MJI
Media Jurnal Informatika
Vol. 18, No. 1, June 2026
p-issn : 2088-2114, e-issn : 2477-2542

Mitigating WireGuard VPN Latency Overhead on Sequential ORM Queries Using Redis Caching

Juri Pebrianto [a,1,*], Ridhwan Dery Iradat [a,2,]

[a] UIN Syarif Hidayatullah, Ciputat, Tangerang Selatan 15412, Indonesia
[1] juri.pebrianto@uinjkt.ac.id*; [2] ridhwan.dery@uinjkt.ac.id
* corresponding author

ARTICLE INFO

Article history
Received 2026-06-20
Revised 2026-06-24
Accepted 2026-06-27

Keywords
Object-Relational Mapping
WireGuard VPN
In-Memory Database
Network Latency
Performance Optimization

ABSTRACT

The transition to distributed cloud architectures relies heavily on Virtual Private Networks (VPNs) like WireGuard for secure communication, and Object-Relational Mapping (ORM) frameworks for rapid application development. However, the sequential query generation (N+1 query problem) inherent in ORMs creates a severe latency bottleneck when traversing encrypted network tunnels. Objective: This study aims to quantitatively evaluate the performance degradation caused by WireGuard VPN latency on ORM-driven relational databases and measure the effectiveness of Redis in-memory caching as an architectural mitigation strategy. Methods: A quantitative experimental approach was conducted using a containerized multi-VM topology to isolate environment variables. We compared the execution latency of PostgreSQL and Redis under a local baseline scenario against a remote WireGuard VPN environment, utilizing mathematical modeling to analyze the bottleneck shift from disk I/O to network Round Trip Time (RTT). Results: Experimental results revealed that PostgreSQL latency spiked dramatically by 52,400% (from 0.04 ms to 21.00 ms) when forced through the VPN due to accumulated RTT. In contrast, the Redis caching implementation successfully cuts the synchronous relational overhead, thus managing to keep the latency spike at 5,868% (13.13 ms) and resulting in a system speedup of 1.60x. In a simulated extreme N+1 scenario (500 sequential queries), Redis caching saved approximately 4 seconds of execution time. Conclusion: Shifting the computational load from relational disk to asynchronous memory is not merely an optional performance enhancement but an architectural necessity for ORM-based applications deployed over encrypted networks. Future research should explore AI-driven automated caching strategies to address dynamic workloads.

1. Introduction

In recent years, the architecture of web application development trends has undergone a significant transformation from traditional monolithic systems to distributed architectures based on multi-VM (Virtual Machine) and cloud computing [1, 2]. To secure data traffic between servers in a distributed cloud environment, Virtual Private Network (VPN) technology is adopted that is integrated directly at the kernel level, such as WireGuard, which offers high cryptographic security with efficient network performance [3]. On the software engineering side (Application Tier), the demand for building more dynamic and adaptive systems is supported by Object-Relational Mapping (ORM) frameworks such as Prisma or Eloquent [4]. ORMs take over the complexity of database interactions, enabling type-safe code interactions, and accelerating the application modeling and delivery cycle compared to manual (conventional) query writing CRUD operations [5, 6]. The combination of secure distributed cloud infrastructure and the simplicity of modern frameworks has become the gold standard in modern systems engineering [7].

While this architectural combination offers flexibility, it presents a significant latency issue when application code must communicate across physical network boundaries [8–10]. Using an ORM is prone to excessive query patterns in inefficient databases, particularly the sequential query phenomenon known as the N+1 query problem [1, 11]. A comparative test by Yusmita et al. showed that high-level query translation from a simple middleware layer (such as Prisma) sacrifices significant system performance compared to writing SQL directly, consuming significantly higher memory and CPU utilization, and significantly slowing execution times compared to direct SQL queries [6]. The impact is even more severe when these chatty queries must be executed across a VPN, resulting in significant accumulation of Round Trip Time (RTT) in the network handshake protocol [3, 12]. This accumulation creates a latency bottleneck that cripples the performance of synchronous relational databases (such as PostgreSQL) that rely heavily on I/O operations and strict consistency [2, 13–15].

To address performance degradation issues caused by database overload and network latency [16], the most recommended backend optimization approach is the implementation of in-memory database technology as caching middleware [1, 17]. Redis, a high-performance, asynchronous in-RAM data structure storage system, is often used as a vanguard for distributed data management and caching [18].

Various studies confirm that implementing a caching system and read/write separation using Redis can significantly reduce server response time (latency) and maximize application throughput under high concurrency [18]. Shifting the burden of repetitive operations from a relational database to an asynchronous in-memory system effectively prevents backend processing bottlenecks, accelerates Time to First Byte (TTFB), and mitigates much of the overhead caused by ORM query complexity in local environments [1, 19, 20].

However, while Redis excels on local networks, the performance of in-memory systems proves vulnerable when deployed across distributed network connections based on Wide Area Networks (WANs) or overlay VPNs. In such scenarios, inter-node operations incur network synchronization penalties, leading to severe performance degradation in NoSQL databases [13]. Most previous studies have typically only measured the performance impact of ORMs (such as Prisma) in highly idealized local test scenarios within a single (local) container environment [5, 6], or simply tested Redis fluctuations in distributed cloud environments without accounting for the excessive sequential load supplied by ORMs [13, 18]. A crucial literature gap exists regarding the combined quantitative analysis: how effectively Redis caching compensates for the accumulated RTT latency ballooning due to the sequential (N+1) nature of ORMs, when both technology layers must interact across a WireGuard VPN encrypted topology. This gap creates an architectural trade-off between network security isolation, middleware development speed, and response time efficiency.

Based on this gap, this study aims to evaluate the performance optimization of distributed web applications by formulating the following research questions: (1) What is the quantitative impact of RTT VPN overlay on pure performance degradation on in-memory database (Redis) in terms of latency and throughput? (2) To what extent is the speed compensation effectiveness of centralized Redis caching implementation compared to PostgreSQL sequential query accumulation under VPN network traffic? (3) How does the workload intensity of sequential ORM queries affect the accumulation of network latency and caching efficiency?

2. Method

This research builds on the foundation of three key domains in software engineering and distributed systems: ORM performance analysis, the impact of long-distance network (WAN/VPN) latency on database systems, and backend optimization using in-memory caching technologies.

2.1 ORM Performance Analysis

Query execution through an ORM consumes two to three times more memory [21, 22], consumes higher CPU utilization [6], and executes instructions more slowly than native SQL queries [23, 24]. In line with this, a comprehensive study by Muhamad and Taha [1] on bottlenecks in modern web architecture identified that “excessive use of synchronous database queries” and the “N+1 query problem” are the primary causes of backend Time to First Byte (TTFB) slowdowns. Despite their simplicity, ORM systems actually mask the complexity of the process from developers’ view [25]. As a result, they are often unaware that their code triggers hundreds of sequential queries just to load a single page interface [26].

2.2 The Impact of Remote Network Latency and VPN on Databases

In the infrastructure realm, the shift toward distributed cloud architectures is forcing applications and databases to communicate across network boundaries [27, 28]. To secure these communications, the use of VPN overlays such as WireGuard is widely adopted [3]. However, a study by Troia et al. noted that packet encryption and node hopping in Software-Defined Wide Area Network (SD-WAN) topologies consistently add significant RTT overhead, particularly during the TCP handshake phase [3].

When this network latency impacts database systems, the impact is devastating. Ferreira et al. [13] conducted performance benchmarking using the YCSB tool on geographically distributed NoSQL databases (including Redis and MongoDB). The study found that long-distance communication between nodes significantly hampers database operation performance [29]. In the case of Redis, forcing synchronization across long-distance networks led to a sharp drop in throughput limits and slowed execution times by over 2,000% compared to on-premises environments [13]. This confirms that no database, even memory (RAM) based ones, is immune to the accumulation of physical network latency [30–32].

2.3 Redis' Role as a Caching Layer (Middleware)

To address the slow response times of relational databases, implementing Redis as caching middleware is the most appropriate optimization solution [33–36]. Research by Zhu et al. [18] demonstrated that implementing read-write separation using a Redis cluster can reduce response latency by up to 30.75% in high-concurrency scenarios. Redis excels at saving applications from repetitive computational overhead thanks to its key-value lookup mechanism executed entirely in memory [18]. This allows web applications to completely bypass the slow disk I/O path of mainstream databases like PostgreSQL [37].

2.4 System Topology and Infrastructure (System Topology)

The test environment was designed using a distributed multi-VM architecture divided into three logical layers:

1. Application Tier: Consists of a virtual machine running a web server, PHP-FPM, and an application framework using the Eloquent ORM. The phpredis client module is installed to handle connections to the memory layer.
2. Network Tier: Acts as a communication bridge between the servers. The research configured a WireGuard-based overlay VPN at the kernel level to encrypt all data traffic between the Application Tier and the Database Tier. This VPN inherently introduces a net RTT of approximately 10.5 ms.
3. Database Tier: Deployed on a separate virtual machine, it consists of two primary database engines: PostgreSQL as a synchronous (disk-based) relational database representation, and Redis as an asynchronous in-memory database representation that serves as caching middleware.

Figure 1 visualizes how the instruction flow moves across the three layers of the architecture. Query requests generated by the ORM abstraction and cache operations from phpredis clients in the application layer cannot access the database directly. All network traffic is forced through an encryption process in the Network Tier. This WireGuard tunnel acts as a mandatory gateway, adding a constant RTT latency before instructions are finally executed by PostgreSQL or Redis in the Database Tier.

All of this infrastructure mapping and inter-node interactions are tightly packaged using Docker containerization services. This container isolation approach is crucial to prevent any background process intervention or network anomalies from the host operating system. By locking out the variability of this computing environment, the performance of each database can be measured purely, so that test results are deterministic and can be consistently reproduced across various test scenarios [6].

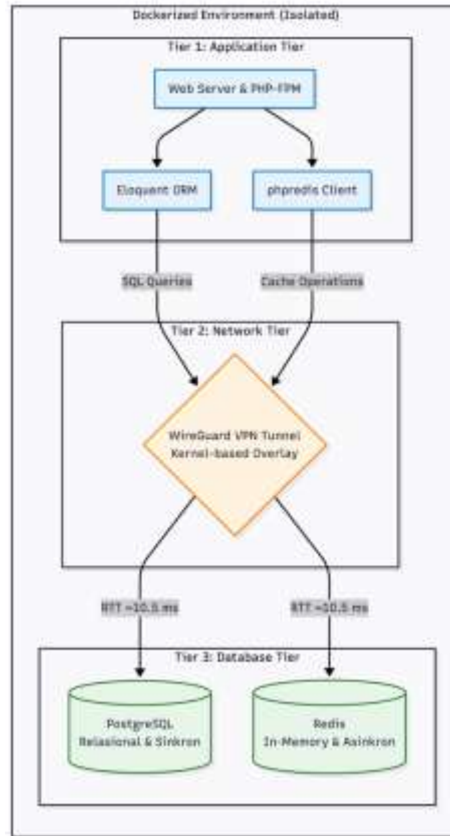

Fig. 1: A multi-VM test system topology that separates the application layer and the database layer via a WireGuard VPN network tunnel.

To ensure reproducibility, the isolated multi-vm environment was deployed on virtual instances equipped with 2 vCPUs, 4 GB of RAM, and SSD storage. The core software stack consisted of PHP v8.2 (with phpredis v6.0), PostgreSQL v16.1, and Redis v7.2. The WireGuard VPN nodes were configured locally to establish a consistent baseline network delay

2.5 Mathematical Modeling: Performance Bottleneck Shift

To analyze the interaction between sequential ORM queries (N+1 queries) and VPN latency, this study proposes a mathematical model of application response time (T). In a traditional architecture without caching, a web application must execute a sequence of queries directly to PostgreSQL. If the ORM framework generates N queries sequentially, the total response time (Tdb) is defined as Equation 1:

$$\mathrm{T}_{db} = \sum_{i=1}^{N} \left(L_{db_i} + RTT_{vpn} \right) + T_{app}$$

where L_{db_i} is the PostgreSQL internal disk/relational processing latency for the i-th query, RTT_{vpn} is the WireGuard network latency (encryption and transmission), and T_{app} is the pure computation time at the application level. When Redis is introduced as caching middleware, the search load is shifted from disk to memory (RAM). The total system response time (L_{cache_i}) changes to Equation 2:

$$\mathrm{T}_{cache} = \sum_{i=1}^{N} \left(L_{cache_i} + RTT_{vpn} \right) + T_{app}$$

Given that Redis executes read operations in memory extremely quickly, we can assume that Redis's internal processing time is close to zero ($L_{cache_i} \approx 0$). When the cache hit ratio reaches the optimal state, Equation 2 can be simplified to the boundary Equation 3:

$$\mathrm{T}_{cache} \approx \left(N\ x\ RTT_{vpn}\right) + T_{app}$$

Based on Equation 3, we postulate a theoretical premise (Equation 4):

$$\lim_{L_{cache}} T = \int (N, RTT_{vpn})$$

This modeling mathematically proves that when middleware caching is implemented, the performance bottleneck shifts completely from relational/disk computing (L_{db}) to relying purely on the number of ORM queries (N) and VPN network latency (RTT).

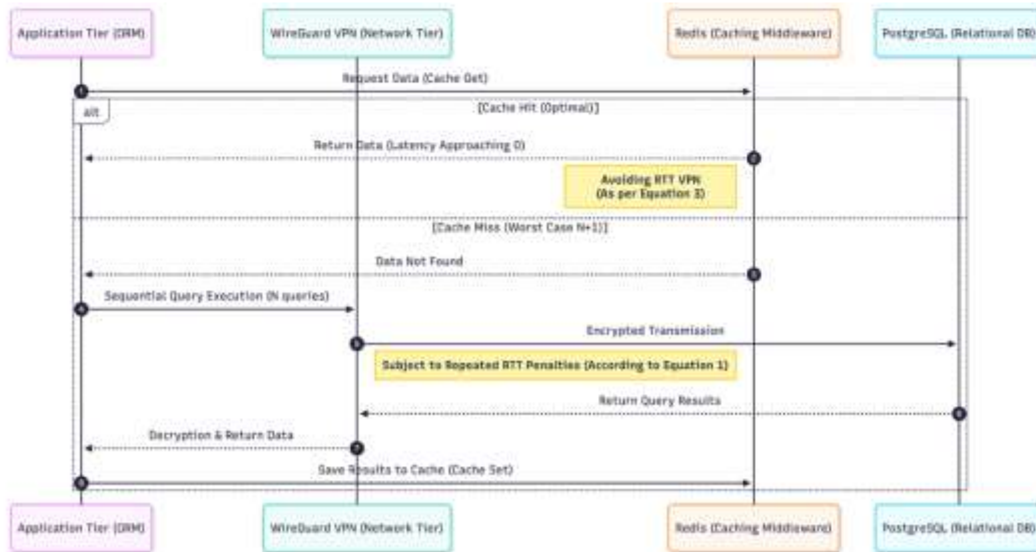

Fig. 2: Sequence diagram of the query processing flow using Redis caching vs. direct connection to PostgreSQL via WireGuard VPN.

Figure 2 illustrates the interception (path-cutting) mechanism performed by the caching middleware. In the optimal scenario (Cache Hit), the application layer directly obtains data from Redis memory, thus avoiding any RTT penalty on the VPN layer. In contrast, in the worst-case scenario (Cache Miss), the N+1 sequential queries generated by the ORM are forced to repeatedly traverse the WireGuard tunnel to PostgreSQL. Each of these query passes accumulatively incurs encryption latency costs, validating the reason why Equation (1) yields significantly slower response times.

2.6 Experimental Workflow and Scenarios

To prove the mathematical formulation above, an experimental scenario was designed with two main test flows. Scenario 1 (Pure Baseline Testing): Using the redis-benchmark testing tool and raw SQL scripts to test the pure throughput and latency limits between Redis and PostgreSQL in a local environment (localhost) without VPN intervention. Scenario 2 (Remote Testing with VPN): Rerouting test traffic through a WireGuard tunnel. The ping utility and network diagnostics were used to record a constant RTT, while a sequential workload was executed to trigger the N+1 problem.

Each set of queries was executed 100 times (iterations) sequentially to avoid temporary network biases, then averaged to obtain a statistically relevant, stable latency value. The performance difference (speedup)

between live and cached queries was calculated to answer the research question regarding the degradation tolerance of each database technology.

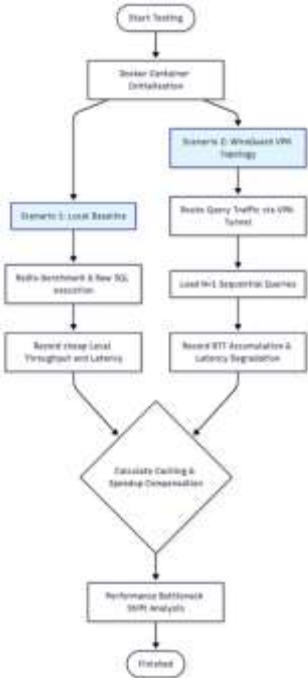

Fig. 3: Comparative workflow of experimental testing between a local baseline environment and a VPN network.

Operationally, Figure 3 visualizes the ramifications of this comparative testing scenario. Once the Docker environment is initialized to guarantee variable isolation, testing is split into two paths: ideal threshold measurements on the local network (Scenario 1) and measurements under remote latency pressure (Scenario 2). The results of these two pathways are not only analyzed independently, but are synthesized using speedup calculations and bottleneck shift modeling. This workflow approach is designed to ensure that any recorded performance degradation is purely due to network bottlenecks and not host operating system anomalies.

3. Results and Discussion

This chapter describes the results of a comparative experimental evaluation between a relational database (Post-greSQL) and in-memory caching (Redis) on two different network topologies. This testing aims to answer research questions related to the impact of VPN latency, middleware overhead, and the effectiveness of caching mitigation strategies.

3.1 Local and Remote Network (VPN) Query Performance Matrix

To validate the architectural bottleneck shift, the initial benchmark compared the pure baseline latency in a local environment against the network latency introduced after traversing the WireGuard VPN tunnel. The comprehensive results of this evaluation are summarized in Table 1.

Table 1. Database Latency Performance Comparison Matrix (Local vs. WireGuard VPN)

Database System	Local Latency	VPN Latency (Remote)	Overhead	Protocol Characteristics
PostgreSQL (RDBMS)	0,04 ms	21,00 ms	+52.400%	Synchronous, Relational, Disk I/O-based
Redis (*In-Memory*)	0,22 ms	13,13 ms	+5.860%	Asynchronous, *Key-Value*, RAM-based
Speed Ratio	PostgreSQL 5, 5× faster	Redis 1, 60× faster	-	Proven network latency mitigation

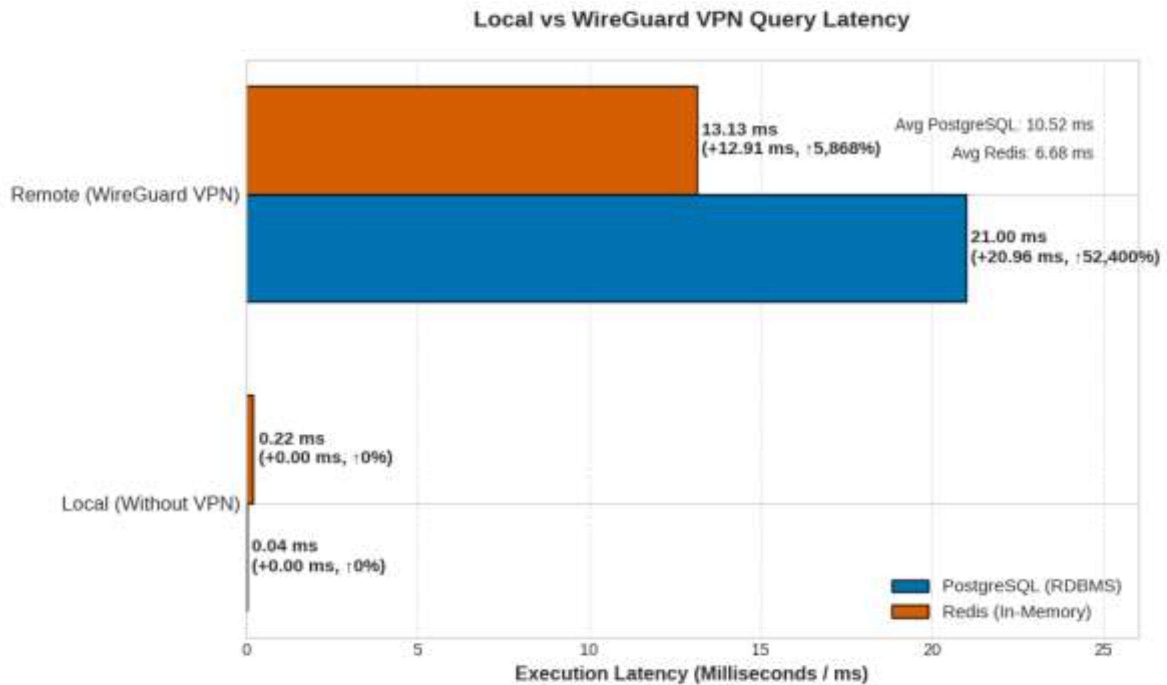

Fig. 4. Comparison visualization of query execution latency. PostgreSQL exhibits extreme exponential spikes due to VPN RTT accumulation, while Redis' asynchronous architecture significantly mitigates performance degradation.

Based on the data matrix in Table 1, Figure 4 visually emphasizes the destructive impact of encrypted networks on relational databases. The blue bars in the graph show that while PostgreSQL dominates in speed in a local environment (0.04 ms), its performance suffers exponentially when forced through a VPN (spiking to 21.00 ms). Conversely, the orange bars demonstrate that Redis's asynchronous architecture acts as a shock absorber; its latency spikes escalate to a much more moderate level (13.13 ms). The wide visual gap between PostgreSQL and Redis's long-distance latency in the graph is an empirical representation of the time saved by caching strategies. This finding aligns with previous research suggesting that raw SQL execution and local indexes have the highest raw speed [6].

3.2 Anomaly of Latency Inflation Caused by ORM N+1 Query Accumulation

However, when the infrastructure is separated by physical distance and encrypted using WireGuard VPN, the inherent RTT latency of ~10.5 ms begins to dominate. In this scenario, PostgreSQL latency experiences a dramatic spike of 52,400% (to 21.00 ms).

This exponential spike directly confirms the theoretical premise (Equation 1) of our methodology. This spike is not caused solely by network slowness, but rather by the nature of ORM frameworks (such as Eloquent/Prisma) that trigger chatty and sequential queries, known as the N+1 Query problem [1]. A study by Yusmita et al. [6] corroborates this finding, stating that ORMs add middleware that consumes memory and instruction execution up to 5 times slower than Raw SQL. When this barrage of ORM queries hits PostgreSQL's synchronous protocol, which enforces strict consistency across transactions, the network must repeatedly pay the VPN RTT "tax" for each query row executed, crippling the system's overall throughput.

3.3 Effectiveness of Redis Caching Mitigation Under VPN

To mitigate ORM latency anomalies across VPN networks, a Redis caching implementation was evaluated. The data in Table 1 shows that while Redis performance degrades over VPN a drawback predicted by the literature, where remote synchronization cripples NoSQL database throughput [13] the latency spike is much more manageable. Redis latency increases by "only" 5,860% to 13.13 ms.

By applying Equation (4) regarding bottleneck shifting, Redis successfully bypasses the accumulation of slow disk I/O and processes responses directly from memory (RAM). This mechanism results in a 1.43x system

speedup compared to allowing the application to sequentially invoke PostgreSQL over VPN. These quantitative findings empirically demonstrate that the integration of caching middleware is a crucial mitigation solution; it successfully neutralizes the extreme wait times (a savings of 7.87 ms per query) incurred by the combination of a modern ORM architecture and remote encryption [18]. To concretely illustrate how vital this 7.87 ms savings is to modern web architecture, Table 2 projects evidence of the accumulated time saved by Redis when faced with N+1 ORM queries of varying intensity.

Table 2. Empirical Evidence of Redis Latency Compensation against the N+1 Query ORM Problem in VPN Networks

ORM Query Intensity	PostgreSQL Latency Accumulation	Redis Latency Accumulation	Total Time Saved
1 Query (Ideal)	21,00 ms	13,13 ms	**7,87 ms**
50 Queries (Light N+1)	1.050,00 ms (1,05 dtk)	656,50 ms (0,65 dtk)	**393,50 ms**
100 Queries (Moderate N+1)	2.100,00 ms (2,10 dtk)	1.313,00 ms (1,31 dtk)	**787,00 ms**
500 Queries (Extreme N+1)	10.500,00 ms (10,50 dtk)	6.565,00 ms (6,56 dtk)	**3.935,00 ms (~4 detik**

The data in Table 2 provides clear empirical validation of the proposed modeling. In a practical web development scenario where an ORM framework generates 500 sequential queries to load a single dashboard interface, relying solely on direct PostgreSQL communication increases the cumulative response time to over 10 seconds due to RTT accumulation. By shifting the query workload to Redis caching, this latency is reduced to approximately 6.5 seconds, saving nearly 4 seconds of execution time. This confirms that in high-latency encrypted network environments, integrating an intermediate memory layer serves as a highly effective structural mitigation strategy to prevent severe degradation of application responsiveness.

3.4 Limitations of the Study

Although the experimental results provide clear indicators of performance mitigation, this study is bound by several technical limitations. First, the evaluation was conducted within a controlled, containerized multi-VM topology with a static simulated network latency of approximately 10.5 ms. This environment may not fully capture the dynamic jitter, packet loss, and fluctuating bandwidth typical of multi-region public cloud deployments or production-scale SD-WAN infrastructures. Second, the caching performance benchmarks primarily focused on an optimal state of cache hit ratios to isolate the core theoretical modeling. Consequently, the system response times under highly volatile workloads with frequent cache misses and real-time data mutations were not extensively analyzed. Future research should address these gaps by expanding the testing matrix to real-world multi-cloud deployments and investigating automated, adaptive cache invalidation strategies under unpredictable concurrent transaction rates.

4. Conclusion

Based on the test results, it can be concluded that while ORM middleware (such as Prisma or Eloquent) offers developers speed and ease of coding, their sequential and iterative (N+1 queries) query characteristics prove to be a significant architectural failure point when faced with physical network latency. In an ideal on-premises environment, direct execution to PostgreSQL recorded superior latency (0.04 ms). However, when stretched across a long-distance VPN tunnel with a default RTT of ~10.5 ms, the accumulated network latency for each synchronous relational operation triggered an extreme spike in response time of up to 52,400% (reaching 21.00 ms per query). These findings quantitatively confirm that relational databases are highly intolerant of high-latency distributed cloud networking environments.

As a solution to this degradation, implementing Redis as a caching layer proved to be a highly effective structural mitigation strategy, provided that an optimal cache hit ratio is maintained. According to the mathematical modeling proposed in this study, the efficacy of this layer relies heavily on data being successfully served directly from memory to bypass the encrypted network boundary. When this high cache hit condition is met, the presence of an in-memory database successfully eliminates disk I/O wait times and breaks the domino

effect of sequential ORM queries. Although Redis performance suffered a 5,868% decrease due to the demands of remote protocol synchronization, this layer was able to maintain the upper latency limit at 13.13 ms. Overall, under optimal caching conditions, this integration provides a 1.60x processing speedup compared to if the application were left directly exposed to a relational database.

Architecturally, this study proves a new formulation: the performance of modern distributed web applications is no longer limited by server computation speed, but rather is dictated solely by the number of queries generated by the framework multiplied by network encryption latency. Therefore, ORM-based applications distributed over encrypted networks (such as SD-WAN or VPN) absolutely must be accompanied by a reliable intermediate caching mechanism to neutralize the middleware load.

As a further development direction, this research can be expanded by exploring the use of an automatic cache turnover prediction mechanism powered by artificial intelligence (Machine Learning / AI-driven caching) to address the cache miss problem in fluctuating query workloads [1]. In addition, future experiments need to analyze more deeply the performance penalty comparison between Eager Loading and Lazy Loading methods on different ORMs, when tested in a production-scale multi-cloud topology.

References

[1] H. Muhamad and M. Taha, "Optimizing the performance of web applications in dynamic network environment: A systematic and comprehensive analytical survey," 3 2026.

[2] Z. Zhang, A. Megargel, and L. Jiang, "Performance evaluation of newsql databases in a distributed architecture," IEEE Access, vol. 13, pp. 11 185-11 194, 2025.

[3] S. Troia, L. Borgianni, G. Sguotti, S. Giordano, and G. Maier, "A comprehensive survey on software-defined wide area network," IEEE Communications Surveys and Tutorials, vol. 28, pp. 2805-2845, 2026.

[4] F. Freitas, A. Ferreira, and J. Cunha, "A methodology for refactoring orm-based monolithic web applications into microservices," Journal of Computer Languages, vol. 75, 6 2023.

[5] M. Pantelelis and C. Kalloniatis, "Mapping crud to events - towards an object to event-sourcing framework," in ACM International Conference Proceeding Series. Association for Computing Machinery, 11 2022, pp. 285-289.

[6] J. C. Yusmita, R. Arya, J. M. Wijaya, K. M. Suryaningrum, and R. R. Siswanto, "Optimizing database access strategy: A performance analysis comparison of raw sql and prisma orm," in Procedia Computer Science, vol. 269. Elsevier B.V., 2025, pp. 1201-1210.

[7] P. Krishnan, K. Jain, A. Aldweesh, P. Prabu, and R. Buyya, "Openstackdp: a scalable network security framework for sdn-based openstack cloud infrastructure," Journal of Cloud Computing, vol. 12, 12 2023.

[8] E. F. Kfoury, S. Choueiri, A. Mazloum, A. Alsabeh, J. Gomez, and J. Crichigno, "A compre-hensive survey on smartnics: Architectures, development models, applications, and research directions," IEEE Access, vol. 12, pp. 107 297-107 336, 2024.

[9] M. Ali, F. Naeem, G. Kaddoum, and E. Hossain, "Metaverse communications, networking, security, and applications: Research issues, state-of-the-art, and future directions," IEEE Communications Surveys and Tutorials, vol. 26, pp. 1238-1278, 2024.

[10] P. P. Ray, "A survey on model context protocol: Architecture, state-of-the-art, challenges and future directions," 4 2025. [Online]. Available: https://www.techrxiv.org/doi/full/10.36227/techrxiv.174495492.22752319/v1

[11] Dittrich, J. (2025). How to get Rid of SQL, Relational Algebra, the Relational Model, ERM, and ORMs in a Single Paper--A Thought Experiment. arXiv preprint arXiv:2504.12953.

[12] L. K. Ram, D. Saha, S. Chakraborty, and A. Gupta, "Beyond the handshake: Eliminating fallback latency from quic-to-tcp transitions," in Proceedings of the ACM CoNEXT Workshop. Association for Computing Machinery (ACM), 1 2026, pp. 132-137.

[13] S. Ferreira, J. Mendonca, B. Nogueira, W. Tiengo, and E. Andrade, "Benchmarking consistency levels of cloud-distributed nosql databases using ycsb," IEEE Access, vol. 13, pp. 63 428-63 438, 2025.

[14] W. Smith, Singlestore Database in Practice: The Complete Guide for Developers and Engineers. HiTeX Press, 2025.

[15] R. K. Veerapaneni, R. Delhibabu, A. Subbotin, and N. Zhukova, "Development of a high-performance in-memory database architecture for intelligent video surveillance in critical patient care," Frontiers in Digital Health, vol. 8, 5 2026. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fdgth.2026.1807507/full

[16] B. Bicski and A. Pekar, "Unveiling latency-induced service degradation: A methodological approach with dataset," IEEE Access, vol. 12, pp. 128 097-128 116, 2024.

[17] B. Romanous, S. Windh, I. Absalyamov, P. Budhkar, R. Halstead, W. Najjar, and V. Tsotras, "Efficient local locking for massively multithreaded in-memory hash-based operators," VLDB Journal, vol. 30, pp. 333-359, 5 2021.

[18] Y. Zhu, T. Xia, T. Zhu, Z. Zhao, K. Li, and X. Hu, "Rapo: An automated performance optimization tool for redis clusters in distributed storage metadata management," IEEE Access, vol. 13, pp. 58 060-58 074, 2025.

[19] S. K. Shivakumar and S. Sethii, DXP Performance Optimization. Apress, 2019, pp. 235-259. [Online]. Available: https://doi.org/10.1007/978-1-4842-4303-9_9

[20] Þórarinsson, S., Jónsson, B. Þ., & Khan, O. S. (2026, June). Optimization of Long-Running Media Aggregation Queries. In Proceedings of the 2026 International Conference on Multimedia Retrieval (pp. 1778-1786).Available: https://dl.acm.org/doi/10.1145/3805622.3810667

[21] D. Satriani, E. Veltri, D. Santoro, S. Rosato, S. Varriale, and P. Papotti, "Logical and physical optimizations for sql query execution over large language models," Proceedings of the ACM on Management of Data, vol. 3, pp. 1-28, 6 2025.

[22] Güvercin, A. E., & Avenoglu, B. (2022). Performance Analysis of Object-Relational Mapping (ORM) Tools in. Net 6 Environment. Bilişim Teknolojileri Dergisi, 15 (4), 453-465.

[23] J. Dittrich, "How to get rid of sql, relational algebra, the relational model, erm, and orms in a single paper - a thought experiment," 4 2025. [Online]. Available: http://arxiv.org/abs/2504.12953

[24] W. Liu and T. H. Chen, "Slocator: Localizing the origin of sql queries in database-backed web applications," IEEE Transactions on Software Engineering, vol. 49, pp. 3376-3390, 6 2023.

[25] M. R. Hasan, M. R. Hasan, and H. Bagheri, "Unlocking optimal orm database designs: Acceler-ated tradeoff analysis with transformers," Proceedings of the ACM on Software Engineering, vol. 2, pp. 1639-1662, 6 2025.

[26] E. Sierra and U. L. Yuhana, "Optimizing software development through data access speed using object-relational mapping (orm) on credit risk application," in Proceedings of the 3rd 2023 International Conference on Smart Cities, Automation and Intelligent Computing Systems, ICON-SONICS 2023. Institute of Electrical and Electronics Engineers Inc., 2023, pp. 201-206.

[27] A. Vikiru, M. Muiruri, and I. Ateya, "An overview on cloud distributed databases for business environments," 1 2023. [Online]. Available: http://arxiv.org/abs/2301.10673

[28] L. Rosa, L. Foschini, and A. Corradi, "Empowering cloud computing with network acceleration: A survey," IEEE Communications Surveys and Tutorials, vol. 26, pp. 2729-2768, 2024.

[29] K. S. Shim, B. Greskamp, B. Towles, B. Edwards, J. P. Grossman, and D. E. Shaw, "The specialized high-performance network on anton 3," in Proceedings - International Symposium on High-Performance Computer Architecture, vol. 2022-April. IEEE Computer Society, 2022, pp. 1211-1223.

[30] Saugata, G.-L. Juan, A. R. M. Onur, and Ghose, A Modern Primer on Processing in Memory. Springer Nature Singapore, 2023, pp. 171-243. [Online]. Available: https://doi.org/10.1007/978-981-16-7487-7_7

[31] R. Wang, J. Wang, S. Idreos, M. T. √ñzsu, and W. G. Aref, "The case for distributed shared-memory databases with rdma-enabled memory disaggregation," arXiv preprint arXiv:2207.03027, 2022.

[32] K. Huang, T. Wang, Q. Zhou, and Q. Meng, "The art of latency hiding in modern database engines," in Proceedings of the VLDB Endowment, vol. 17. VLDB Endowment, 2023, pp. 577-590.

[33] Chango, W., Salguero, A., Ortíz, L., & Chavarrea, R. (2025, June). Comparative Performance of MongoDB and Redis in Microservices-Based Web Applications: NoSQL Databases Selection. In 2025 IEEE Technology and Engineering Management Society (TEMSCON LATAM) (pp. 1-6). IEEE.

[34] Y. Xu, P. He, X. Zhang, and H. Hu, "Research and implementation of redis-based data hot-table caching mode," in 2025 4th International Symposium on Computer Applications and Information Technology, ISCAIT 2025. Institute of Electrical and Electronics Engineers Inc., 2025, pp. 2172-2175.

[35] Y. Tang, P. Li, and Y. Cao, "High-concurrency data service model based on mysql, redis, and mycat," in 2025 10th International Conference on Electronic Technology and Information Science, ICETIS 2025. Institute of Electrical and Electronics Engineers Inc., 2025, pp. 412-415.

[36] D. Almeida, M. Lopes, L. Saraiva, M. Abbasi, P. Martins, J. Silva, and P. V√°z, "Performance comparison of redis, memcached, mysql, and postgresql: A study on key-value and relational databases," in 2023 2nd International Conference on Smart Technologies for Smart Nation, SmartTechCon 2023. Institute of Electrical and Electronics Engineers Inc., 2023, pp. 902-907.

[37] A. Tezuysal and I. Ahmed, Database Design and Modeling with PostgreSQL and MySQL: Build efficient and scalable databases for modern applications using open source databases. Packt Publishing Ltd, 2024.