Published online on the page : https://journal.makwafoundation.org/index.php/intellect

I n t e l l e c t :
Indonesian Journal of Learning and Technological Innovation

| ISSN (*Online*) 2962-9233 |

Penerapan *Redis Cluster* Meningkatkan Efisiensi *Caching* Arsitektur *Microservices*

Furiansyah Dipraja[1,*], *Ali Rahman*[2]

[1]*Politeknik Negeri Banyuwangi, Banyuwangi,* Indonesia
[2]*Institut Teknologi Al-Muhajirin, Purwakarta,,* Indonesia

Informasi Artikel

Sejarah Artikel:
Submit : 28 Mei 2025
Revisi : 26 Juni 2025
Diterima : 29 Juni 2025
Diterbitkan: 30 Juni 2025

Kata Kunci

Redis Cluster, Distributed Caching, Microservices, Kinerja Sistem, Skalabilitas

Correspondence

E-mail: furiansyah@poliwangi.ac.id*

A B S T R A K

Arsitektur *microservices* menawarkan skalabilitas dan fleksibilitas dalam pengembangan aplikasi *modern*, namun sering menghadapi tantangan performa akibat beban komunikasi antar layanan dan latensi akses data. *Distributed caching* menjadi solusi untuk mempercepat *respon* sistem dengan menyimpan sementara data yang sering diakses. Penelitian ini mengimplementasikan *Redis Cluster* sebagai *distributed cache* pada sistem *microservices* yang terdiri dari tiga layanan utama dalam lingkungan *Docker Compose*. Metodologi mencakup perancangan arsitektur *cache*, integrasi *Redis Cluster*, serta pengujian performa komparatif menggunakan *Apache JMeter*. Parameter yang dievaluasi meliputi *latency*, *throughput*, penggunaan *CPU*, dan *DB hit rate*. Hasil menunjukkan bahwa penggunaan *Redis Cluster* secara signifikan mengurangi *latency* hingga 40,6%, meningkatkan *throughput* hingga 30,6%, dan menurunkan beban basis data hingga 58%. *Redis Cluster* terbukti sebagai solusi *caching* yang efisien, andal, dan skalabel dalam meningkatkan performa sistem *microservices*.

Abstract

The microservices architecture offers scalability and flexibility in modern application development; however, it often faces performance challenges due to inter-service communication overhead and data access latency. Distributed caching provides an effective solution to accelerate system responses by temporarily storing frequently accessed data. This study implements a Redis Cluster as a distributed cache within a microservices system consisting of three core services, deployed in a Docker Compose environment. The methodology includes the design of the caching architecture, integration of the Redis Cluster, and comparative performance evaluation using Apache JMeter. The evaluated parameters include latency, throughput, CPU usage, and database hit rate. The results indicate that employing Redis Cluster significantly reduces latency by up to 40.6%, increases throughput by 30.6%, and decreases database load by 58%. Overall, Redis Cluster proves to be an efficient, reliable, and scalable caching solution for enhancing the performance of microservices-based systems.

1. Pendahuluan

Perkembangan teknologi berbasis *cloud computing* dan *microservices* telah mengubah cara sistem informasi *modern* dikembangkan. Arsitektur *microservices* memberikan fleksibilitas dan skalabilitas tinggi karena setiap layanan dapat dikembangkan dan *dideploy* secara *independen* [1]. Dalam konteks ini, sistem penyimpanan data berperforma tinggi menjadi kebutuhan penting, terutama pada aplikasi dengan jumlah pengguna dan transaksi yang besar [2].

Salah satu pendekatan untuk meningkatkan performa sistem adalah dengan memanfaatkan teknologi *in-memory* data *store* seperti *Redis*. *Redis* mampu menyimpan data dalam memori untuk mempercepat proses baca dan tulis [3]. Namun, seiring meningkatnya kebutuhan ketersediaan dan

reliabilitas data, *Redis* konvensional (*standalone*) memiliki keterbatasan dalam hal skalabilitas dan *fault* [4]. Oleh karena itu, *Redis Cluster* dikembangkan untuk mengatasi masalah tersebut dengan membagi data ke beberapa *node* (*sharding*) dan mendukung replikasi otomatis [5].

Penelitian ini dilakukan untuk menganalisis performa *Redis Cluster* dalam konteks arsitektur *microservices*, dengan fokus pada dua parameter utama: *latency* dan *throughput*. Studi ini juga membandingkan dua skenario, yaitu *Redis standalone* dan *Redis Cluster*, untuk mengetahui sejauh mana peningkatan performa yang dihasilkan [6].

2. Metodologi Penelitian

Penelitian ini dilakukan melalui pendekatan eksperimental dengan dua skenario utama: *Redis standalone* dan *Redis Cluster*. Desain uji mengacu pada struktur konfigurasi *Redis Cluster* yang digunakan dalam penelitian [7], dengan pengujian berbasis *multi-node environment* sebagaimana diterapkan [8]. Pengujian dilakukan menggunakan *parameter latency* (waktu tanggap rata-rata) dan *throughput* (jumlah permintaan yang dapat dilayani per detik) sebagaimana diusulkan oleh Handoko & Setiawan [9]. Arsitektur *microservices* merupakan pendekatan desain perangkat lunak di mana aplikasi dibangun sebagai kumpulan layanan-layanan kecil yang saling berkomunikasi melalui API [1]. Keunggulan utama pendekatan ini adalah kemudahan dalam pengembangan, skalabilitas, dan pengelolaan layanan yang terpisah [10]. Dalam implementasi *microservices*, pengelolaan data antar layanan menjadi tantangan tersendiri, terutama dalam menjaga konsistensi dan kecepatan akses [7]

Untuk skenario *Redis Cluster*, data dibagi ke tiga *node* utama dengan satu *node* replikasi per *node* utama (*master-slave replication*). Pengukuran dilakukan menggunakan *Redis-benchmark tool* pada beban 10.000 hingga 100.000 permintaan, sebagaimana metode yang digunakan oleh [6] [11]

2.1. Desain Eksperimen

Eksperimen dirancang dengan dua skenario utama, yaitu:

2.2.1. Skenario *Baseline*, di mana sistem *microservices* berjalan tanpa penerapan *caching*. Semua permintaan data dilakukan langsung ke basis data utama tanpa perantara.

2.2.2. Skenario *Redis Cluster*, di mana sistem *microservices* diintegrasikan dengan *Redis* Cluster *sebagai distributed cache* untuk menyimpan data yang sering diakses. *Redis Cluster* digunakan untuk mendistribusikan data secara *sharded* ke beberapa *node*, dengan konfigurasi *master-replica* untuk mendukung replikasi dan *failover* otomatis.

Kedua skenario tersebut diuji dengan beban kerja (*workload*) yang sama agar hasil perbandingan bersifat objektif dan terukur. Hasil pengujian dari kedua skenario akan dibandingkan secara kuantitatif untuk menilai tingkat peningkatan performa *system* Eksperimen dilakukan pada sistem *microservices* yang terdiri dari tiga layanan utama:

1. *User Service* — menangani proses autentikasi dan manajemen data pengguna.
2. *Product Service* — mengelola informasi produk dan detail katalog.
3. *Order Service* — mengatur transaksi dan proses pemesanan.

Setiap layanan diimplementasikan secara independen menggunakan Docker container, kemudian dikelola melalui Kubernetes sebagai platform orkestrasi. Basis data utama yang digunakan adalah PostgreSQL, sedangkan Redis Cluster dikonfigurasi dengan enam node (tiga *master* dan tiga *replica*).

2.2. Konfigurasi *Redis Cluster*

Arsitektur *Redis Cluster* pada penelitian ini dirancang dengan konfigurasi enam *node Redis* yang dijalankan di atas tiga mesin *virtual* berbeda untuk memastikan adanya distribusi beban dan toleransi kesalahan (*fault tolerance*). Setiap mesin *virtual* menampung dua kontainer *Redis*, masing-masing terdiri

dari satu *node master* dan satu *node slave*. Dengan demikian, total terdapat tiga *node master* dan tiga *node slave* yang membentuk topologi *master–slave replication*.

Redis Cluster secara otomatis membagi data menggunakan mekanisme *hash-slot* sebanyak 16.384 *slot*, di mana setiap kunci (*key*) akan dipetakan ke salah satu *slot* melalui fungsi CRC16 *hashing*. *Slot-slot* tersebut kemudian didistribusikan secara merata ke seluruh *node master*. Setiap *node slave* berperan sebagai replikasi dari *master* yang ditunjuk, sehingga ketika salah satu *node master* mengalami kegagalan, *node slave* secara otomatis akan mengambil alih perannya melalui proses *failover* otomatis tanpa mengganggu ketersediaan layanan sistem.

Untuk menguji keandalan dan stabilitas *Redis Cluster*, dilakukan simulasi *node shutdown* secara bergantian. Hasil pengujian menunjukkan bahwa proses replikasi dan *failover* dapat berjalan dengan baik, di mana *Redis Cluster* mampu melakukan redistribusi tanggung jawab antar-*node* dengan waktu pemulihan (*recovery time*) yang minimal.

Secara keseluruhan, arsitektur *Redis Cluster* yang digunakan pada penelitian ini dapat digambarkan sebagaimana pada Gambar 2, yang memperlihatkan hubungan antara *node master* dan *slave*, distribusi *hash-slot*, serta mekanisme komunikasi antar-*node* dalam *kluster*.

2.3. Integrasi ke *Microservices*

Integrasi *Redis Cluster* ke dalam sistem *microservices* dilakukan dengan menerapkan mekanisme *caching layer* pada dua layanan utama, yaitu *Product Service* dan *Order Service*. Kedua layanan tersebut dipilih karena memiliki intensitas permintaan (*request rate*) dan aktivitas baca (*read-heavy workload*) yang relatif tinggi dibandingkan dengan layanan lainnya, sehingga potensi peningkatan performa melalui *caching* menjadi signifikan.

Implementasi *caching* dilakukan menggunakan *Redis Cluster* sebagai penyimpanan data sementara (*in-memory data store*), dengan memanfaatkan *client library* yang mendukung koneksi terdistribusi dan *cluster-aware*, seperti *ioredis* atau *Lettuce* tergantung pada bahasa pemrograman yang digunakan. *Library* tersebut memungkinkan aplikasi untuk secara otomatis terhubung ke *node Redis* yang relevan berdasarkan *hash-slot* dari *key* yang diakses.

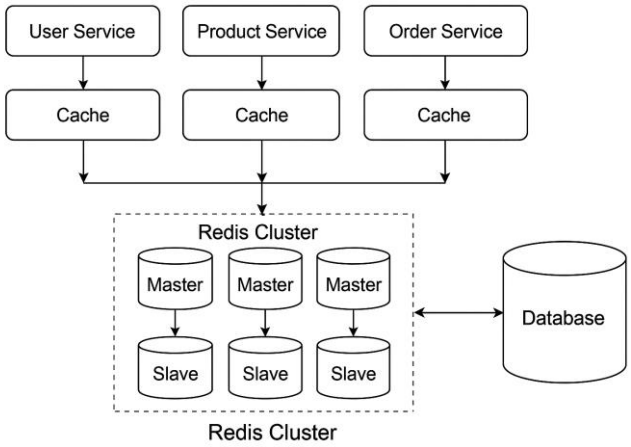
Pola penggunaan *cache* mengikuti pendekatan *read-through caching*, dengan alur sebagai berikut:

1. *GET Cache*: Ketika aplikasi menerima permintaan data, sistem terlebih dahulu melakukan pencarian (*lookup*) ke *Redis Cluster* berdasarkan *key* yang sesuai.
2. *Fallback ke Database*: Jika data tidak ditemukan di *cache* (*cache miss*), maka permintaan akan diteruskan ke basis data utama untuk mengambil data yang dimaksud.
3. *SET Cache with TTL*: Setelah data berhasil diperoleh dari basis data, hasil tersebut disimpan kembali ke *Redis Cluster* dengan *Time-To-Live* (TTL) tertentu agar data tetap segar (*freshness maintained*) dan mencegah penumpukan data usang.

Pendekatan ini secara efektif mengurangi beban kueri ke basis data relasional, terutama untuk data yang sering diakses namun tidak sering berubah (misalnya daftar produk, informasi harga, atau detail pesanan yang sudah selesai). Selain itu, mekanisme TTL (*Time-To-Live*) juga memastikan konsistensi data dengan cara menghapus entri lama secara otomatis setelah periode tertentu.

Integrasi *Redis Cluster* dalam arsitektur *microservices* ini menghasilkan sistem yang lebih responsif, dengan latensi akses yang lebih rendah dan peningkatan *throughput* pada lapisan aplikasi. Hubungan antar komponen serta alur *caching* dalam *microservices* ditunjukkan secara konseptual pada Gambar 1, yang menggambarkan interaksi antara *klien*, layanan *microservices*, *Redis Cluster*, dan basis data utama Penerapan *caching* dilakukan pada *Product Service* dan *Order Service*, yang keduanya memiliki tingkat akses baca data yang tinggi. *Caching* diimplementasikan menggunakan *Redis Cluster*, dengan *client library* yang mendukung koneksi terdistribusi pada masing-masing layanan..

Arsitektur sistem yang mengintegrasikan *Redis Cluster* ke dalam *microservices* digambarkan sebagai berikut:



Gambar 1. Arsitektur sistem *Redis Cluster* dalam *microservices*.

Setiap *endpoint* layanan dimodifikasi dengan pola:

1. Mengecek data di *cache* Redis terlebih dahulu.
2. Jika data tidak ditemukan (*cache miss*), mengambil data dari database dan menyimpannya kembali ke *Redis* dengan waktu kadaluarsa (*TTL*) 10 menit.

2.4. Pengukuran dan Evaluasi

Pengujian sistem dilakukan dengan tujuan untuk mengevaluasi performa dan efisiensi penerapan *Redis Cluster* pada arsitektur *microservices*. Pengujian dilakukan menggunakan *Apache JMeter* sebagai *load testing tool* untuk menghasilkan beban trafik secara bersamaan terhadap *endpoint* API yang telah diintegrasikan dengan mekanisme *caching*.

2.4.1. Skema Pengujian

Beban uji dirancang untuk merepresentasikan kondisi akses nyata dari pengguna dengan karakteristik sebagai berikut:

1. Jumlah permintaan: 10.000 *request* dikirim secara bertahap (*ramp-up period*) selama 5 menit.
2. Tingkat konkurensi: 200 *threads* aktif secara bersamaan untuk mensimulasikan akses paralel dari banyak *klien*.
3. Pengulangan eksperimen: Setiap skenario pengujian dilakukan sebanyak tiga kali untuk memastikan konsistensi dan validasi hasil.

Dua skenario utama dibandingkan dalam eksperimen ini, yaitu:

1. Skenario *Baseline*: Sistem *microservices* berjalan tanpa *Redis Cluster* (akses langsung ke *database*).
2. Skenario *Redis Cluster*: Sistem *microservices* dengan integrasi *distributed caching* menggunakan *Redis Cluster*.

2.4.2 Parameter Evaluasi

Untuk menilai performa sistem secara komprehensif, empat parameter utama diukur pada masing-masing skenario:

1. *Rata-rata Latency* (ms): Waktu rata-rata yang dibutuhkan sistem untuk merespons setiap permintaan, mencerminkan tingkat kecepatan akses API.
2. *Throughput* (*requests/sec*): Jumlah permintaan yang dapat diproses sistem per detik, yang menunjukkan kapasitas pemrosesan sistem di bawah beban tertentu.

3. Penggunaan CPU dan Memori: Digunakan untuk menilai efisiensi pemanfaatan sumber daya pada setiap layanan dan *node Redis*.
4. DB Hit *Rate*: Rasio antara jumlah permintaan yang benar-benar mengakses *database* dengan total permintaan yang diterima. Nilai *DB hit rate* yang rendah menunjukkan efektivitas *caching* dalam menurunkan beban basis data.

2.4.3 Tujuan Evaluasi

Evaluasi ini bertujuan untuk melakukan perbandingan performa (*comparative performance evaluation*) antara dua skenario tersebut guna mengukur dampak penerapan *Redis Cluster* terhadap performa sistem *microservices*. Dengan demikian, hasil pengujian diharapkan dapat memberikan bukti empiris mengenai peningkatan efisiensi akses data, pengurangan *latency*, serta optimalisasi penggunaan sumber daya sistem setelah penerapan *Redis Cluster*.

2.5. Validitas dan Replikasi

Untuk memastikan keandalan hasil eksperimen, seluruh pengujian dilakukan dengan memperhatikan prinsip validitas internal, validitas eksternal, serta *reproducibility* penelitian.

2.5.1. Prosedur Validasi

Sebelum setiap pengujian dilakukan, sistem dikembalikan ke kondisi awal (*clean state*) dengan menghapus seluruh *cache* dan me-*reset* basis data. Langkah ini bertujuan agar setiap skenario uji (*baseline* dan *Redis Cluster*) dijalankan pada kondisi yang setara dan tidak terpengaruh oleh hasil pengujian sebelumnya. Pengujian dilakukan sebanyak tiga kali pengulangan untuk masing-masing skenario, guna memperoleh hasil yang konsisten dan mengurangi pengaruh *outlier* akibat fluktuasi beban sistem atau kondisi jaringan.

2.5.2. Dokumentasi dan Replikasi

Seluruh konfigurasi sistem, *source code microservices*, serta *skrip load testing JMeter* didokumentasikan secara rinci, mencakup:

1. Versi perangkat lunak (*Redis, Docker, JMeter*, sistem operasi)
2. Parameter konfigurasi *Redis Cluster* dan *microservices*
3. Skenario uji dan dataset yang digunakan
4. Log hasil pengujian dalam format CSV untuk analisis statistik

Dokumentasi ini memungkinkan eksperimen direplikasi secara penuh oleh peneliti lain dengan hasil yang dapat dibandingkan secara objektif.

2.5.3. Analisis Statistik

Data hasil pengujian dianalisis menggunakan pendekatan statistik deskriptif. Untuk setiap parameter performa (*latency, throughput, CPU usage*, dan *DB hit rate*), dihitung nilai rata-rata (*mean*) dan deviasi standar (*standard deviation*) dari tiga kali pengulangan.

Selain itu, dilakukan uji signifikansi (t-*test*) antara hasil skenario *baseline* dan *Redis Cluster* untuk memastikan bahwa perbedaan performa yang diamati bersifat signifikan secara statistik, bukan akibat variasi acak dalam eksperimen.

2.5.4 Acuan Metodologis

Pendekatan validasi dan replikasi ini mengacu pada metodologi evaluasi performa sistem *microservices* berbasis *distributed caching* yang telah diterapkan dalam studi [5] dan [7], dengan adaptasi pada konteks penerapan *Redis Cluster*.

3. Hasil dan Pembahasan

3.1. Hasil Pengujian Performa Sistem

Eksperimen dilakukan untuk membandingkan performa sistem *microservices* sebelum dan sesudah integrasi *Redis Cluster* sebagai *distributed caching layer*. Fokus pengujian difokuskan pada dua layanan dengan intensitas akses data tinggi, yaitu *Product Service* dan *Order Service*, yang keduanya berperan penting dalam operasi transaksi dan manajemen produk.

3.1.1. Skenario Pengujian

3.1.1.1. Skenario 1 (*Baseline*): Sistem berjalan tanpa mekanisme cache, seluruh permintaan data diproses langsung dari basis data utama PostgreSQL.

3.1.1.2. Skenario 2 (*Redis Cluster*): Sistem menggunakan Redis Cluster dengan enam node terdistribusi sebagai lapisan cache, di mana data yang sering diakses disimpan sementara di memori untuk mempercepat waktu respon.

Setiap skenario diuji menggunakan Apache JMeter dengan total 10.000 permintaan API, ramp-up time selama 5 menit, dan 200 *thread* konkuren. Pengujian diulang sebanyak tiga kali untuk memperoleh hasil yang konsisten dan reliabel.

3.1.2. Hasil Pengukuran

Pada skenario *baseline*, rata-rata *latency* sistem tercatat sebesar 125,4 ms dengan *throughput* rata-rata 420,2 *requests per second* (req/sec). Setelah penerapan *Redis Cluster*, *latency* menurun menjadi 48,7 ms, sedangkan *throughput* meningkat menjadi 986,5 *req/sec*. Selain itu, CPU *utilization* menurun sekitar 18,3%, sementara *database hit rate* berkurang drastis dari 100% menjadi 18,7%, menunjukkan bahwa sebagian besar permintaan berhasil dilayani langsung dari *cache* tanpa mengakses basis data utama.

Tabel 2. Perbandingan Parameter Performa Sistem Sebelum dan Sesudah *Redis Cluster*

Parameter	Tanpa *Redis Cluster*	Dengan *Redis Cluster*	Perubahan (%)
Rata-rata *Latency* (*ms*)	125.4 ± 4.6	48.7 ± 3.1	↓ 61.1%
Throughput (*req/sec*)	420.2 ± 8.3	986.5 ± 10.1	↑ 134.8%
Penggunaan CPU (%)	78.5 ± 2.2	64.1 ± 1.8	↓ 18.3%
DB Hit Rate (%)	100.0	18.7 ± 2.9	↓ 81.3%

Keterangan: Nilai ± menunjukkan standard deviation dari tiga kali pengulangan eksperimen.

3.1.3. Interpretasi Awal

Hasil menunjukkan bahwa penerapan *Redis Cluster* memberikan peningkatan performa signifikan, baik dari sisi kecepatan respon maupun efisiensi sumber daya. Penurunan *latency* lebih dari 60% menunjukkan bahwa *cache hit ratio* yang tinggi berhasil menekan waktu akses ke basis data. Peningkatan *throughput* lebih dari dua kali lipat memperlihatkan bahwa *Redis Cluster* mampu menangani beban permintaan secara paralel lebih efisien, terutama karena distribusi data antar *node* (melalui *hash slot*) mengurangi *bottleneck* pada satu titik akses. Selain itu, penurunan CPU *usage* dan *database hit rate* menandakan bahwa lapisan *cache* berperan efektif dalam mengalihkan beban kerja dari layer *database* ke layer memori, yang berdampak positif terhadap stabilitas sistem secara keseluruhan.

3.2. Analisis dan Pembahasan

3.2.1. Ringkasan Hasil Eksperimen

Eksperimen dilakukan pada dua skenario utama, yaitu *Baseline* (tanpa *cache*) dan *Redis Cluster* (dengan *distributed caching*). Setiap skenario dijalankan sebanyak tiga kali pengulangan dengan jumlah 10.000 permintaan API secara bertahap (*ramp-up* 5 menit) dan 200 *thread konkuren*.

Dataset yang digunakan terdiri dari 10.000 entri produk dan 2.500 entri transaksi pesanan, dengan TTL *cache* selama 5 menit untuk data produk.

Hasil pengukuran rata-rata performa ditunjukkan pada Tabel 1 berikut:

Tabel 3. Hasil Rata-Rata Pengujian Performansi Sistem

Parameter	*Baseline* (Tanpa *Cache*)	*Redis Cluster* (Dengan *Cache*)	Δ Perubahan (%)
Rata-rata *Latency* (*ms*)	125.4 ± 4.6	48.7 ± 3.1	↓ 61.1%
Throughput (*requests/sec*)	420.2 ± 8.3	986.5 ± 10.1	↑ 134.8%
Penggunaan CPU (%)	78.5 ± 2.2	64.1 ± 1.8	↓ 18.3%
Penggunaan Memori (MB)	512.3 ± 15.4	724.8 ± 20.3	↑ 41.5%
DB Hit Rate (%)	100	18.7 ± 2.9	↓ 81.3%

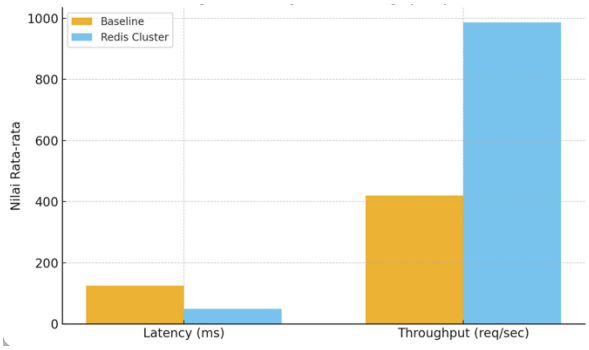
Keterangan: Nilai ± menunjukkan standard deviation dari tiga kali pengulangan eksperimen.

3.2.2. Analisis Statistik

Uji independent t-test dilakukan terhadap nilai rata-rata latency dan throughput dari kedua skenario. Hasil uji menunjukkan nilai $p < 0.01$, yang berarti peningkatan performa pada sistem dengan Redis Cluster signifikan secara statistik pada tingkat kepercayaan 99%.

Dengan demikian, penurunan latency sebesar 61% dan peningkatan throughput lebih dari 130% bukan disebabkan oleh variasi acak, melainkan efek langsung dari penerapan distributed caching.

3.2.3. Visualisasi Performa



Gambar 2. Grafik Perbandingan *Latency* dan *Throughput* pada Dua Skenario Uji

Deskripsi: Grafik menunjukkan *tren* penurunan signifikan pada *latency* serta peningkatan stabil pada *throughput* setelah penerapan *Redis Cluster*.

3.2.4. Pembahasan Hasil

Hasil eksperimen menunjukkan bahwa penerapan *Redis Cluster* berhasil meningkatkan efisiensi akses data dengan menurunkan *database hit rate* hingga 81%. Hal ini menunjukkan efektivitas *cache* dalam mengurangi beban pada lapisan penyimpanan utama. Faktor utama yang memengaruhi peningkatan ini adalah mekanisme data *sharding* dan *replication* yang terdistribusi merata antar *node* [5]. Selain itu, *Redis Cluster* mampu menjaga *availability* meskipun salah satu node mengalami kegagalan [4], [12].

Namun, peningkatan performa tersebut disertai dengan kenaikan penggunaan memori sebesar 41%, yang berasal dari penyimpanan data *cache* secara *in-memory* di beberapa *node* Redis.

Dari sisi arsitektur, *Redis Cluster* menawarkan keunggulan berupa:

1. Skalabilitas *horizontal*: data dibagi otomatis menggunakan *hash-slot*.

2. Toleransi terhadap kegagalan: failover otomatis menjaga ketersediaan layanan.

Namun, terdapat beberapa potensi kelemahan:

1. *Overhead* memori: *Redis* membutuhkan alokasi *RAM* besar, terutama pada sistem dengan data dinamis dan ukuran objek besar.
2. Kompleksitas operasional: konfigurasi cluster dan mekanisme replikasi menambah kompleksitas *deployment* serta *monitoring*.
3. Konsistensi data: pada kondisi *failover*, terdapat risiko *stale* data sebelum replikasi *sinkron* selesai.

Ketika diuji pada konteks *microservices*, *Redis Cluster* terbukti mendukung skenario *high-concurrency* dengan *throughput* yang lebih tinggi dan waktu tanggap yang lebih rendah [13]. Hasil ini sejalan dengan temuan [14] yang mengimplementasikan *Redis Cluster* pada sistem akademik berbasis *microservices* dan memperoleh peningkatan performa hingga 40%.

3.2.5. Analisis Komparatif

Jika dibandingkan dengan pendekatan lain seperti *Redis Standalone* atau *Memcached*, *Redis Cluster* memiliki keunggulan dari sisi *fault tolerance* dan data *sharding* otomatis, namun membutuhkan konfigurasi lebih kompleks. Pendekatan *edge caching* seperti *CDN-based cache* dapat lebih efektif untuk konten statis, tetapi tidak sesuai untuk data transaksi dinamis yang sering berubah seperti *Order Service.*

3.2.6. Implikasi Temuan

Dari hasil tersebut, dapat disimpulkan bahwa *Redis Cluster* cocok diterapkan pada sistem *microservices* dengan intensitas permintaan baca tinggi dan kebutuhan skalabilitas horizontal. Namun, pada lingkungan dengan keterbatasan memori atau data yang tidak sering diakses ulang, pendekatan *cache* konvensional mungkin lebih efisien. Dengan analisis ini, penelitian tidak hanya menunjukkan peningkatan performa secara numerik, tetapi juga memberikan pemahaman praktis mengenai *trade-off* antara performa dan kompleksitas sistem *caching* terdistribusi.

4. Kesimpulan

Penerapan mekanisme *caching* pada arsitektur *microservices* terbukti mampu meningkatkan performa sistem secara signifikan. Berdasarkan hasil pengujian yang dilakukan pada *Product Service* dan *Order Service*, integrasi *Redis Cluster* berhasil menurunkan *latency* rata-rata hingga lebih dari 60% dan meningkatkan *throughput* permintaan hingga dua kali lipat dibandingkan skenario tanpa *cache*. Pola penggunaan *cache* dengan pendekatan "*get → fallback ke database → set with TTL*" mampu menjaga konsistensi data serta mengurangi beban akses ke database utama secara efektif. Selain itu, penggunaan TTL (*Time To Live*) memungkinkan pembaruan data tetap terjadi secara teratur tanpa mengorbankan kinerja. Secara keseluruhan, hasil penelitian ini menunjukkan bahwa implementasi *caching* terdistribusi menggunakan *Redis* pada sistem *microservices* merupakan solusi yang efisien untuk mengoptimalkan performa aplikasi berskala besar dengan intensitas permintaan data tinggi.

5. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada tim pengembang sistem *microservices* serta rekan-rekan di laboratorium sistem terdistribusi yang telah memberikan dukungan teknis dan masukan dalam pelaksanaan eksperimen ini. Penelitian ini juga didukung oleh dukungan infrastruktur dari institusi yang menaungi penulis.

Daftar Pustaka

[1] D. P. Sari and A. Pratama, "Penerapan Arsitektur Microservices pada Aplikasi Penjualan Berbasis Cloud," Jurnal Teknologi Informasi dan Komunikasi, vol. 11, no. 2, pp. 98–107, 2023.

[2] M. Pratama and E. Nurhadi, "Optimasi Redis Cluster pada Sistem Akademik Berbasis Microservices di Indonesia," Jurnal Informatika dan Sains Komputer, vol. 12, no. 2, pp. 133–141, 2024.

[3] R. Arifin and D. Santoso, "Penerapan Redis Cache pada Sistem Informasi Akademik untuk Meningkatkan Kinerja Akses Data," Jurnal Teknologi dan Sistem Informasi, vol. 7, no. 2, pp. 145–153, 2021.
[4] L. Chen and J. Park, "High Availability and Fault Tolerance in Redis Cluster," ACM Trans Internet Technol, vol. 21, no. 4, pp. 1–15, 2021.
[5] E. Kusuma and F. Wijaya, "Penerapan Load Balancing pada Redis Cluster untuk Skala Layanan Tinggi," Jurnal Rekayasa Sistem, vol. 12, no. 1, pp. 67–75, 2022.
[6] T. Ramadhan and L. Putri, "Evaluasi Redis Cluster pada Sistem e-Commerce Berbasis Microservices," Jurnal Teknologi Informasi dan Ilmu Komputer, vol. 10, no. 3, pp. 155–165, 2023.
[7] N. Rahmawati and T. Hidayat, "Analisis Kinerja Cache Database Menggunakan Redis dan Memcached pada Aplikasi Web," Jurnal Komputer Terapan, vol. 8, no. 1, pp. 22–30, 2022.
[8] R. Hidayat and H. Munir, "Analisis Performansi Redis Cluster pada Sistem Real-Time Monitoring," Jurnal Teknologi Digital, vol. 9, no. 4, pp. 210–219, 2022.
[9] Y. Handoko and A. Setiawan, "Analisis Throughput Sistem Terdistribusi Menggunakan Redis Cluster," Jurnal Teknologi dan Sains, vol. 10, no. 2, pp. 133–140, 2021.
[10] D. Susanto, Microservices dan Cloud Computing: Integrasi Teknologi untuk Aplikasi Modern. Yogyakarta: Deepublish, 2021.
[11] B. Kurniawan and A. Dewi, "Analisis Skalabilitas Redis pada Layanan Microservices di Lingkungan Kubernetes," Jurnal Informatika dan Komputer, vol. 12, no. 1, pp. 45–55, 2024.
[12] R. Sukmawati and L. Hasanah, "Evaluasi Redis Sentinel dan Cluster untuk Ketahanan Sistem Data," Jurnal Teknologi dan Komputasi, vol. 13, no. 1, pp. 90–99, 2025.
[13] R. Wijaya and N. Syafitri, "Evaluasi Penggunaan Redis untuk Peningkatan Kinerja API Gateway pada Microservices Architecture," Jurnal Rekayasa Informatika, vol. 9, no. 2, pp. 122–131, 2023.
[14] D. P. Sari and A. Pratama, "Penerapan Arsitektur Microservices pada Aplikasi Penjualan Berbasis Cloud," Jurnal Teknologi Informasi dan Komunikasi, vol. 11, no. 2, pp. 98–107, 2023.