JURNAL LOCUS: Penelitian & Pengabdian
Volume 5 No. 8 August 2026
E-ISSN 2829-7334| P-ISSN 2829-5439
Hompage: https://locus.rivierapublishing.id/index.php/jl

LRU-Based Caching Optimization for REST API and GraphQL in a PBB-P2 Information System

Rudi Setiawan Samosir*, David JM Sembiring, Roberto Kaban
Institut Teknologi dan Bisnis Indonesia, Indonesia
Email: rudisetiawansamosir@gmail.com*, davidjms366@gmail.com, roberto.kaban@yahoo.com

Keywords:
Application-Level Caching; GraphQL; LRU Cache; PBB-P2; Performance Testing; REST API

Abstract

The increasing demand for fast and reliable access to taxpayer, tax object, billing, and payment information requires efficient data retrieval mechanisms in PBB-P2 information systems. Repeated database access for frequently requested data can increase server workload and prolong response times. Application-level caching can reduce redundant database operations; however, cache performance depends on the replacement policy applied when memory capacity is limited. This study aimed to evaluate the performance impact of Least Recently Used (LRU) caching compared with simple bounded caching and no-cache conditions in REST API- and GraphQL-based PBB-P2 information systems. A quantitative comparative experimental method was employed using a Node.js prototype connected to a MySQL/MariaDB database. The experiment evaluated three cache conditions (no cache, simple cache, and LRU cache) across four workload patterns: random, repeat, mixed, and pressure. Performance was assessed using response time, throughput, cache hit ratio, database query count, CPU utilization, and eviction accuracy. The results showed that LRU provided the most significant improvement under high cache eviction pressure conditions. In REST API testing, LRU reduced response time from 10.41 ms to 7.89 ms and decreased database queries by 74.66%. In GraphQL testing, LRU reduced response time from 19.58 ms to 17.03 ms and decreased database queries by 74.96%. However, LRU was not consistently superior, particularly when workloads had sufficient cache capacity or irregular access patterns. The study concludes that LRU caching is effective when temporal locality and cache pressure are present; however, its implementation should be guided by workload characteristics and system telemetry.

INTRODUCTION

PBB-P2 administration requires rapid access to taxpayer, tax object, assessment, billing, and payment data (Azmar et al., 2025; Darmawan & Citrawan, 2025; Mandiri et al., 2024; Prabowo & Tambunan, 2024; Rahmawati & Putra, 2024; Yusuf et al., 2024). The same tax object may be accessed repeatedly within a short service session to verify its identification number, current-year liability, or payment status. Sending every read request directly to the database repeats processing tasks, increases database load, and may extend user waiting times. Application-level caching can reduce repeated database reads; however, a bounded cache also requires a replacement policy when the number of candidate objects exceeds the available memory capacity.

REST API and GraphQL expose application data through different execution models. REST commonly represents resources through HTTP endpoints, whereas GraphQL executes

client-defined fields through a typed schema and resolver pathways. Empirical studies have shown that latency, throughput, payload efficiency, and resource utilization depend on endpoint design, query structure, resolver implementation, and workload characteristics rather than solely on the API architecture (Neumann et al., 2021; Lawi et al., 2021; Śliwa & Pańczyk, 2021; Quiña-Mera et al., 2023; Muzaki & Salam, 2024; Chandra & Farisi, 2025; Bogner et al., 2023; Brito & Valente, 2020; Hartina et al., 2018; Lee et al., 2020; Hadinata & Stianingsih, 2024). JIKO has also discussed REST APIs in resource-constrained environments, although from a security perspective rather than a cache replacement perspective (Sinaga et al., 2025). These findings support the evaluation of both API models under consistent data and workload conditions.

Cache effectiveness is highly dependent on workload characteristics. Large-scale cache studies have identified popularity, working-set size, object lifetime, and temporal locality as major factors influencing data reuse (Yang et al., 2021). Replacement-policy evaluations have also demonstrated that hit ratio and computational overhead vary according to cache capacity and the proportion of unique requests (Zulfa et al., 2023; Zulfa et al., 2024; Wang et al., 2020; Inoue, 2021). LRU applies a temporal recency strategy in which a cache hit promotes an entry, while capacity pressure removes the entry that has remained unused for the longest period. Therefore, LRU is expected to preserve frequently accessed data under high-pressure conditions but may introduce additional management overhead when access patterns are random or when the active dataset already fits within the available cache capacity.

Previous API studies have generally focused on comparisons between REST and GraphQL, whereas cache replacement studies have primarily evaluated differences among caching algorithms. However, fewer studies have isolated the contribution of recency-based replacement strategies from the general benefits of in-memory data storage. This study addresses this gap by evaluating three controlled conditions: no cache (0), a bounded simple cache without recency promotion (S), and an LRU cache (L). Both API architectures were assessed under random, repeat, mixed, and pressure workload patterns. The primary research question was: under which workload conditions did LRU provide measurable user-facing and database-level benefits compared with a simpler bounded cache? The contribution of this study was a reproducible comparison framework and cache-policy validation based on latency, throughput, cache performance, database activity, resource utilization, and eviction telemetry within a PBB-P2 information-system context.

METHOD

Experimental Design and Cache Policies

A quantitative comparative experiment was conducted using a Node.js prototype that provided equivalent PBB-P2 read operations through REST endpoints and GraphQL resolvers. Both API models accessed the same MySQL/MariaDB schema and returned equivalent taxpayer, tax object, billing, and payment information. Client-side load metrics were collected using k6, while application telemetry recorded cache events, database query counts, CPU usage, resident set size (RSS), heap utilization, and system memory usage. The experiment varied only the API model, workload pattern, and cache condition, while business data and core request semantics were kept constant.

Condition 0 bypassed memory caching. Condition S used the same cache keys, maximum capacity, and time-to-live settings as Condition L; however, successful reads did not update recency priority, meaning that replacement decisions were not based on the latest access history. Condition L promoted entries upon every cache hit and evicted the least recently used entry when the cache reached its capacity limit. The comparison between Conditions S and L represented the primary evaluation of replacement-policy performance because cache existence, capacity, and expiration settings were controlled consistently. The comparisons between Conditions 0 and S, as well as Conditions 0 and L, were used to evaluate the overall benefits of caching.

Table 1. Controlled experimental configuration

Parameter	Value
API models	REST API and GraphQL
Conditions	0: no cache; S: simple; L: LRU
Virtual users	10
Iterations	5,000 per combination
Cache maximum	200 entries
Time-to-live	300,000 ms
Hot subset	50 items
Mixed traffic	85% hot access
Pressure traffic	5,000-item set; 90% hot access
Prewarming	Enabled for cached conditions
Resource sampling	Every 2,000 ms

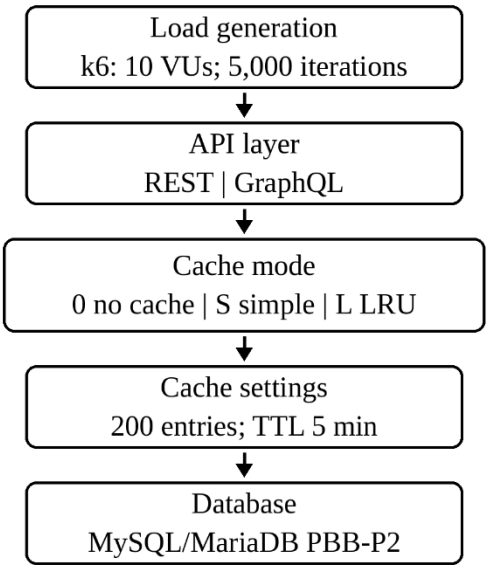

Figure 1. Architecture and controlled factors of the experiment

Workloads and Execution Controls

Four patterns represented different temporal-locality levels. Random selected objects from a broad set with little expected reuse. Repeat repeatedly selected a 50-item hot set that fit within the cache. Mixed directed 85% of accesses to the hot set and the remainder to less predictable objects. Pressure used a 5,000-item set against a cache capacity of 200 entries and

directed 90% of accesses to the hot set, intentionally creating eviction pressure. Every pattern was executed for REST and GraphQL under 0, S, and L, producing 2 × 4 × 3 = 24 combinations.

Each combination comprised 10 virtual users and 5,000 iterations. Cached conditions used a 200-entry maximum, a 300,000-ms time-to-live, and consistent hot-set prewarming before measured traffic. Resource telemetry was sampled every 2,000 ms. The same request counts and cache settings were used across corresponding S and L cases. No inferential significance claim is made because the available study dataset contains aggregate k6 and server telemetry rather than independent replicated experimental runs; conclusions are based on effect direction, magnitude, tail latency, resource trade-offs, and consistency across multiple metrics.

Metrics and Calculation

The primary effect compares S with L. Let T_s and T_l be mean response times, TP_s and TP_l be throughputs, Q_s and Q_l be database-query totals, H and M be cache-hit and cache-miss events, N be completed HTTP requests, σ be response-time standard deviation, μ be mean response time, E^c be evictions consistent with the expected victim, and E^t be all evictions. Positive response-time and query reductions indicate an LRU advantage, while a positive throughput change indicates higher LRU capacity. Cache telemetry counts cache lookup events, not HTTP requests; consequently H+M may exceed N when one request performs more than one cacheable lookup.

$$R_{RT} = \frac{T_S - T_L}{T_S} \times 100\% \quad (1)$$

$$\Delta TP = \frac{TP_L - TP_S}{TP_S} \times 100\% \quad (2)$$

$$S_p = \frac{T_S}{T_L} \quad (3)$$

$$HR = \frac{H}{H + M} \times 100\% \quad (4)$$

$$QPR = \frac{Q}{N} \quad (5)$$

$$R_{DB} = \frac{Q_S - Q_L}{Q_S} \times 100\% \quad (6)$$

$$CV = \frac{\sigma}{\mu} \times 100\% \quad (7)$$

$$EA = \frac{E_c}{E_t} \times 100\% \quad (8)$$

Mean latency was interpreted together with p95 and p99 to avoid hiding tail degradation. CV was used to normalize variability against different means. Database queries per request measured database pressure independent of raw request count, while eviction accuracy and hot-entry eviction verified whether LRU protected actively reused objects. CPU per request, RSS, heap, and system memory were treated as costs rather than assumed

improvements. The mechanism was considered supported only when the performance effect was consistent with hit ratio, database-query reduction, and eviction telemetry.

RESULTS AND DISCUSSION

In Tables 2–5, 0/S/L denotes no cache, simple cache, and LRU cache. S/L denotes the policy comparison. RT is response time, TP is throughput, HR is hit ratio, DBQ is database-query total, and EA is eviction accuracy.

REST API Results

Table 2. REST user-facing performance

Scenario	Mean RT 0/S/L (ms)	p95 S/L p99 S/L (ms)	TP S/L (req/s)
Random	16.09 15.63 14.91	24.11/23.74 32.79/34.17	487.96/506.91
Repeat	13.08 5.92 6.12	11.81/11.54 18.01/17.47	955.41/935.23
Mixed	13.92 13.58 13.34	23.63/23.71 30.77/30.79	549.34/550.78
Pressure	14.56 10.41 7.89	22.34/20.15 30.89/28.54	659.70/802.57

No-cache REST cases produced approximately 15,010 database queries for 5,000 requests, or about three queries per request. Random access remained near a 1% hit ratio, so neither cache substantially reduced database work. LRU improved mean response time only from 15.63 to 14.91 ms relative to S, which is a small 4.61% change and should not be interpreted as strong policy evidence.

Repeat isolated the generic cache benefit. Both S and L reached a 100% hit ratio and reduced database queries from 15,010 to 6, but S achieved 5.92 ms while L achieved 6.12 ms. The −3.42% S-to-L response-time reduction means LRU was slightly slower because the working set already fit within capacity and no replacement decision was needed. However, L had marginally lower p95/p99 than S (11.54/17.47 versus 11.81/18.01 ms), illustrating why the mean cannot be read alone.

REST pressure provided the clearest policy evidence. LRU reduced mean response time from 10.41 to 7.89 ms and p95/p99 from 22.34/30.89 to 20.15/28.54 ms, while throughput increased from 659.70 to 802.57 requests/s. Its hit ratio rose from 62.87% to 89.92%, database queries fell from 5,998 to 1,520, and CPU per request fell from 1.07 to 0.85 ms. L performed 908 evictions with 100% accuracy and no hot-entry eviction; S performed 1,444 evictions, evicted 587 hot entries, and reached 59.35% accuracy. This triangulation connects the performance gain to recency-aware residency rather than merely to the presence of memory caching. CV nevertheless increased from 61.17% to 70.07%, so relative variability remained a trade-off.

GraphQL Results

Table 3. GraphQL user-facing performance

Scenario	Mean RT 0/S/L (ms)	p95 S/L p99 S/L (ms)	TP S/L (req/s)
Random	24.42 25.05 24.87	37.22/36.70 47.40/48.36	335.05/335.80
Repeat	21.10 15.76 15.14	27.75/26.64 36.84/34.82	482.50/500.99
Mixed	24.63 20.54 23.32	32.93/36.63 45.09/49.25	394.99/351.74
Pressure	23.57 19.58 17.03	38.66/42.25 50.18/55.73	410.33/458.21

GraphQL random traffic also produced little reusable locality: L recorded 24.87 ms against 25.05 ms for S, with hit ratios below 1.4%. Under repeat, both caches reached 100% hit ratio and only 10 database queries; L modestly improved mean response time from 15.76 to 15.14 ms and throughput from 482.50 to 500.99 requests/s. The main improvement, however, was caching itself relative to the 21.10-ms no-cache baseline.

GraphQL mixed was the strongest counterexample to a universal LRU claim. S achieved 20.54 ms, 394.99 requests/s, a 35.26% hit ratio, and 10,687 database queries. L worsened to 23.32 ms, 351.74 requests/s, a 32.11% hit ratio, and 11,131 queries. Its response-time effect was −13.52% and database-query effect −4.15%. The observed sequence and resolver path therefore did not reward recency promotion, consistent with evidence that GraphQL cost depends on query and resolver structure (Lawi et al., 2021; Śliwa & Pańczyk, 2021; Quiña-Mera et al., 2023; Muzaki & Salam, 2024; Chandra & Farisi, 2025; Cha et al., 2020; Brito & Valente, 2020; Hartina et al., 2018; Lee et al., 2020).

GraphQL pressure again favored LRU at the mean and database levels. Mean response time improved from 19.58 to 17.03 ms, throughput from 410.33 to 458.21 requests/s, hit ratio from 61.94% to 89.82%, database queries from 6,139 to 1,537, and CPU per request from 2.07 to 1.87 ms. L made 918 accurate evictions with no hot-entry eviction, whereas S made 1,674 evictions, including 696 hot entries, at 58.42% accuracy. Unlike REST pressure, GraphQL p95/p99 worsened from 38.66/50.18 to 42.25/55.73 ms and CV increased from 52.17% to 61.21%. Thus, LRU protected the database and improved the mean without improving every part of the latency distribution.

Policy Effects and Mechanism Validation

Table 4. LRU effects relative to the simple cache

Case	HR S/L (%)	DBQ S/L	RT red. (%)	DB red. (%)
REST random	1.27/1.37	14,823/14,803	4.61	0.13
REST repeat	100/100	6/6	−3.42	0.00
REST mixed	33.83/34.83	10,790/10,667	1.82	1.14
REST pressure	62.87/89.92	5,998/1,520	24.20	74.66
GQL random	0.92/1.37	14,877/14,809	0.71	0.46
GQL repeat	100/100	10/10	3.89	0.00
GQL mixed	35.26/32.11	10,687/11,131	−13.52	−4.15
GQL pressure	61.94/89.82	6,139/1,537	13.02	74.96

Table 5. Eviction and CPU validation in mixed and pressure cases

Case	Evictions S/L	Hot evict. S/L	EA S/L (%)	CPU/req S/L (ms)
REST mixed	2,370/2,295	761/748	67.89/67.41	1.35/1.35
REST pressure	1,444/908	587/0	59.35/100	1.07/0.85
GQL mixed	2,184/2,429	749/716	65.71/70.52	2.28/2.48
GQL pressure	1,674/918	696/0	58.42/100	2.07/1.87

Across both APIs, the boundary is consistent. LRU added the most value when cache capacity was smaller than the active working set, requests retained a recurring hot subset, and eviction pressure was high enough for replacement order to matter. In REST and GraphQL pressure, LRU reduced database queries by 74.66% and 74.96% relative to S. The response-time reductions were smaller—24.20% and 13.02%—because serialization, resolver execution, network handling, and telemetry remain in the request path after a database read is avoided.

LRU added little value when reuse was absent or capacity was sufficient. Random traffic produced continual churn and near-zero hit ratios. Repeat benefited greatly from caching but not consistently from recency-aware replacement. Mixed traffic occupied an intermediate region: REST showed a small L advantage, while GraphQL favored S. This supports the workload-dependent behavior reported in cache studies (Yang et al., 2021; Zulfa et al., 2023; Zulfa et al., 2024; Wang et al., 2020; Inoue, 2021) and indicates that policy selection should follow measured locality and capacity pressure rather than a universal rule.

REST generally recorded lower response time and CPU per request than GraphQL in this prototype, but this is not an inherent ranking. It reflects the tested endpoints, schema, resolver path, payload, database access, and implementation. API style and cache policy should therefore be selected separately: the API should match client data requirements, while capacity, TTL, invalidation, and replacement should be tuned from production telemetry. HTTP freshness remains relevant at client and intermediary layers (Nottingham, 2022), whereas this experiment concerns application-level objects controlled inside the service.

Threats to Validity and Practical Implications

Internal validity was strengthened by identical request counts, data semantics, cache size, TTL, prewarming, and telemetry across corresponding S and L cases. Remaining threats include execution-order effects, operating-system noise, garbage collection, and the absence of independently replicated runs. Construct validity was improved by combining mean, p95,

p99, CV, cache, database, CPU, memory, and eviction measures; however, eviction accuracy depends on the telemetry definition of the expected victim. Cache events are lookup-level measures and must not be interpreted as HTTP-request counts.

External validity is limited to one Node.js prototype, one MySQL/MariaDB environment, a fixed capacity and TTL, and synthetic access patterns at 10 virtual users. The study does not cover write-heavy invalidation, distributed cache coordination, multi-node consistency, or alternative policies such as LFU, ARC, TinyLFU, or SIEVE. For deployment, LRU should first be enabled in observation mode or a limited rollout and retained only when production telemetry confirms a hot subset, capacity pressure, database reduction, and acceptable tail latency.

CONCLUSION

The experiment showed that LRU was not synonymous with caching itself and should be evaluated against an existing cache baseline. Its strongest contribution occurred under pressure conditions, where the working set exceeded cache capacity while a recurring hot subset remained. Compared with a bounded simple cache, REST pressure testing reduced the mean response time from 10.41 ms to 7.89 ms, increased throughput by 21.66%, raised the cache hit ratio to 89.92%, and reduced database queries from 5,998 to 1,520. GraphQL pressure testing reduced the mean response time from 19.58 ms to 17.03 ms, increased throughput by 11.67%, raised the cache hit ratio to 89.82%, and reduced database queries from 6,139 to 1,537. In both cases, eviction accuracy reached 100%, and no frequently accessed entries were evicted, confirming the expected behavior of the LRU replacement mechanism.

However, the policy was not universally superior. The simple cache condition (S) achieved slightly lower response times for REST repeat workloads because the active dataset already fit within the available cache capacity, while it performed substantially better for GraphQL mixed workloads. GraphQL pressure workloads also exhibited higher p95, p99, and coefficient of variation (CV) values despite achieving better mean latency and database efficiency. Therefore, LRU is most appropriate when system telemetry indicates the presence of temporal locality and cache pressure; however, it should not be adopted without considering workload characteristics. Future research should incorporate repeated experimental runs, production workload traces, multiple cache capacities and time-to-live (TTL) configurations, cache invalidation strategies for write operations, distributed caching environments, and comparisons with alternative cache replacement policies.

REFERENCES

Azmar, M., Junita, A., & Kusmanto, H. (2025). Effectiveness of Tax Object Information Management System in Improving PBB P2 Services. *Journal La Sociale*, *6*(2), 600–609.

Bogner, J., Kotstein, S., & Pfaff, T. (2023). Do RESTful API design rules have an impact on the understandability of Web APIs? *Empirical Software Engineering, 28*(6), Article 132. https://doi.org/10.1007/s10664-023-10367-y

Brito, G., & Valente, M. T. (2020). REST vs. GraphQL: A controlled experiment. In *2020 IEEE International Conference on Software Architecture (ICSA)* (pp. 81–91). IEEE. https://doi.org/10.1109/ICSA47634.2020.00016

Cha, A., Wittern, E., Baudart, G., Davis, J. C., Mandel, L., & Laredo, J. A. (2020). A principled approach to GraphQL query cost analysis. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 11–22). ACM. https://doi.org/10.1145/3368089.3409670

Chandra, S., & Farisi, A. (2025). Comparative analysis of RESTful, GraphQL, and gRPC APIs: Performance insight from load and stress testing. *Jurnal Sisfokom (Sistem Informasi dan Komputer), 14*(1), 81–85. https://doi.org/10.32736/sisfokom.v14i1.2315

Darmawan, R. T., & Citrawan, A. L. (2025). Effectiveness of Official Assessment System for PBB-P2 Tax Collection in Bekasi City. *Studia: Journal of Humanities and Education Studies*, *1*(2), 14–27.

Hadinata, W., & Stianingsih, L. (2024). Analisis perbandingan performa RESTful API antara Express.js dengan Laravel Framework. *Jurnal Informatika dan Teknik Elektro Terapan, 12*(1). https://doi.org/10.23960/jitet.v12i1.3845

Hartina, D. A., Lawi, A., & Panggabean, B. L. E. (2018). Performance analysis of GraphQL and RESTful in SIM LP2M of Hasanuddin University. In *Proceedings of the 2nd East Indonesia Conference on Computer and Information Technology* (pp. 237–240). IEEE. https://doi.org/10.1109/EIConCIT.2018.8878524

Inoue, H. (2021). Multi-step LRU: SIMD-based cache replacement for lower overhead and higher precision. In *2021 IEEE International Conference on Big Data* (pp. 174–180). IEEE. https://doi.org/10.1109/BigData52589.2021.9671363

Lawi, A., Panggabean, B. L. E., & Yoshida, T. (2021). Evaluating GraphQL and REST API services performance in a massive and intensive accessible information system. *Computers, 10*(11), Article 138. https://doi.org/10.3390/computers10110138

Lee, E., et al. (2020). Performance measurement of GraphQL API in home ESS data server. In *Proceedings of the International Conference on Information and Communication Technology Convergence*. IEEE. https://doi.org/10.1109/ICTC49870.2020.9289569

Mandiri, P., Alfitri, A., Thamrin, M. H., & Najib, A. (2024). Modernizing local tax policy: the role of digitalization in land and building tax administration. *Journal Public Policy*, *10*(3), 204–213.

Muzaki, R. N., & Salam, A. (2024). Reducing under-fetching and over-fetching in REST API with GraphQL for web-based software development. *Jurnal Teknik Informatika (JUTIF), 5*(2), 447–453. https://doi.org/10.52436/1.jutif.2024.5.2.1725

Neumann, A., Laranjeiro, N., & Bernardino, J. (2021). An analysis of public REST web service APIs. *IEEE Transactions on Services Computing, 14*(4), 957–970. https://doi.org/10.1109/TSC.2018.2847344

Nottingham, M. (2022). *HTTP caching* (RFC 9111). Internet Engineering Task Force. https://doi.org/10.17487/RFC9111

Prabowo, B. E., & Tambunan, M. R. U. D. (2024). Factors Influencing Taxpayer Compliance In The Implementation Of The Land And Building Tax (PBB-P2) Policy In The Revenue Office Districts Of North Jakarta Administrative City And The Kepulauan Seribu Regency. *Eduvest-Journal of Universal Studies*, *4*(4), 1756–1766.

Quiña-Mera, A., Fernandez, P., García, J. M., & Ruiz-Cortés, A. (2023). GraphQL: A systematic mapping study. *ACM Computing Surveys, 55*(10), Article 202. https://doi.org/10.1145/3561818

Rahmawati, R., & Putra, R. (2024). Performance of The Public Service Bureaucracy: Urban, Rural, and Building Tax Service Model (PBB-P2) Based on Information and Technology. *BISNIS & BIROKRASI: Jurnal Ilmu Administrasi Dan Organisasi*, *31*(2), 64–73.

Sinaga, R., Samsinar, S., Fatima, S., & Frangky, F. (2025). Analysis of security challenges in REST API in edge computing-based IoT ecosystem: A review. *JIKO (Jurnal Informatika dan Komputer), 8*(2), 153–160. https://doi.org/10.33387/jiko.v8i2.10097

Śliwa, M., & Pańczyk, B. (2021). Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. *Journal of Computer Sciences Institute, 21*, 356–361. https://doi.org/10.35784/jcsi.2744

Wang, Y., Yang, J., & Wang, Z. (2020). Dynamically configuring LRU replacement policy in Redis. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)* (pp. 272–280). ACM. https://doi.org/10.1145/3422575.3422799

Yang, J., Yue, Y., & Rashmi, K. V. (2021). A large-scale analysis of hundreds of in-memory key-value cache clusters at Twitter. *ACM Transactions on Storage, 17*(3), 1–35. https://doi.org/10.1145/3468521

Yusuf, O. S., Sholahuddin, A., & Suhartono, T. (2024). Absorption of Rural and Urban Land and Building Tax (PBB-P2) Through the Tax Object Management Information System: Study of Implementation of Tax Object Management Information System Policy Based on East Kutai Regent Regulation Number 44 of 2013 in the East Kutai Regency Area. *International Journal of Research in Social Science and Humanities (IJRSS) ISSN: 2582-6220, DOI: 10.47505/IJRSS*, *5*(2), 49–63.

Zulfa, M. I., Fadli, A., Permanasari, A. E., & Ahmed, W. A. (2023). Performance comparison of cache replacement algorithms on various internet traffic. *Jurnal INFOTEL, 15*(1), 1–7. https://doi.org/10.20895/infotel.v15i1.872

Zulfa, M. I., Maryani, S., Ardiansyah, T., Widiyaningtyas, T., & Ali, W. (2024). Application-level caching approach based on enhanced aging factor and Pearson correlation coefficient. *JOIV: International Journal of Informatics Visualization, 8*(1), 31–37. https://doi.org/10.62527/joiv.8.1.2143