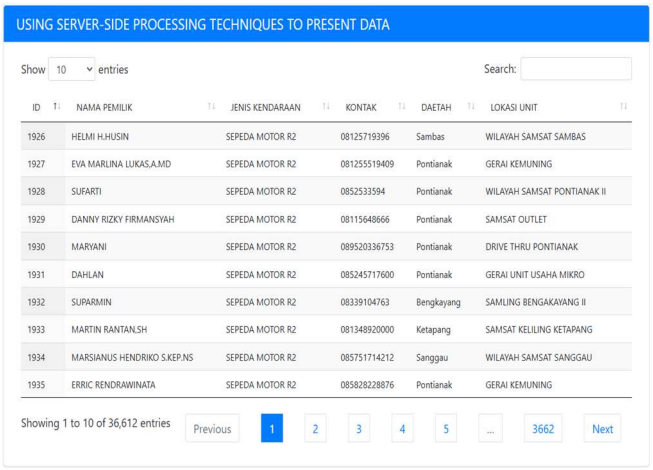


Figure 3.1: Utilizing Server-Side Processing Techniques to Present Data

3.1 Server-Side Processing Workflow

Users communicate with the server by sending requests through the web application. Next, the server verifies that the user has the necessary access rights and that the request is legitimate. In order to obtain the desired data, the server makes use of a database query. Following collection, the data is processed using needs-based techniques like filtering, sorting, and pagination. Following processing, the prepared data is returned to the browser by the server in HTML format. After data is received by the browser, it is rendered or presented using the client-side template.

3.2 Server-Side Processing Pagination Techniques

In server-side processing, pagination algorithms break up large amounts of data into smaller pages so that users don't have to load everything at once. Pagination in server-side processing is carried out on the server. Pagination techniques combined with server-side processing are a potent combo that can handle huge data applications while maintaining excellent user experience and speed.

Evaluating the timeliness of data presentation using server-side processing methods guarantees that consumers receive data in a timely and effective manner. Testing is necessary to determine how server-side processing impacts user experience overall, resource consumption, and data loading speed. Two copies of the program are created for the testing scenario; the first version uses server-side processing techniques, while the second version does not.

The application is hosted on the same server for both versions in order to provide testing settings that are similar. database containing 36,612 records, a significant amount of data, to test the scalability of the program. Version 1 (with Server-Side Processing) uses ajax and DataTable to dynamically fetch data from the server when the serverSide option is set to true. Version 2 (without Server-Side Processing) loads and processes all data initially on the client side using DataTable;

server-side processing is not enabled. Test results for these two versions of the program should indicate that version 1 (Server-Side Processing) performs better and is intended to handle huge data sets more responsively. When handling huge datasets, Version 2 (without Server-Side Processing) is expected to be less responsive and efficient. Gatling is the testing instrument utilized (https://cloud.gatling.io/).

3.3 Server-side processing in version 1 application testing.

The performance of a web application under load can be understood by examining two important metrics seen in Gatling reports: requests per second and responses per second. Figure 3.2 Requests and Responses per Second (version 1) is shown below.

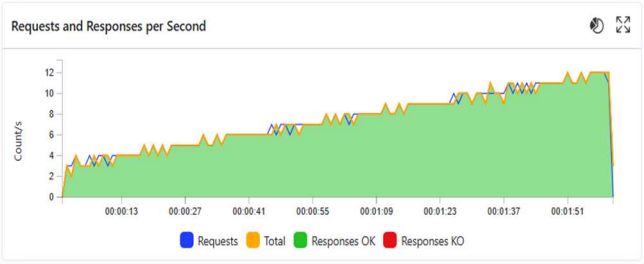


Figure 3.2 Requests and Responses per Second (version 1)

Figure 3.2 above suggests that the number of requests and successful responses is gradually increasing at the beginning of the graph. This shows that requests are starting to come in and be handled by the server gradually. The graph exhibits stability after a brief period of time, perhaps 30 seconds, during which the quantity of requests and responses stays largely unchanged. This suggests that the server can currently manage the load satisfactorily. The graph's predominant green hue suggests that the majority of requests are handled satisfactorily.

Nearly all requests receive a successful response, indicating that the server is able to handle the load during most of the testing, according to server performance analysis (shown by the dominant green areas). The lack of the color red suggests that all requests were answered correctly. The graph's rapid fall at the conclusion usually signifies the end of the testing process and is normal. Figure 3.3 Response Time Percentiles (version 1) is shown below.

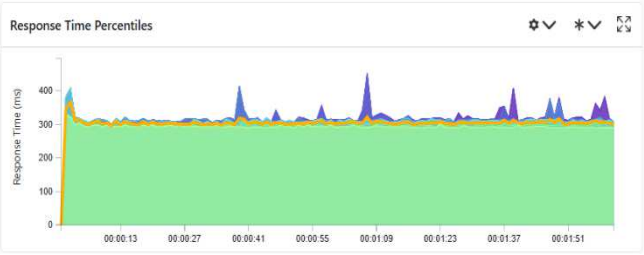


Figure 3.3 Response Time Percentiles (version 1)

According to the interpretation of Figure 3.3 above, there is a significant spike in response time (365 ms) at the beginning of the graph. This happens as a result of the server loading or handling the initial request. The green area dominates the graph for the most part, meaning that at least 50% of requests are handled quickly and reliably in the range of 300 ms. Longer response times are indicated by the hues blue and purple, but only for a tiny percentage of queries (5% and 1%, respectively). Spikes in these regions indicate that a variety of variables, including server load, complex or heavy background operations, or the complexity of the requests itself, may contribute to delayed response times for some requests. A tiny percentage of queries at specific times may have lengthier response times, as seen by some spikes in the graph (especially those in the purple color). This could be a sign of irregularities or an unequal server load.

The bulk of queries have comparatively quick and consistent response times, according to server performance research, notably in the 50th and 75th percentiles. There are certain requests that take longer to respond, especially those in the 95th and 99th percentiles, but these are the exception rather than the rule. Response Time, as used in Gatling performance testing, is the amount of time the system takes to reply to a user request. Figure 3.4 Response Time (version 1) is shown below.

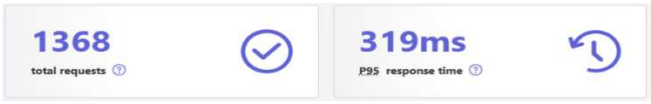


Figure 3.4 Response Time (version 1)

The response time figure of 319 ms shows that, during testing, the server typically responds to requests in 319 ms, or 0.319 seconds. Quicker server response is indicated by a lower number. In general, faster reaction times offer a more favorable customer experience. Response times longer than one second, or 1000 milliseconds, are frequently regarded as the point at which users become aware of delays. A time of 319 ms is less than a second, which is still generally acceptable for web applications because end users typically don't notice answers in this range. Using the browser's built-in tools to analyze and optimize a web application is known as performance testing using DevTools (Developer Tools). Figure 3.5 below displays the web page's performance recorded (version 1).

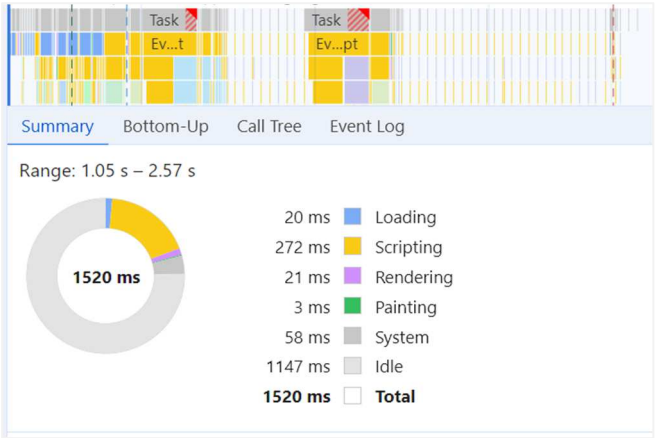


Figure 3.5 Web Page Performance Record (version 1)

The accompanying graphic explains why the monitoring session lasted for a total of 1520 milliseconds, or 1.52 seconds. To get the total, all of the monitored activities—loading, scripting, rendering, painting, system, and idle—are added together. The time it takes to load resources from the network, including HTML, CSS, JavaScript, and pictures, is 20 ms (0.02 seconds). The scripting value, or the amount of time spent running JavaScript scripts, is 272 ms (0.272 seconds). The rendering value, which includes layout and composition, is 21 ms (0.021) in rendering time. The majority of the time is spent in idle, suggesting that there might be delays elsewhere,

like while awaiting certain events or server answers. Since performance optimization is the main objective, scripting and idle time reduction can take precedence.

The GPU (Graphics Processing Unit) displays the amount of RAM that is being used. The amount of memory that the GPU uses to render a page is measured by the GPU Memory. Figure 3.6 GPU Memory Usage (version 1) is shown below.

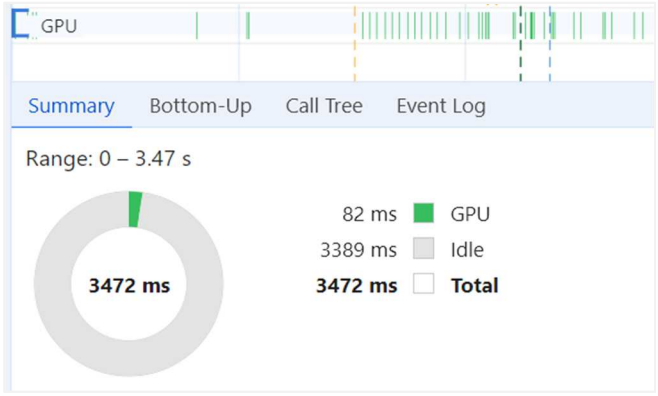
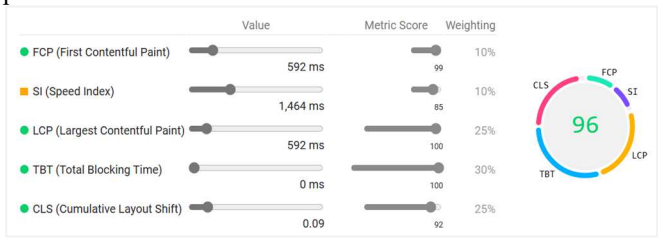


Figure 3.6 GPU Memory Usage (version 1)

It is evident from Figure 3.6 above that the GPU needs 82 ms, or 0.082 seconds, to render the web page. A lower number denotes higher performance. This value represents the speed at which the GPU renders the webpage.

Next, Google PageSpeed Insights will be used to measure performance.



Gambar 3.7 Lighthouse Scoring Calculator (versiion 1)

A web application with an overall performance score of 96, as demonstrated in Figure 3.10 above, is considered to be doing exceptionally well. The value of the FCP (First Contentful Paint) is 592 ms, or around 0.592 seconds. This indicates that the first element on the screen appears 0.592 seconds after the page request. Because it shows how soon viewers can begin seeing material from the web page, this FCP number is very important. The value of the SI (Speed Index) is 1,464 ms, or roughly 1.46 seconds. This indicates that it takes 1.46 seconds on average for the page's viewable content to fill the screen. A webpage that produces visible material more quickly is said to have a lower Speed Index rating and usually offers a better user experience. The LCP value is equivalent to 0.592 seconds, or 592 ms. This indicates that 0.592 seconds after the website loads, the largest content element shows on the page. Better performance is shown by a lower LCP value since the user experiences the website as being faster. TBT is equal to 0 ms, or around 0 seconds. This indicates that the main thread was blocked and unable to react to user input for a total of 0 milliseconds during the page load. A page that has a lower TBT (Total Blocking Time) number loads more quickly and offers a better user experience since it responds better to user inputs. With a CLS (Cumulative Layout change) value of 0.09, this page has a layout change that is sufficiently minor for consumers to not notice it. Version 2 of the application (without server-side processing).

The testing procedure for version 2 of the program is the same as that for version 1, beginning with the Requests per Second and Responses per Second tests. Figure 3.8, which displays the Web page's performance record results (version 2), is shown. below.

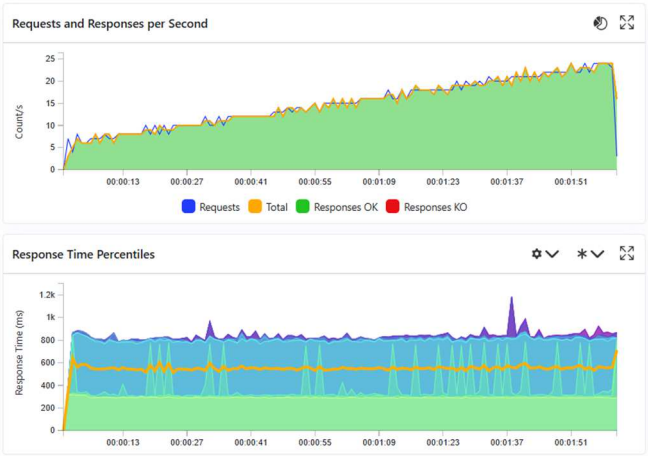


Figure 3.8 Requests and Responses per Second (version 2)

It is evident from picture 3.8 above that the blue and purple regions have a higher spike. This condition suggests that there are some requests that have an excessive load and take longer to complete. Figure 3.9 below displays the web page's performance recorded (version 2).

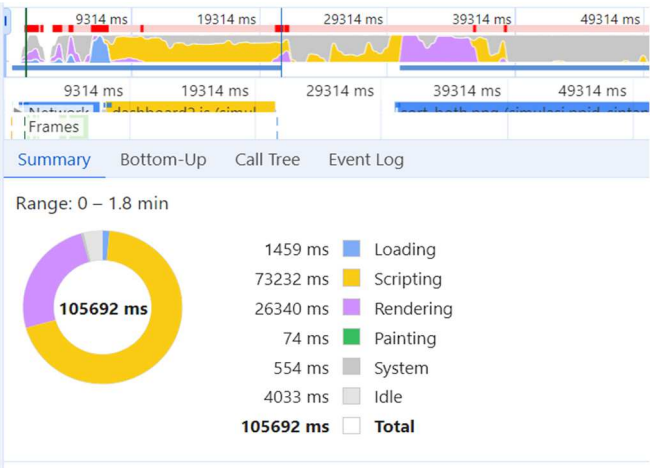


Figure 3.9 Web Page Performance Record (version 2)

It is clear from figure 3.9 above that 105,692 ms, or 105 seconds, was spent in total throughout the monitoring period. Scripting takes up the most time (69.3% of the total). This condition suggests that there is a significant amount of JavaScript usage in the application. The fact that painting and rendering take a long time suggests that the browser must process a lot of visual elements. Since scripting takes up the majority of the time, it should be the main target of optimization. The user experience will be sped up overall if scripting can be streamlined to cut down on processing time. The web performance findings in Figure 3.9 are substantially slower (105 seconds) than the results in Figure 3.5 (1.52 seconds) when compared to the performance tests as displayed in Figure 3.5.

After that, test the GPU (Graphics Processing Unit) to see how long it takes to render the page. Figure 3.10, which displays the GPU Memory Usage (version 2), is provided below.

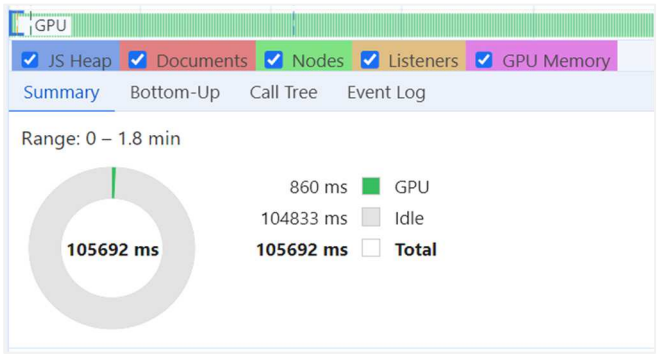


Figure 3.10 GPU Memory Usage (version 2)

It is evident from image 3.10 above that 860 ms (0.86 seconds) of GPU RAM is consumed during the web page rendering process. Version 1 of the application renders web pages faster (0.082 seconds) than version 2 of the application (0.86 seconds) when compared to the GPU value in image 3.6.

It is also required to do Gatling testing on the application's version 2 in order to gauge response time to user requests. Figure 3.11 Response Time (version 2) is shown below.

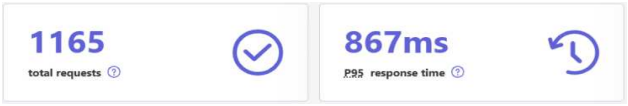


Figure 3.11 Response Time (version 2)

The response time value of 867 ms shows that, throughout the testing, the server responds to requests in an average period of 867 ms (0.86 seconds). This 867 ms number is still fairly decent for a web application because it is less than 1 second. However, the response time of 0.319 seconds is significantly faster than the 0.86 seconds in response to requests when compared to the testing of application version 1 (Figure 3.4).

Server-side processing techniques have been demonstrated to considerably increase the responsiveness of data presentation, especially in applications with big data volumes and high user counts, based on test results obtained with the Gatling tool. For applications that need to process massive amounts of data in real time, query optimization is crucial to reducing response times. This is because it expedites database access. According to test results, server-side processing performs better overall than client-side processing, particularly when managing complicated data and high user demands. Response times in client-side processing typically deteriorate as the volume of data to be processed grows and the processing load is shifted to the user device with limited resources. On the other hand, client-side processing might still be useful if the application needs real-time communication with low network latency or if the amount of data being processed is tiny. Even while server-side processing methods exhibit superior performance, using methods like caching might present additional difficulties, such as staleness when the data is not updated in real-time. Applications that depend on continuously updated data may be impacted by this. Applications that need to browse a whole dataset at once could find that pagination techniques are unsuitable because users may feel limited by the amount of information shown on a single page. The present study's findings are consistent with other research emphasizing the value of server-side processing for applications that handle substantial data loads and heavy user traffic.

IV. Conclusion

In order to evaluate how well the application presents data to users via a browser, version 1 of the program—which uses server-side processing techniques—has been tested. Testing was also done on version 2 of the program, which does not use server-side processing techniques, as a comparison in this measurement. Based on the conducted tests, it is evident that version 1 of the application (0.319 seconds) responds faster than version 2 (0.86 seconds). Version 1 of the application performs better on web pages than version 2, taking 1.52 seconds as opposed to 105 seconds. Version 1's (0.082 seconds) web page rendering procedure also required less GPU RAM than version 2's (0.86 seconds). The web performance metrics evaluation yielded an overall performance score of 96, which is quite favorable for an online application. Consequently, while developing web applications, utilizing server-side processing techniques is the best option for managing massive data quantities and preserving performance while maximizing the responsiveness of data presentation. To determine the ideal ratio between server-side and client-side processing methods, more study may concentrate on evaluating various combinations of the two. It's also worthwhile to investigate alternative server-side processing strategies like load balancing and serverless architectures to see how the newest developments may affect how responsively data is presented.

References

[1] A. D. Praba, M. Safitri & F. Faridi, Implementasi Datatables Server-Side Untuk Mempercepat Load Halaman Pada Aplikasi E-Commerce," JIKA (Jurnal Informatika), vol. 5, pp. 139-144, 2021.

[2] I. Ismail, S. Naja & R. Akbar, "Sistem Informasi Pengelolaan Arsip Digital Pada Kantor Dinas Pertanian Provinsi Aceh Berbasis Web," Jurnal Sistem Komputer (SISKOM), vol. 4, pp. 60-71, 2024.

[3] S. Hendriani, R. Y. Sari & N. Gistituati, "Pengaruh Gaya Kepemimpinan Terhadap Efektivitas Pengambilan Keputusan," Jurnal Niara, vol. 17, pp.171-184, 2024.

[4] M. S. Anwar & N. F. Rozi, "Optimisasi Performa Akses Data dalam Grafana Menggunakan Indeks B-Tree MySQL," In Prosiding Seminar Implementasi Teknologi Informasi dan Komunikasi (Vol. 3, No. 2, pp. 286-292), 2024.

[5] J. A. Putra, A. E. Putri & D. Atmoko, "Mengoptimalkan Pengalaman Pengguna: Meningkatkan Desain Website melalui Riset Sistem Komputer Interaktif," Jurnal Minfo Polgan, vol. 13, pp. 278-288, 2024.

[6] S. Hidayatullah, W. C. Wahyudin, A. Prihandono & S. Ulya, "Perancangan Website Responsif Simas Untuk Penyuluhan Stunting Dan Gizi Anak Pada Masyarakat," Jurnal Ilmu Komputer Dan Matematika, vol. 5, pp. 36-44, 2024.

[7] R. A. Pramono, F. O. Saputra & G. A. Trisnapradika, "Implementasi Index Mysql Dan Server-Side Datatables Untuk Optimalisasi Pemrosesan Data Sistem MBKM Fik Berbasis Codeigniter 4," Jurnal Ilmiah Dinamika Rekayasa, vol. 20, pp. 49-57, 2024.

[8] N. Ilame, "Enhancing Web Application Performance through Database Optimization: A Comprehensive Study," MZ Journal of Artificial Intelligence, vol. 1, pp. 1-13, 2024.

[9] H. Sulastri, A. Rahmatulloh & D. K. Hidayat, "Server-Side Processing Techniques for Optimizing the Speed of Presenting Big Data," Jurnal Pilar Nusa Mandiri, vol. 15, pp. 47-52, 2019.

[10] R. Kurniawan, H. L. Sari & H. Aspriyono, "Web Server Load Balance Design In Internet Network Using Nth Method," Jurnal Media Computer Science, vol. 2, pp. 185-202, 2023.

[11] C. Deming, P. R. Baddam & V. R. Vadiyala, "Unlocking PHP's Potential: An All-Inclusive Approach to Server-Side Scripting," Engineering International, vol. 6, pp. 169-186, 2018.