DINAMIKA REKAYASA
http://dinarek.unsoed.ac.id

Volume 21 Nomor 2 (2025) Hal. 166-174
e-ISSN 2527-6131
http://jurnaldinarek.id

EVALUATION OF REDIS IN-MEMORY BASED CACHE ALGORITHM ON WEB APPLICATION DATA ACCESS PERFORMANCE USING IRCACHE DATASET

THE EFFECT IN-MEMORY BASED CACHE SYSTEM ON WEB APPLICATIONS IN IMPROVING DATA ACCESS PERFORMANCE

Fazelian Alsya Pramudia*[1], Mulki Indana Zulfa[1], Muhammad Syaiful Aliim[1]

*Email: fazelian.pramudia@mhs.unsoed.ac.id

[1]Electrical Engineering, Jenderal Soedirman University, Purwokerto, Indonesia

Abstrak

Dalam era digital, penetrasi internet di Indonesia mencapai 79,5% pada 2024, menjadikan kinerja website kunci untuk pengalaman pengguna optimal. Waktu respon lambat dan beban server tinggi dapat menurunkan kepuasan pengguna. Sistem caching, dengan penyimpanan sementara data, menjadi solusi untuk mempercepat akses dan mengurangi beban server. Berbeda dengan RDBMS yang terbatas karena berbasis harddisk, *In-Memory Database* seperti Redis menawarkan kecepatan lebih tinggi dengan penyimpanan di RAM. Penelitian ini mengevaluasi efektivitas *caching* pada aplikasi web berbasis Laravel melalui simulasi beban menggunakan Apache JMeter dan *dataset* IRCache. Pengujian algoritma *cache Least Recently Used* (LRU), *Least Frequently Used* (LFU), *Random Replacement* (RR), *and First In First Out* (FIFO) pada berbagai skenario beban pengguna menunjukkan pengurangan *response time* rata-rata sebesar 63,78% dan kenaikan *throughput* rata-rata sebesar 32,84% dibandingkan tanpa *caching*, dan algoritma yang lebih unggul pada seluruh *dataset* yaitu RR sebesar 62,06%. Penelitian memberikan kontribusi dalam pengembangan strategi *caching* efisien untuk aplikasi web berskala besar menggunakan Redis.

Kata kunci: aplikasi web, *cache*, *cache replacement*, IMDB, RDBMS, Redis

Abstract

In the digital era, internet penetration in Indonesia reached 79.5% in 2024, making website performance crucial for optimal user experience. Slow response times and high server loads can reduce user satisfaction. Caching systems, which temporarily store data, offer a solution to accelerate access and reduce server load. Unlike RDBMS, which is limited due to its reliance on hard disk storage, In-Memory Databases like Redis offer higher speed by storing data in RAM. This study evaluates the effectiveness of caching in Laravel based web applications through load simulation using Apache JMeter and the IRCache dataset. Testing of cache algorithms Least Recently Used (LRU), Least Frequently Used (LFU), Random Replacement (RR), and First In First Out (FIFO) under various user load scenarios showed an average response time reduction of 63.78% and an average throughput increase of 32.84% compared to non-cached systems. Among all datasets, the most effective algorithm was RR with a performance gain of 62.06%. This research contributes to the development of efficient caching strategies for large-scale web applications using Redis.

Keywords: *web application, cache, cache replacement*, IMDB, RDBMS, Redis

I. INTRODUCTION

In the current digital era, web-based applications have become integral to daily life. The Indonesian Internet Service Providers Association (APJII) reported that in 2024, Indonesia had 221,563,479 internet users, with a penetration rate of 79.5%, marking a 1.4% increase from the previous year [1]. A website is a collection of interconnected web pages stored on the same server to provide information to users [2]. With the growing number of users, website performance is crucial for delivering an optimal user experience. Slow response times and high server loads can reduce user satisfaction, making caching a viable solution to enhance web application performance. Caching enables temporary storage of static and dynamic data to accelerate access and reduce server load [3].

In database systems like MySQL, the query cache feature stores SELECT statement text and results to avoid re-executing identical queries. However, in some conditions, query cache can reduce efficiency if it becomes too large or is frequently invalidated due to write operations [4]. Relational Database Management Systems (RDBMS) remain a primary choice for data storage due to their ACID properties, which ensure transaction integrity, but disk-based storage slows data access as data volume grows [5]. As an alternative, In-Memory Databases (IMDB) like Redis offer faster performance by storing data directly in RAM. IMDBs have been widely adopted by cloud providers such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure for their ability to increase throughput and reduce latency [6]. According to Zulfa et al. (2020), Redis proved more effective in memory usage and had the fastest execution time compared to other databases like Memcached, H2, Cassandra, and MongoDB [5].

Cache management requires optimal replacement strategies to determine which data to evict when the cache is full [7]. The Least Recently Used (LRU) algorithm replaces the least recently accessed data to make room for new data [8], while Least Frequently Used (LFU) retains objects with the highest request frequency, maintaining historical access statistics [9]. The Random Replacement algorithm removes items randomly, though this approach is not always efficient [10]. First In First Out (FIFO) replaces the earliest cached item with the newest, applying a queue-based method for cache management [11]. With the right algorithm, caching can enhance web application efficiency and optimize data access times for users.

Few studies have empirically compared LRU, LFU, FIFO, and Random Replacement (RR) algorithms under varying user load scenarios using Redis in the Laravel framework. Previous research, such as Ridhalri et al. [3], compared Redis and MySQL performance for news data delivery based on response time metrics. Julastri et al. [12] investigated Redis and PostgreSQL performance, testing FIFO, LFU, and LRU eviction algorithms across different network conditions (5G, 4G, 3G, and offline). These studies highlight Redis's significant potential as an in-memory caching system for accelerating web application performance. Averoes et al. [13] applied Redis as an in-memory caching mechanism in PHP web applications, while Joshi et al. [14] implemented Redis in a cloud-based payment gateway, measuring response time, throughput, and hit ratio. Su et al. [15] compared caching algorithms FIFO, LRU, VBBMS, Req-block, and HaParallel based on response time in SSDs. Chen et al. [16] compared LRU with GCaR-CFLRU, a hybrid of GCaR and CFLRU, on modern flash-based SSDs using response time metrics. Hou et al. [17] compared a user activity-based cache replacement strategy with LFU across varied object access scenarios in an information-centric network and electronic music composition system. Zulfa et al. [18] compared cache replacement algorithms LFU, FIFO, LRU, GDSF, GDS, SIZE, and LFUDA using the IRCache internet traffic dataset, finding that LRU exhibited excellent hit ratio performance for regular internet traffic. However, these studies have not comprehensively explored caching algorithm comparisons under varying user loads in Laravel, as addressed in this research.

II. RESEARCH METHOD

This research began with a literature review and concluded with an analysis of application testing results. The research phases are shown in **Figure 1**.

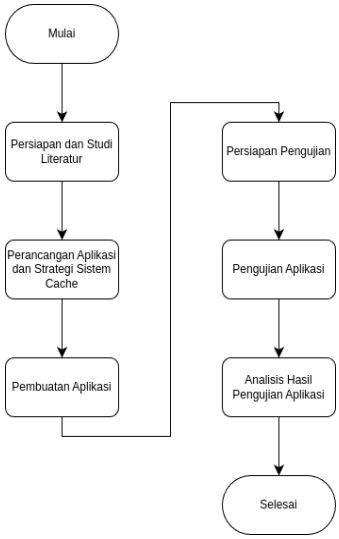


Figure 1. Research step

A. Preparation and Literature Review

The initial phase of the research began with a literature review to gain in-depth insights into the research topic, involving the search for references related to caching systems, cache replacement, RDBMS, IMDB, Redis, and web-based applications. This phase continued with formulating the research problem, defining research objectives, and identifying the benefits and limitations of the study. The aim was to provide a solid theoretical foundation and outline the direction of the research.

B. Application Design and Cache System Strategy

This phase involves designing the application and implementing the cache system to be used within it. Application design includes developing application features and defining how caching is integrated as a feature. The application was built using the Laravel framework with PHP programming language, and the data used was sourced from the IRCache dataset.

The IRCache dataset is an accurate global proxy dataset collected from proxy servers located in four cities in the United States: Urbana Champaign (UC), Boulder (BO2), Silicon Valley (SV), and New York (NY).

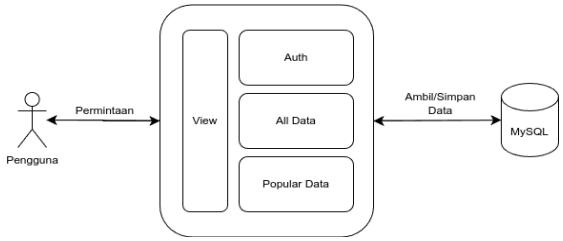


Figure 2. Application architecture

As shown in

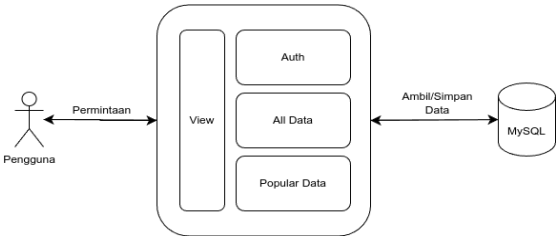


Figure *2*, the application features several key components: Auth, Popular Data, and All Data. The Auth feature enables users to log in and register to access the application. The Popular Data feature displays a table of cities with the highest data frequency based on the IRCache dataset, covering Urbana-Champaign (UC), Boulder (BO2), Silicon Valley (SV), and New York (NY), thus aiding in identifying access patterns and optimizing caching strategies. Meanwhile, the All Data feature provides full access to data from these cities, enabling in-depth analysis of data distribution, content types, sizes, and access times. By integrating the IRCache dataset, this application serves not only as a data management system but also as an analytical tool to understand user access patterns based on location and enhance caching efficiency.

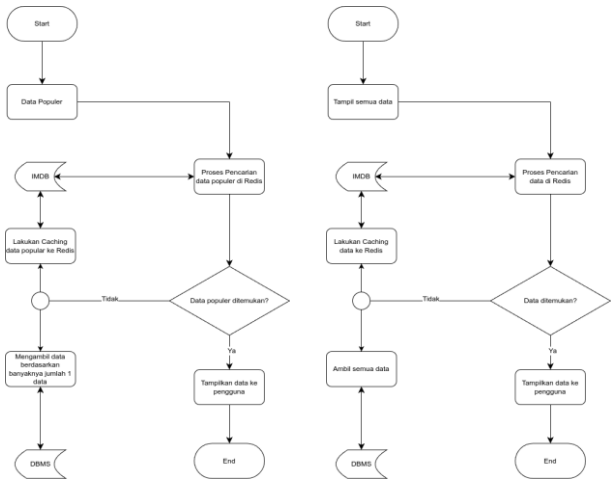


Figure 3. (a) popular data and (b) displaying all data.

Figure 3 illustrates the caching strategy workflow. The features to be implemented in the caching system are Popular Data and All Data. The data for the Popular Data feature will be retrieved based on the frequency of identical data IDs, with caching applied to this feature to ensure frequently needed data remains readily available, thus reducing query load. Caching for the All Data feature aims to alleviate the query burden of displaying all data, expected to reduce access time and lighten the database load. This research will utilize four algorithms LRU, LFU, FIFO, and Random Replacement. By comparing these four algorithms, the study will identify which algorithm performs better in handling cache storage when it reaches full capacity.

The data used from the IRCache dataset will include sections BO2, NY, SV, and UC. Each section consists of 19,000 data entries with variables including user ID, date, timestamp, client address, HTTP code, request method, size, URL, hierarchy data, and content type. Testing will be evaluated using the metrics RT (Response Time), TH (Throughput), and HR (Hit Ratio).

Table 1. Testing scheme for the all data feature

Feature	User	Maxmemory	City	Alg	Metric	Data
All Data	50	1 MB, …, 8 MB	UC, BO2, SV, NY	LRU, LFU RR FIFO	RT, TH, HT	19000

Table 2. Testing scheme for the popular data feature

Feature	User	City	Metric	Data
Popular Data	100, …, 800	UC, BO2, SV, NY	RT, TH	20

C. Application Development Phase

This phase will incorporate the Database Management System (DBMS) model, as shown in **Figure 1**, with an In-Memory Database (IMDB), as depicted in **Figure 1**, which retrieves data from the memory cache.

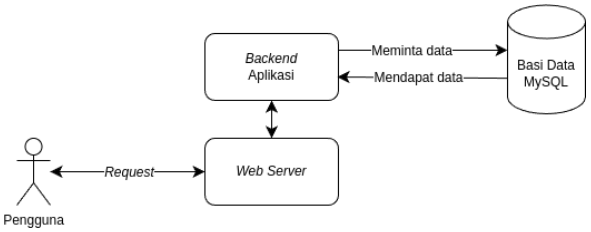


Figure 4. Database management system model

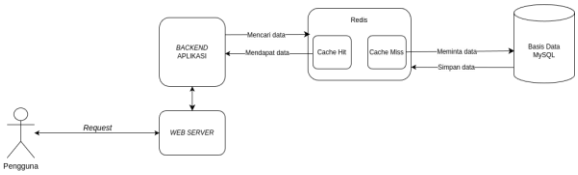


Figure 5. In-memory database model

Figure *5* illustrates the workflow when a user sends a data request to the application via the web server. The backend first checks the cache, if a cache hit occurs, the data is directly returned to the user. However, if a cache miss occurs, the data is retrieved from the database, stored in the cache, and then sent to the user. In contrast, **Figure *4*** depicts a system using only a DBMS, where every request is directly forwarded to the database without cache checking, and the results are returned to the user through the web server.

D. Application Testing Preparation

This phase involves setting up the testing environment using tools to simulate user interactions. Apache JMeter will be used to simulate users (threads) accessing application endpoints via HTTP GET methods. The application will be tested and compared before and after the implementation of the caching system.

Based on previous research references in the introduction, the testing metrics to be used are as follows:

1. *Response Time*

The time required for the server to respond

after receiving a request.

2. *Throughput*

   The amount of data transmitted per unit of time during the testing process.

3. *Hit Ratio*

   The ratio of successful data requests found in the cache compared to the total number of data requests. [12].

$$HitRatio = \frac{totalhit}{totalofallrequests} x100\%$$

E. Application Testing

This phase of the research involves testing the developed application based on predefined parameters: response time, throughput, and hit ratio. The testing will compare results across different users. Specifically, the popular data feature will be tested, as will maxmemory and four different algorithms applied to the all data feature.

F. Analysis of Application Testing Result

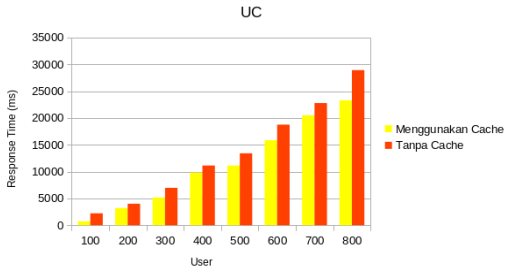
This phase is the final phase of this research. In this phase, an analysis is conducted on all data obtained from the application performance testing. After the analysis process is carried out based on the metrics measured during the testing, conclusions are then drawn based on the theories and literature that have been studied previously.

III. RESULTS AND DISCUSSION

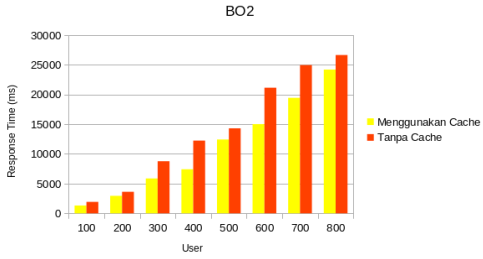
A. Testing of the Popular Data Feature Using Response Time and Throughput Metrics

In the Popular Data feature, testing was conducted using 8 different scenarios, each with a varying number of users: 100, 200, 300, 400, 500, 600, 700, and 800 users. Each test was configured with a ramp-up period of 1 second, meaning that virtual users were started incrementally every 1 second. The loop count was set to 1 so that each user would send a request only once. Additionally, within each thread group, an HTTP Request configuration was added to send a GET request to a specific server endpoint.

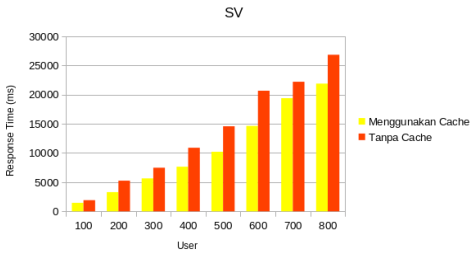
The endpoints used in this testing were divided into two types. The endpoint /populer/{city} was used to access data without caching, while the endpoint /populer/cache/{city} was used to access data with caching. The HTTP method used was GET, and the cities tested included four different locations: NY, BO2, SV, and UC.



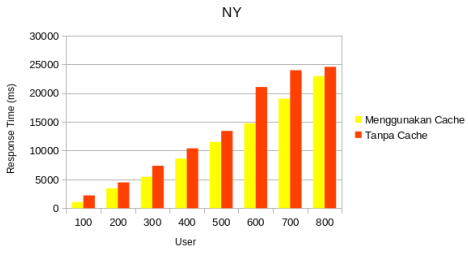
(a)



(b)



(c)



(d)

Figure 6. Response Time Testing Results for Each Dataset of the Popular Data Feature: (a) UC, (b) BO2, (c) SV, (d) NY

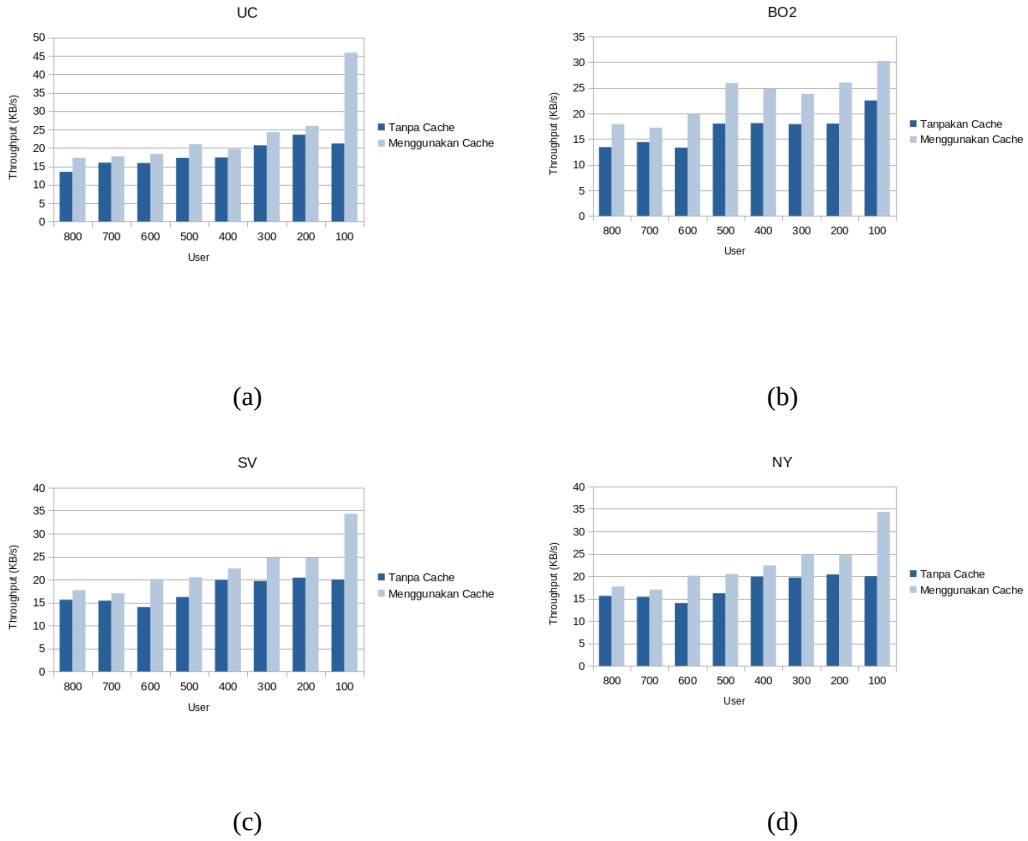


Figure 7. Throughput testing results for each dataset of the popular data feature: (a) UC, (b) BO2, (c) SV, (d) NY

Table 3. Average response time metric results for the popular data feature testing.

AVG Response Time (ms)	UC	BO2	SV	NY
Without Cache	13528	14199	13727	13438
Cache Implementation	11216	11078	10524	10867

Table 4. Hasil pengujian fitur data populer rata-rata matriks throughput

AVG Response Time (ms)	UC	BO2	SV	NY
Without Cache	18.2	17.5	16.9	17.65
Cache Implementation	23.7	23.8	23.2	22.6

After testing was conducted, **Figure 6** and **Figure 7** show that the implementation of a caching system is more effective compared to no cache, especially in the popular data feature. This is demonstrated by the decrease in response time and the increase in throughput as the number of virtual users increased from 100 to 800. Popular data that was previously retrieved from the MySQL database is stored in Redis, so subsequent requests can be served directly from memory, speeding up access time. Based on Table 3, cache implementation reduced the average response time in all tested cities: UC from 13,528 ms to 11,216 ms, BO2 from 14.199 ms to 11.078 ms, SV from 13.727 ms to 10.524 ms, and NY from 13.438 ms to 10.867 ms. This indicates the effectiveness of cache in accelerating the server's response time to frequently accessed data requests.

According to Table 4, the average throughput increased after the cache system was applied. In UC, throughput increased from 18.2 KB/s to 23.7 KB/s, BO2 from 17.5 KB/s to 23.8 KB/s, SV from 16.9 KB/s to 23.2 KB/s, and NY from 17.65 KB/s to 22.6 KB/s. This improvement indicates that caching not

only accelerates response time but also enhances server efficiency in handling a large number of simultaneous requests. The small data scale causes the performance difference between methods to be not very significant yet. The impact of caching would be more substantial if applied to heavy queries or large-scale data. Therefore, cache remains relevant and important as a performance optimization strategy, especially to anticipate spikes in user access load.

B. Testing the Show All Data Feature Using Response Time and Throughput Metrics

In the Show All Data feature, testing was conducted with 1 thread group scenario using 50 threads (users). The test was configured with a ramp-up period of 1 second, meaning virtual users were started incrementally every 1 second. The loop count was set to 1 so that each user only performed the request once. An HTTP Request configuration was added to send a GET request to a specific server endpoint.

There were 8 endpoints used in this test, divided into two types. The endpoint data/{city} was used to access data without cache, while data/cache/{city} was used to access data with cache. The HTTP method used was GET, and the tested cities included four different datasets: NY, BO2, SV, and UC.

Table 5. Testing results of the show all data feature for the four datasets using 50 threads.

Dataset	RT (ms) without cache	RT (ms) with cache	TH (KB/s) without cache	TH (KB/s) with cache
UC	21361	7971	1,2	3,1
BO2	21656	8481	1,1	2,9
SV	21255	7368	1,2	3,4
NY	21605	7293	1,1	3,5

Based on Table 5, the implementation of a caching system has been proven to improve application performance in terms of both response time and throughput metrics across the four cities UC, BO2, SV, and NY. For the response time metric, UC experienced a decrease from 21.361 ms to 7.971 ms, BO2 from 21.656 ms to 8.481 ms, SV from 21.255 ms to 7.368 ms, and NY from 21.605 ms to 7.293 ms. Meanwhile, throughput also increased significantly: UC rose from 1.2 KB/s to 3.1 KB/s, BO2 from 1.1 KB/s to 2.9 KB/s, SV from 1.2 KB/s to 3.4 KB/s, and NY from 1.1 KB/s to 3.5 KB/s. These results demonstrate that an in-memory caching system significantly improves response time and increases throughput when serving data requests, especially for large-scale data such as 19,000 rows.

C. Testing the Show All Data Feature Using Hit Ratio Metrics

This test was conducted by varying the maxmemory setting from 1 to 8 MB and applying four eviction algorithms: LRU, LFU, RR, and FIFO, following the scheme outlined in Table 1.

The initial step involved configuring Redis's maxmemory parameter using the command “CONFIG SET MAXMEMORY [value]” to limit memory usage, since Redis sets maxmemory to 0 by default (meaning it uses all available RAM).

To support the test, the maxmemory-policy was also set using the command “CONFIG SET MAXMEMORY-POLICY [algorithm]”, allowing Redis to automatically evict data based on the selected algorithm when memory is full. The effectiveness of each algorithm was evaluated by monitoring Redis statistics via the “INFO STATS” command, which displays the number of key hits and key misses.

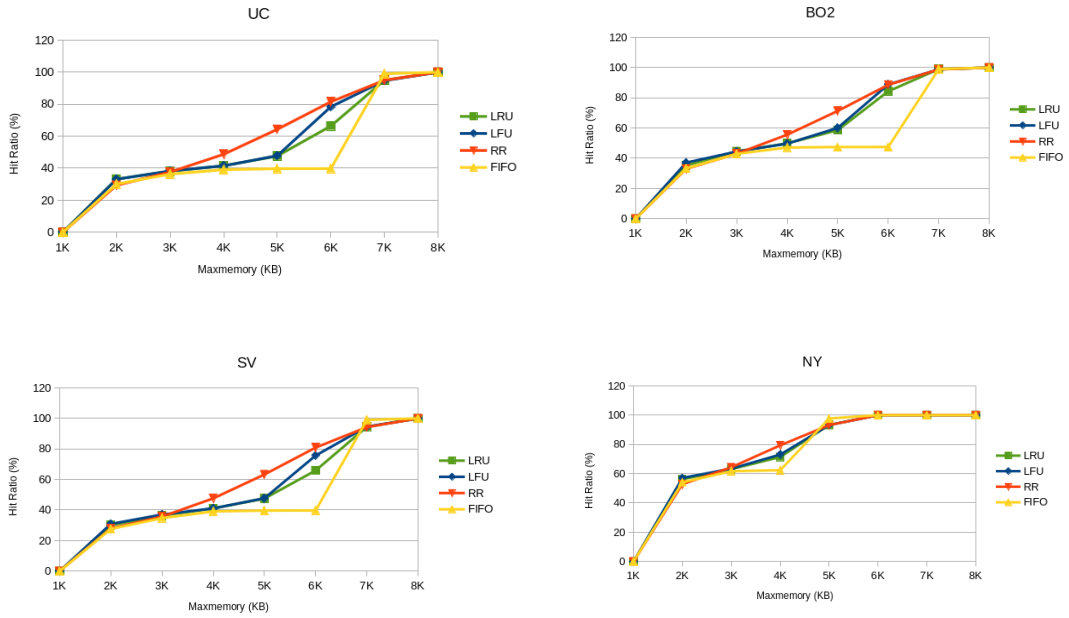


Figure 8. Comparison of hit ratio test result graphs for each dataset: (a) UC, (b) BO2, (c) SV, (d) NY

Table 6. Overall average hit ratio comparison.

Alg	Avg. Hit Ratio (%)				Avg. Total (%)
	UC	BO2	SV	NY	
LRU	52,72	58,93	51,91	73,01	59,14
LFU	54,22	59,86	53,37	73,36	60,20
RR	57,05	61,28	56,21	73,72	62,06
FIFO	47,99	52,14	47,46	71,98	54,89

Based on Table 6, the Random Replacement (RR) algorithm demonstrated the most consistent and superior performance, achieving the highest average hit ratio across all datasets particularly in NY with 73.72%, and an overall average of 62.06%. This was followed by LFU with 60.20%, and LRU with 59.14%. FIFO recorded the lowest average at 54.89%, indicating that strategies which do not consider access time or frequency are less effective. In contrast to the findings by Zulfa et al. [18], this result highlights that the RR algorithm excels under random load conditions across all IRCache datasets. The strength of RR lies in its simplicity and efficiency, as it does not require additional data structures like those used in LRU, LFU, or FIFO, thus reducing overhead while remaining competitive especially when data access patterns are random or unpredictable.

IV. CONCLUSION

The design of a caching system for web applications using the Redis in-memory database with the Laravel framework has been successfully implemented, where the caching system was effectively integrated into the application, enabling frequently accessed data to be stored in Redis to reduce the query load on the MySQL database. Redis proved effective in supporting a responsive and efficient application architecture. Performance testing of the caching system based on hit ratio, response time, and throughput demonstrated significant improvements in web application performance, with response time reduced by up to three times faster in scenarios involving large datasets of 19,000 entries and a notable increase in throughput. Additionally, in terms of cache replacement algorithm efficiency, the Random Replacement (RR) algorithm recorded the highest average hit ratio of 62.06%, followed by LFU, LRU, and FIFO, indicating that the choice of cache eviction strategy significantly impacts overall caching effectiveness. Besides Redis, other caching technologies such as Memcached or Varnish could be

used for comparison in terms of response time, throughput, and memory usage efficiency. This research confirms that cache management based on the RR algorithm provides high efficiency in scenarios involving massive usage and large datasets.

REFERENCE

[1] "Asosiasi Penyelenggara Jasa Internet Indonesia." Accessed: Feb. 25, 2025. [Online]. Available: https://apjii.or.id/berita/d/apjii-jumlah-pengguna-internet-indonesia-tembus-221-juta-orang

[2] Suliman, "Analisis Performa Website Universitas Teuku Umar Dan Universitas Samudera Menggunakan Pingdom Tools Dan Gtmetrix," *SIMKOM*, vol. 5, no. 1, pp. 24–32, Jan. 2020, doi: 10.51717/simkom.v5i1.47.

[3] R. Ridhalri, "PEMANFAATAN CACHING SYSTEM MENGGUNAKAN REDIS UNTUK SISTEM PENGELOLAAN INFORMASI AMBALAN ASHABUL KAHFI BERBASIS WEB," *J. DIALOGIKA Manaj. Dan Adm.*, vol. 4, no. 1, pp. 39–56, Dec. 2022, doi: 10.31949/dialogika.v4i1.3750.

[4] "MySQL :: MySQL 5.7 Reference Manual :: 8.10.3 The MySQL Query Cache." Accessed: Mar. 01, 2025. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/query-cache.html

[5] M. I. Zulfa, A. Fadli, and A. W. Wardhana, "Application caching strategy based on in-memory using Redis server to accelerate relational data access," *J. Teknol. Dan Sist. Komput.*, vol. 8, no. 2, pp. 157–163, Apr. 2020, doi: 10.14710/jtsiskom.8.2.2020.157-163.

[6] J. Yang, Y. Yue, and K. V. Rashmi, "A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter," *ACM Trans. Storage*, vol. 17, no. 3, pp. 1–35, Aug. 2021, doi: 10.1145/3468521.

[7] M. Kusuma, Widyawan, and R. Ferdiana, "Evaluasi Performa Web Server Menggunakan Varnish HTTP Reserve Proxy dan Redis Database Cache," *Pros. SENIATI*, vol. 2, no. 2, Art. no. 2, Mar. 2016, doi: 10.36040/seniati.vi0.824.

[8] E. S. br Haloho, N. A. Batubara, E. S. Situmorang, K. J. H. Sinaga, P. G. Sianipar, and I. Gunawan, "Analisis Manajemen Cache Sistem Operasi dalam Pengoptimalisasi Kinerja SSD Menggunakan Algoritma Least Recently Used (LRU)," *J. Inov. Artif. Intell. Komputasional Nusant.*, vol. 2, no. 1, Art. no. 1, Jan. 2025, doi: 10.260396/ejgxqk64.

[9] G. Hasslinger, J. Heikkinen, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "Optimum caching versus LRU and LFU: Comparison and combined limited look-ahead strategies," in *2018 16th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, Shanghai: IEEE, May 2018, pp. 1–6. doi: 10.23919/WIOPT.2018.8362880.

[10] "Eviction policy," Docs. Accessed: Mar. 01, 2025. [Online]. Available: https://redis.io/docs/latest/operate/rs/databases/memory-performance/eviction-policy/

[11] S. Ali, "Cache Replacement Algorithm," Jul. 30, 2021, *arXiv*: arXiv:2107.14646. doi: 10.48550/arXiv.2107.14646.

[12] B. A. Julastri, A. P. Sari, dan M. H. Prami Swari, "Pengelolaan Cache pada Aplikasi Pencatatan Penjualan Menggunakan Fuzzy Page Replacement Algorithm," *JIKO J. Inform. Dan Komput.*, vol. 8, no. 2, hlm. 404, Sep 2024, doi: 10.26798/jiko.v8i2.1319.

[13] F. Averoes, "Peningkatan Performa Apli kasi Web Dinamis Berbasis PHP melalui Implementasi Redis Caching," 2025.

[14] M. P. K. Joshi, "Redis Cache Optimization for Payment Gateways in the Cloud," 2024.

[15] L. Su, M. Lin, B. Mao, J. Zhang, and Z.Xu, "HaParallel: Hit Ratio-Aware Parallel Aggressive Eviction Cache Management Algorithm for SSDs," *ACM Trans Storage*, Apr. 2025, doi: 10.1145/3728644.

[16] H. Chen, Y. Pan, C. Li, and Y. Xu, "ECR: Eviction-cost-aware cache management policy for page-level flash-based SSDs," *Concurr. Comput. Pract. Exp.*, vol. 33, no. 15, p. e5395, 2021, doi: 10.1002/cpe.5395.

[17] R. Hou, "Performance analysis of cache replacement algorithm in information center network and construction of electronic music composition system," *Alex. Eng. J.*, vol. 61, no. 1, pp. 863–872, Jan. 2022, doi: 10.1016/j.aej.2021.04.082.

[18] M. I. Zulfa, A. Fadli, A. E. Permana sari, and W. A. Ahmed, "Performance comparison of cache replacement algorithms onvarious internet traffic," *J. INFOTEL*, vol. 15, no. 1, pp. 1–7, Feb. 2023, doi: 10.20895/infotel.v15i1.872.