Implementasi Strategi Default Connection Pool dan Prewarm Connection Pool pada Integrasi Minio Menggunakan Go dan K6

Muhammad Miftakul Salam[1], Ronggo Alit[2]

[1,2] Program Studi S1 Teknik Informatika, Universitas Negeri Surabaya
[1]muhammadmiftakul.22101@mhs.unesa.ac.id
[2]ronggoalit@unesa.ac.id

Abstrak— **Pengelolaan koneksi antara aplikasi *backend* dan *object storage* sering kali menghadapi tantangan efisiensi, terutama terkait latensi awal dan penggunaan sumber daya. Strategi *default connection pool* yang bersifat *lazy initialization* cenderung memicu *cold start penalty* yang tinggi, sedangkan strategi *prewarm* menawarkan kesiapan koneksi namun membutuhkan alokasi memori awal. Penelitian ini bertujuan untuk membandingkan kinerja kedua strategi tersebut dalam integrasi MinIO menggunakan bahasa pemrograman Go guna menentukan solusi paling optimal. Penelitian menggunakan metode eksperimental dengan melakukan *performance testing* menggunakan alat uji K6 melalui empat skenario beban kerja: *cold start burst*, *gradual ramp-up*, *spike test*, dan *sustained high load*. Metrik utama yang dianalisis meliputi *latency*, *throughput*, *error rate*, serta penggunaan CPU dan memori. Hasil pengujian menunjukkan bahwa strategi *prewarm* berhasil mengeliminasi *cold start*, menurunkan *latency* maksimal secara signifikan sebesar 72,2% (dari 20,69 detik menjadi 5,74 detik) dibandingkan strategi *default*. Selain itu, pada pengujian beban tinggi jangka panjang, strategi *prewarm* menunjukkan stabilitas sumber daya yang superior dengan penggunaan CPU yang stabil di kisaran 25%, berbeda dengan strategi *default* yang mengalami ketidakstabilan berupa lonjakan CPU hingga 70% dan *memory creep*. Berdasarkan hasil tersebut, strategi *prewarm connection pool* direkomendasikan untuk lingkungan produksi dengan *traffic* tinggi atau layanan yang membutuhkan ketersediaan instan.**

Kata Kunci— ***Connection Pool*, Go, MinIO, *Object Storage*, *Performance Testing*, *Prewarm*.**

I. Pendahuluan

Aplikasi digital modern menghadapi tantangan signifikan dalam mengelola volume data yang terus meningkat secara eksponensial, terutama data multimedia tidak terstruktur. Sistem tidak hanya harus menyimpan data dalam jumlah besar, tetapi juga mampu mengelolanya dengan performa tinggi untuk menghindari *bottleneck* I/O [1][2]. Untuk mengatasi kebutuhan skalabilitas ini, arsitektur *Object Storage* seperti MinIO, AWS S3, dan GCP Storage menjadi standar industri karena menawarkan kapasitas tak terbatas dan biaya yang lebih efisien dibandingkan penyimpanan berbasis blok [3]. Namun, adopsi *object storage* dalam arsitektur *backend* menghadirkan tantangan unik dalam pengelolaan koneksi yang berbeda secara fundamental dari *database* relasional.

Interaksi dengan *database* relasional umumnya bersifat transaksional dengan frekuensi tinggi dan latensi rendah di atas protokol yang *stateful* [4]. Sebaliknya, komunikasi dengan *object storage* didominasi oleh transfer data berukuran besar melalui protokol HTTP/HTTPS yang bersifat *stateless*. Meskipun optimalisasi koneksi melalui mekanisme *Connection Pooling* telah terbukti krusial untuk meningkatkan performa sistem *database*, seperti ditunjukkan oleh Sobri et al. yang menemukan bahwa konfigurasi *pool* yang tepat dapat menurunkan latensi hingga 53% pada arsitektur *microservice* [5], literatur yang membahas spesifik mengenai strategi *pooling* untuk *object storage* masih sangat terbatas. Mayoritas penelitian masih berfokus pada ranah *database* relasional atau fungsi *serverless* [6].

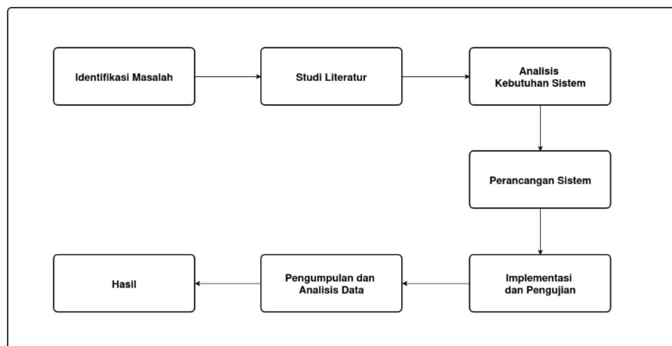
Dalam ekosistem bahasa pemrograman Go, manajemen koneksi HTTP ditangani oleh http.Transport yang secara *default* menerapkan strategi *Lazy Initialization* (pembuatan koneksi secara *on-demand*) [7]. Strategi ini efisien dalam penggunaan memori saat kondisi diam (*idle*), namun rentan terhadap masalah *cold start penalty*, yaitu tingginya latensi pada permintaan pertama karena *overhead* inisialisasi koneksi TCP dan TLS [8]. Sebagai alternatif, strategi *Prewarm* atau *Eager Initialization*, yang mengadopsi konsep *prebaking* pada lingkungan FaaS, menawarkan solusi dengan menyiapkan koneksi di awal *startup* [6], namun dengan konsekuensi alokasi sumber daya yang lebih besar. Efisiensi akses terhadap *storage* sangat bergantung pada konfigurasi *pool* yang tepat, terutama dalam lingkungan dengan tingkat konkurensi tinggi [9][10].

Ketidakpastian mengenai *trade-off* performa antara kedua strategi ini pada konteks *object storage* menciptakan celah pengetahuan yang perlu diisi. Penelitian ini bertujuan untuk membandingkan secara empiris performa strategi *Default Connection Pool* dan *Prewarm Connection Pool* pada integrasi aplikasi Go dengan MinIO. Fokus utama penelitian adalah menganalisis dampak masing-masing strategi terhadap metrik kritis meliputi *latency*, *throughput*, dan efisiensi sumber daya (CPU & Memori) di bawah berbagai skenario beban kerja menggunakan alat uji K6. Hasil penelitian diharapkan dapat memberikan panduan teknis bagi pengembang dalam memilih strategi manajemen koneksi yang paling optimal sesuai karakteristik beban kerja aplikasi mereka.

II. Metodologi Penelitian

Penelitian ini menggunakan metode eksperimental untuk mengukur dan membandingkan performa strategi *connection pool* secara terkontrol. Pelaksanaan penelitian dilakukan

melalui tujuh tahapan sistematis, yaitu: (1) identifikasi masalah, (2) studi literatur, (3) analisis kebutuhan, (4) perancangan sistem, (5) implementasi dan pengujian, (6) pengumpulan dan analisis data, serta (7) hasil.



Gbr. 1 Tahapan Penelitian

A. *Identifikasi Masalah*

Tahap ini berfokus pada analisis kesenjangan (*gap analysis*) terkait minimnya literatur mengenai optimasi koneksi pada *object storage*. Masalah utama yang diidentifikasi adalah perbedaan karakteristik antara *database* relasional dan *object storage*, di mana strategi *pooling* yang umum digunakan pada *database* belum tentu optimal jika diterapkan langsung pada *object storage* yang bersifat *stateless*.

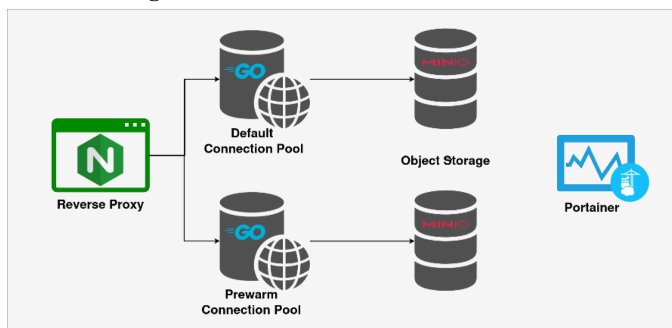
B. *Studi Literatur*

Kajian mendalam dilakukan terhadap konsep *Connection Pool* (khususnya strategi *Lazy* vs *Eager Initialization*), arsitektur MinIO, dan bahasa pemrograman Go. Studi ini menjadi landasan teoretis untuk merancang skenario pengujian yang relevan dan memahami parameter kinerja yang krusial.

C. *Analisis Kebutuhan*

Spesifikasi infrastruktur ditetapkan untuk mendukung validitas pengujian. Sisi *Server* menggunakan VPS dengan prosesor AMD EPYC 9354P (32-Core), RAM 8 GB, dan penyimpanan SSD berbasis Ubuntu 20.04. Sisi *Client* (*load generator*) menggunakan perangkat berbasis AMD Ryzen 3 dengan RAM 8 GB. Perangkat lunak utama meliputi Docker untuk orkestrasi kontainer, K6 untuk *load testing*, serta Prometheus dan Grafana untuk pemantauan metrik.

D. *Perancangan Sistem*



Gbr. 1 Arsitektur Sistem

Arsitektur sistem dirancang dengan topologi *client-server* seperti tampak di Gbr 1. Nginx dikonfigurasi sebagai *reverse proxy* yang membagi lalu lintas ke dua *instance* aplikasi *backend* (Refina) berbasis Go: satu menerapkan *Default Connection Pool* dan lainnya menerapkan *Prewarm Connection Pool*. Kedua aplikasi terhubung ke *instance* MinIO yang terisolasi untuk memastikan kemurnian data uji.

E. *Implementasi dan Pengujian*

Implementasi dilakukan dengan mengembangkan aplikasi studi kasus "Refina" berbasis Go yang menyediakan *endpoint* unggah *file* ke MinIO. Pengujian performa dilaksanakan menggunakan instrumen K6 melalui empat skenario yang masing-masing difokuskan pada metrik spesifik: (1) *Cold Start Burst* yang menargetkan analisis *Latency* awal akibat inisialisasi koneksi; (2) *Gradual Ramp-up* untuk mengevaluasi batas toleransi sistem berdasarkan *Error Rate*; (3) *Spike Test* yang mengukur stabilitas *Throughput* (RPS) saat terjadi fluktuasi beban ekstrem ; dan (4) *Sustained High Load* untuk memvalidasi efisiensi dan stabilitas *Resource Usage* (CPU dan Memori) dalam jangka panjang.

TABEL I
TEST CASE SCENARIO K6

No	Nama Test Case	Virtual Users	Durasi	File Size
1	TC1 - Cold Start Burst	0 → 100 Vus	2 menit 40 detik	20 KB
2	TC2 - Gradual Ramp-up Load	0 → 150 Vus	8 menit 30 detik	1 MB
3	TC3 - Spike Test	0 → 250 VUs (2x Spike)	8 menit	20 KB
4	TC4 - Sustained High Load	0 → 100 VUs (Konstan)	15 menit	20 KB

F. *Pengumpulan dan Analisis Data*

Data dikumpulkan secara *real-time* menggunakan protokol *Remote Write* dari K6 ke Prometheus. Metrik penggunaan sumber daya (CPU dan Memori) dipantau menggunakan Portainer. Seluruh data disentralisasi ke dalam *dashboard* Grafana untuk mempermudah visualisasi tren performa antar kedua strategi.

G. *Hasil*

Tahap akhir melibatkan ekstraksi dan komparasi metrik performa utama, yaitu *Latency* (Avg, P95, Max), *Throughput* (RPS), *Error Rate*, serta efisiensi sumber daya (CPU dan *Memory*).

III. HASIL DAN PEMBAHASAN

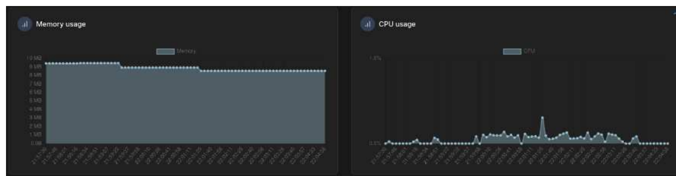
Paragraf harus teratur. Semua paragraf harus rata, yaitu sama-sama rata kiri dan dan rata kanan.

A. *Hasil Pengujian Baseline (Kondisi Idle)*

 ISSN : 2686-2220

Pengujian *baseline* bertujuan untuk mengamati perilaku manajemen memori dan CPU saat aplikasi baru dijalankan (*startup*) dan belum menerima trafik.

1) *Resource Utilization pada Default Connection Pool:* Pada strategi Default (*Lazy Initialization*), aplikasi tidak membuat koneksi saat *startup*. Pola Penggunaan Memori:

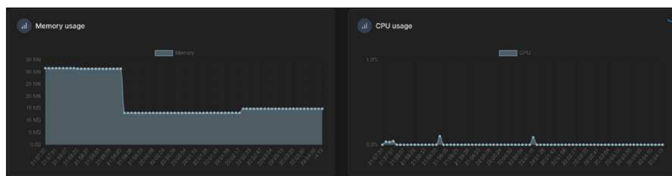


Gbr. 3 Grafik Resource Usage pada Default Connection Pool saat Idle

Berdasarkan pemantauan pada Gbr. 3, karakteristik sistem adalah sebagai berikut:

1. Penggunaan memori relatif rendah, bergerak stabil di kisaran 8.5–9.5 MB. Grafik menunjukkan fluktuasi kecil (naik-turun) sekitar 8–12%. Hal ini wajar karena tidak ada beban koneksi yang disimpan (*stored connections*), sehingga memori murni digunakan oleh *runtime* dasar Go.
2. Pola Penggunaan CPU: Terdapat aktivitas CPU dengan lonjakan periodik (*periodic spikes*) mencapai 0.4–0.6%, meskipun rata-rata tetap rendah (0.1–0.2%). Ini menunjukkan aktivitas standar *container* tanpa beban manajemen *pool* yang kompleks.

2) Resource Utilization pada Prewarm Connection Pool: Pada strategi Prewarm, aplikasi menginisialisasi koneksi ke MinIO segera setelah startup.



Gbr. 4 Grafik Resource Usage pada Prewarm Connection Pool saat Idle

Berdasarkan Gbr. 2, terlihat perbedaan signifikan akibat proses *prewarming*:

1. Pola Penggunaan Memori: Terjadi lonjakan penggunaan memori yang tinggi di awal *startup* hingga mencapai 30–32 MB. Lonjakan ini adalah dampak langsung dari inisialisasi koneksi massal. Setelah fase ini, memori turun dan stabil di angka 13–15 MB, yang mana lebih tinggi dibandingkan strategi *Default* karena adanya alokasi untuk menjaga koneksi tetap aktif .
2. Pola Penggunaan CPU: Setelah inisialisasi selesai, grafik CPU cenderung sangat datar dan stabil di kisaran 0.0–0.1% dengan lonjakan yang sangat minim. Ini mengindikasikan bahwa setelah koneksi terbentuk (*established*), sistem tidak memerlukan daya komputasi besar untuk memeliharanya.

Perbandingan kedua strategi pada kondisi *idle* dirangkum dalam Tabel II.

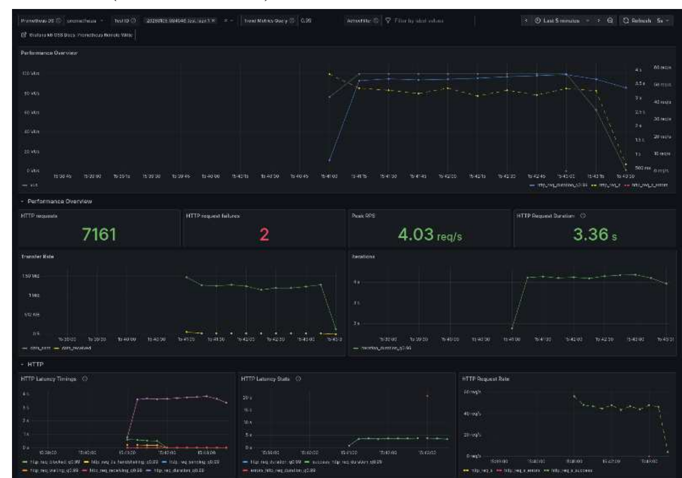
TABEL III
PERBANDINGAN RESOURCE USAGE KONDISI IDLE

Parameter	Default	Prewarm	Analisis
Konsumsi Memori (*Idle*)	Rendah (~9 MB)	Lebih Tinggi (~13 MB)	*Prewarm* membutuhkan memori ekstra untuk memelihara *pool* koneksi yang aktif.
Perilaku *Startup*	Stabil rendah	Lonjakan Tinggi (~32 MB)	Dampak dari inisialisasi koneksi massal di awal pada strategi *Prewarm*.
Stabilitas Grafik	Fluktuatif	Sangat Stabil (Flat)	Setelah *startup*, *Prewarm* cenderung lebih tenang secara *resource* karena koneksi sudah siap (*established*).

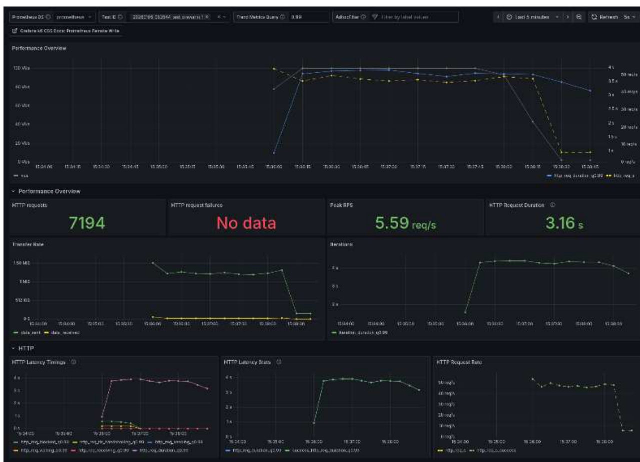
B. *Hasil Pengujian Performa*

Pengujian performa dilakukan dalam empat skenario untuk mengukur *Latency*, *Error Rate*, *Throughput*, dan stabilitas *Resource*.

1) *Skenario 1 Cold Start Burst (Fokus Latency):* Skenario ini mengukur respons sistem saat menghadapi lonjakan pengguna tiba-tiba (0 ke 100 VUs) dari kondisi diam.



Gbr. 5 Default Connection Pool Performance Cold Start Burst Test



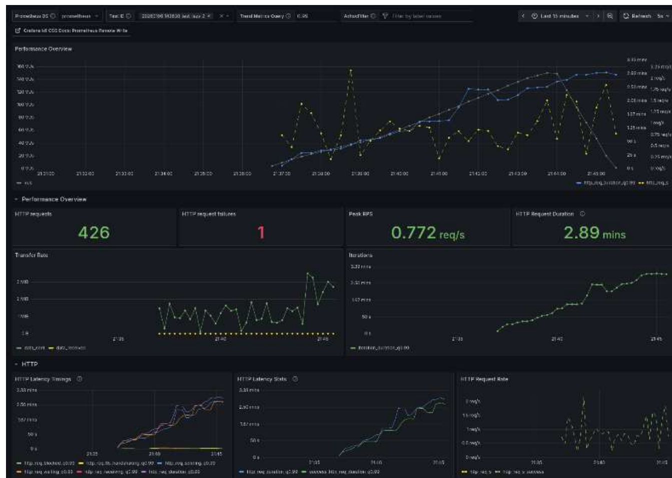
Gbr. 6 Prewarm Connection Pool Performance Cold Start Burst Test

Berdasarkan Gbr. 5 dan Gbr. 6, analisis hasil menunjukkan:

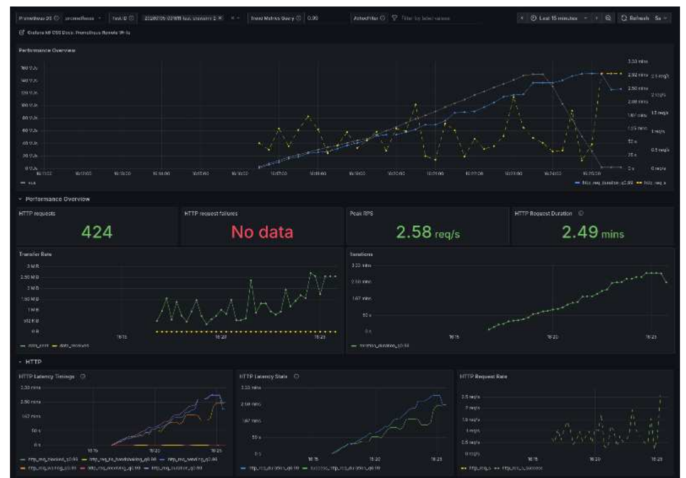
1. *Latency* Maksimal: Strategi *Default* mengalami *Max Latency* hingga 20,69 detik. Hal ini disebabkan oleh *cold start penalty*, di mana request pertama harus menunggu proses TCP/TLS *handshake* selesai. Sebaliknya, *Prewarm* berhasil menahan *Max Latency* di angka 5,74 detik, atau penurunan sebesar 72,2%.

2. Implikasi: Hasil ini membuktikan bahwa strategi *Prewarm* sangat efektif menghilangkan hambatan awal. Untuk aplikasi yang menuntut responsivitas tinggi sejak detik pertama, strategi *Default* tidak memadai karena menyebabkan penundaan signifikan bagi pengguna awal.

2) Skenario 2 Gradual Ramp-up Load (Fokus Error Rate): Skenario ini menguji batas ketahanan sistem dengan mengirimkan *file* besar (1 MB) sembari meningkatkan jumlah pengguna (0 ke 150 VUs).



Gbr. 7 Default Connection Pool Performance Gradual Ramp-up Load Test



Gbr. 8 Prewarm Connection Pool Performance Gradual Ramp-up Load Test

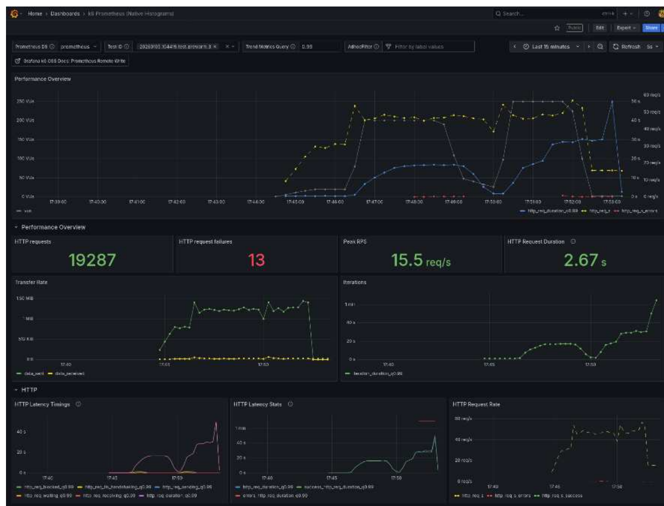
Analisis dari Gbr. 7 dan Gbr. 8 menunjukkan:

1. Tingkat Kegagalan: Kedua strategi mengalami kegagalan tinggi. *Default* mencatat *Error Rate* 55,16%, sedangkan *Prewarm* sebesar 54,48%.

2. Faktor Penyebab: Tingginya *error* pada kedua kubu mengindikasikan bahwa manajemen *connection pool* bukan faktor penentu dalam skenario ini. Kegagalan lebih disebabkan oleh saturasi *bandwidth* jaringan atau *Disk* I/O akibat ukuran *file* yang besar. Optimasi di sisi aplikasi (*pooling*) terbukti tidak dapat sepenuhnya mengompensasi keterbatasan infrastruktur fisik.

3) Skenario 3 Spike Test (Fokus Throughput): Skenario ini mensimulasikan lonjakan trafik ekstrem yang berulang (0 ke 250 VUs) untuk mengukur kapasitas pemrosesan.

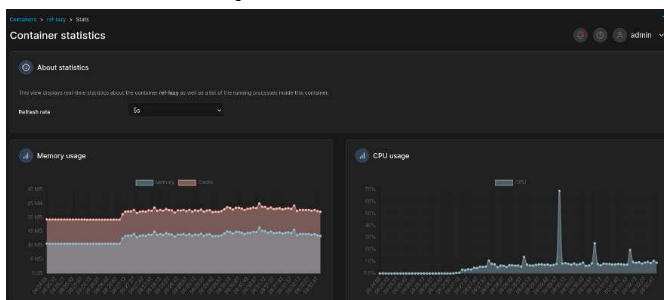
Gbr. 9 Default Connection Pool Performance Spike Test

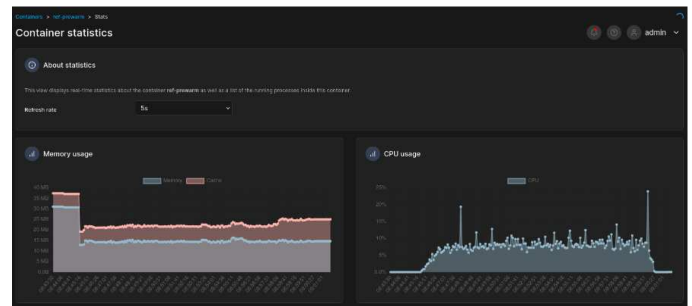

Gbr. 10 Prewarm Connection Pool Performance Spike Test

Berdasarkan Gbr. 9 dan Gbr. 10, ditemukan bahwa:

1. Kapasitas Pemrosesan: Strategi *Prewarm* mencatatkan total permintaan sukses yang lebih banyak (19.287 vs 19.110) dan rata-rata *throughput* yang lebih tinggi (39,94 req/s vs 39,49 req/s).
2. Stabilitas: Meskipun selisih angka terlihat kecil (+1,1%), *Prewarm* menunjukkan konsistensi yang lebih baik dalam menjaga aliran data. Strategi *Default* harus berjuang membuka koneksi baru secara reaktif saat *spike* terjadi, yang berpotensi menghambat antrian permintaan. *Prewarm* yang memiliki koneksi siaga mampu memproses antrian dengan lebih mulus.

4) Skenario 4 Sustained High Load (Fokus Resource): Skenario jangka panjang (15 menit) ini bertujuan mendeteksi efisiensi internal dan potensi kebocoran memori.


Gbr. 11 Default Connection Pool Performance Gradual Ramp-up Load Test


Gbr. 12 Prewarm Connection Pool Performance Gradual Ramp-up Load Test

Perbandingan detail penggunaan sumber daya disajikan dalam Tabel III

TABEL IIIII
PERBANDINGAN RESOURCE USAGE PADA SUSTAINED HIGH LOAD

Parameter	Default	Prewarm	Analisis
Pola Memori	Naik Perlahan	Sangat Stabil	*Lazy* mengalami kenaikan memori +40-50% seiring waktu, berisiko kebocoran (*leak*) jangka panjang.
Rentang Memori	10 → 16 MB (Tren Naik)	35 → 16 MB (Konstan)	*Prewarm* menjaga konsistensi alokasi tanpa akumulasi memori.
Pola CPU	Fluktuatif Tinggi	Datar	*Fluktuasi* CPU pada *Lazy* menandakan manajemen koneksi yang tidak efisien dalam durasi panjang.
Lonjakan CPU (*Spike*)	Ekstrem (~70%)	Terkendali (~25%)	Lonjakan 70% pada *Lazy* berbahaya karena dapat memblokir proses lain dan menyebabkan waktu respons tidak stabil.
Kesimpulan	Berisiko Degradasi	Sangat Stabil	*Lazy* menyimpan risiko instabilitas untuk layanan *always-on*, sedangkan *Prewarm* sangat konsisten.

Analisis mendalam dari Gbr. 11, Gbr. 12, dan Tabel III menunjukkan:

3. Instabilitas Strategi *Default*: Strategi *Default* terbukti tidak stabil untuk beban jangka panjang. Hal ini ditandai dengan fenomena *Memory Creep* (kenaikan penggunaan memori bertahap tanpa pembersihan efektif) dan lonjakan CPU ekstrem hingga 70%. Aktivitas buka-tutup koneksi yang terus menerus membebani *Garbage Collector* dan CPU.
4. Keunggulan Strategi *Prewarm*: Sebaliknya, strategi *Prewarm* menunjukkan profil penggunaan sumber daya yang datar (*flat*). Meskipun membutuhkan alokasi memori awal lebih tinggi, ia berhasil menjaga penggunaan CPU tetap tenang di

kisaran 25%. Hal ini menegaskan bahwa untuk layanan jangka panjang (*long-running services*), *Prewarm* jauh lebih aman dan terprediksi, mencegah degradasi performa seiring berjalannya waktu .

IV. Kesimpulan

Penelitian ini berhasil mengungkap karakteristik performa dan efisiensi antara strategi *Default Connection Pool* (Lazy) dan *Prewarm Connection Pool* pada integrasi MinIO menggunakan bahasa Go. Berdasarkan analisis hasil pengujian, strategi *Prewarm* terbukti secara signifikan mengungguli strategi *Default* dalam aspek responsivitas awal. Strategi ini mampu mengeliminasi *cold start penalty* dan memangkas latensi maksimal (*Max Latency*) hingga 72,2% (dari 20,69 detik menjadi 5,74 detik) dibandingkan strategi *Default* yang mengalami hambatan inisialisasi koneksi berat saat menerima trafik pertama .

Dalam hal stabilitas sistem jangka panjang, strategi *Prewarm* menunjukkan superioritas yang jelas. Strategi ini berhasil mempertahankan penggunaan sumber daya yang datar (*flat*) dan terprediksi, serta menjaga penggunaan CPU stabil di kisaran 25% bahkan di bawah beban kerja konstan selama 15 menit. Sebaliknya, strategi *Default* terbukti rentan terhadap instabilitas internal, ditandai dengan fenomena kenaikan penggunaan memori akumulatif (*memory creep*) dan lonjakan CPU ekstrem hingga 70% yang berisiko mendegradasi performa layanan *always-on*. Meskipun demikian, terdapat *trade-off* yang perlu diperhatikan. Strategi *Prewarm* membutuhkan alokasi memori awal yang lebih besar saat *startup* demi menjaga kesiapan koneksi.

Ucapan Terima Kasih

Puji syukur ke hadirat Tuhan Yang Maha Esa atas selesainya penelitian ini. Penulis mengucapkan terima kasih kepada Fakultas Teknik, Universitas Negeri Surabaya yang telah memfasilitasi pelaksanaan penelitian. Penulis juga mengapresiasi dukungan emosional maupun informasional yang diberikan oleh keluarga dan rekan-rekan selama proses penelitian berlangsung.

Referensi

[1] H. Elshazly, J. Ejarque, and R. Badia, "Storage-Heterogeneity Aware Task-based Programming Models To Optimize I/O Intensive Applications," IEEE Trans. Parallel Distrib. Syst., 2022.

[2] S. Mahmoudi, M. Belarbi, S. Mahmoudi, G. Belalem, and P. Manneback, "Multimedia processing using deep learning technologies, high-performance computing cloud resources, and Big Data volumes," *Concurr. Comput. Pract. Exp.*, vol. 32, 2020.

[3] D. Durner, D. Broneske, G. Graefe, and W. Lehner, "Exploiting Cloud Object Storage for High-Performance Analytics," *Proc. VLDB Endowment*, vol. 16, no. 11, pp. 2769-2781, 2023.

[4] C. Moreau, C. Legroux, V. Peralta, and M. Hamrouni, "Mining SQL workloads for learning analysis behavior," *Inf. Syst.*, vol. 108, 2022.

[5] N. A. N. Sobri, M. A. H. Abas, I. I. Mohd Yassin, M. S. A. M. Ali, et al., "A Study of Database Connection Pool in Microservice Architecture," *JOIV: Int. J. on Informatics Visualization*, vol. 6, no. 2-2, 2022.

[6] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in Proc. 21st Int. Middleware Conf., 2020, pp. 1-13.

[7] "Lazy Initialization in Go: Efficient Resource Management," Software Patterns Lexicon, Oct. 2024. [Online]. Available: https://softwarepatternslexicon.com/patterns-go/11/2/, tanggal akses: 28 Januari 2026.

[8] "Lazy Initialization," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/framework/performance/lazy-initialization, tanggal akses: 28 Januari 2026.

[9] G. Brooker, "Optimising Database Connection Management in Go Applications," *J. of Cloud Computing*, 2021.

[10] D. Yadav and A. If, "Using Asynchronous Frameworks and Database Connection Pools to Enhance Web Application Performance in High-Concurrency Environments," in *Proc. ISMAC*, 2024. [Online]. Available: https://www.researchgate.net/publication/385174027, tanggal akses: 28 Januari 2026.