Perbandingan Performa dan Skalabilitas Arsitektur Multi-Tenant Database pada Sistem Backend Point of Sale

Ahmad Nur Sajidan[1], I Made Suartana[2]

[1,2] Jurusan Teknik Informatika, Fakultas Teknik, Universitas Negeri Surabaya
[1]ahmad.22061@mhs.unesa.ac.id
[2]madesuartana@unesa.ac.id

***Abstrak*— Sistem Point of Sale (POS) berbasis Software as a Service (SaaS) menghadapi tantangan penting dalam memilih arsitektur basis data multi-tenant yang tepat, yaitu antara pendekatan single-database (shared-schema) dan multi-database per tenant, karena masing-masing membawa konsekuensi berbeda terhadap performa, skalabilitas, dan efisiensi sumber daya. Perbandingan kedua arsitektur dilakukan melalui pengujian eksperimental menggunakan API berbasis Golang, basis data PostgreSQL, dan workload generator Locust dengan mengadaptasi pola benchmark TPC-C. Skala pengujian divariasikan dari skala kecil (5 tenant, 50 user) hingga ekstrem (150 tenant, 1.500 user). Hasil pengujian menunjukkan kedua arsitektur memberikan performa setara dan stabil, dengan throughput mencapai 170 RPS, waktu respons di bawah 12 milidetik, dan error rate 0% pada beban puncak, serta mampu berskala secara hampir linear terhadap peningkatan beban. Perbedaan signifikan justru terlihat pada efisiensi server basis data: arsitektur single-database terbukti 2,65 kali lebih hemat memori, stabil pada penggunaan sekitar 950 MB RAM, dan mampu mempertahankan cache hit ratio 98,88%. Sebaliknya, arsitektur multi-database mengalami lonjakan konsumsi RAM hingga 210,9% (mencapai 2.518 MB) akibat beban pemeliharaan ratusan connection pool independen, disertai peningkatan aktivitas disk write 16%-40%. Temuan ini menunjukkan bahwa arsitektur single-database lebih sesuai untuk platform POS berskala startup atau UMKM karena efisiensi biaya infrastruktur, dengan syarat penerapan validasi kode yang ketat, sedangkan arsitektur multi-database lebih cocok bagi bisnis berskala korporasi (enterprise) yang membutuhkan isolasi privasi absolut meskipun harus mengalokasikan kapasitas infrastruktur yang lebih besar.**

***Kata Kunci*— Point of Sale (POS), SaaS, Multi-tenant, Arsitektur Basis Data, Performa, Skalabilitas.**

I. PENDAHULUAN

Sistem Point of Sale (POS) digunakan untuk mencatat, mengelola, dan memproses transaksi penjualan pada berbagai jenis usaha, mulai dari ritel hingga usaha kecil menengah dan restoran [1]. Fleksibilitas dan kemudahan akses yang diberikan sistem POS berbasis cloud menarik perhatian perusahaan, terutama yang memiliki banyak cabang [2]. Namun, seiring bertambahnya cabang dan pengguna yang mengakses sistem secara bersamaan, kebutuhan terhadap arsitektur basis data yang andal, efisien, serta mampu menangani skala permintaan besar semakin meningkat, sehingga pengelolaan sistem POS menjadi faktor strategis yang menentukan efektivitas operasional dan pengalaman pelanggan.

Sistem POS berbasis cloud merupakan salah satu bentuk implementasi Software as a Service (SaaS), dan salah satu tantangan utamanya adalah pengelolaan basis data multi-tenant [3], yaitu model yang memungkinkan satu aplikasi melayani banyak perusahaan (tenant) dengan tetap menjaga isolasi data [4]. Dalam konteks penelitian ini, tenant didefinisikan sebagai pelanggan penyedia layanan retail dan sejenisnya, sedangkan cabang (branch) dipandang sebagai bagian dari tenant yang dibedakan melalui branch_id. Terdapat dua pendekatan umum arsitektur database multi-tenant, yaitu single-database (shared-database shared-schema) dan multi-database per tenant [5], [6]. Pendekatan single-database menggunakan satu basis data dengan kolom tenant_id sebagai pembeda; efisien namun berisiko pada isolasi data dan performa jika salah satu tenant memiliki volume data besar. Sebaliknya, multi-database memberikan setiap tenant basis data tersendiri sehingga isolasi lebih tinggi, namun menimbulkan overhead pengelolaan dan konsumsi sumber daya yang lebih besar. Pemilihan arsitektur yang kurang tepat dapat menimbulkan risiko penurunan kinerja sistem, meningkatnya biaya infrastruktur, serta menurunnya kepuasan pengguna [3].

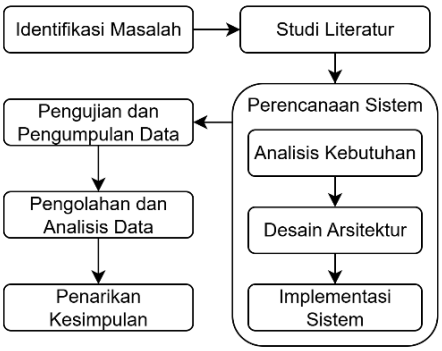
Performa dan skalabilitas merupakan dua aspek penting dalam pemilihan arsitektur multi-tenant. Performa umumnya diukur melalui response time dan throughput [7], sedangkan skalabilitas merujuk pada kemampuan sistem mempertahankan kinerja seiring peningkatan jumlah tenant maupun volume transaksi [8]. Penelitian-penelitian sebelumnya banyak berfokus pada optimasi query database, efisiensi sumber daya, atau perancangan arsitektur multi-tenant pada SaaS secara umum [5], [9], namun belum banyak membahas bagaimana arsitektur multi-tenant memengaruhi performa lapisan backend seperti API, connection pooling, serta konsumsi sumber daya server dalam konteks sistem POS yang membutuhkan pemrosesan transaksi cepat dan real-time. Padahal sistem POS memiliki karakteristik beban kerja read-write yang tinggi, kebutuhan konsistensi stok real-time, dan variasi skala tenant yang beragam, sehingga evaluasi yang hanya menilai performa basis data tidak lagi memadai untuk menggambarkan perilaku sistem POS secara utuh.

Berdasarkan kondisi tersebut, penelitian ini bertujuan menganalisis dan membandingkan performa serta skalabilitas sistem backend POS berbasis arsitektur single-database dan multi-database, melalui pengujian pada level API dan basis data menggunakan variasi beban yang mengadaptasi pola TPC-C. Penelitian ini juga berupaya mengidentifikasi faktor teknis yang memengaruhi performa dan skalabilitas masing-masing

arsitektur, serta merumuskan rekomendasi praktis bagi pengembang sistem POS maupun penyedia layanan SaaS dalam menentukan strategi pengelolaan database yang optimal.

II. METODE PENELITIAN

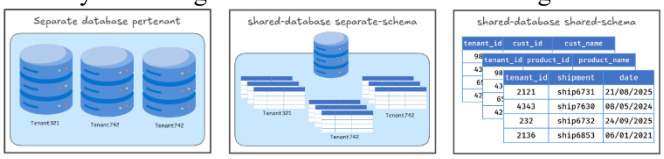
Cara Penelitian ini menggunakan metode eksperimental agar pengamatan terhadap sistem dapat dilakukan secara terkendali, terukur, dan dapat direplikasi, karena fokus penelitian adalah menguji, mengukur, dan membandingkan performa serta skalabilitas sistem backend Point of Sale (POS) yang dibangun di atas dua arsitektur basis data multi-tenant, yaitu single-database (shared-schema) dan multi-database per tenant. Penelitian diawali dengan identifikasi masalah mengenai belum tersedianya perbandingan langsung kedua arsitektur pada lingkungan pengujian terkontrol, dilanjutkan dengan studi literatur untuk memahami konsep multi-tenancy serta metrik performa dan skalabilitas yang relevan, sebelum masuk ke tahap perencanaan sistem, pengujian, dan analisis data sebagaimana ditunjukkan pada Gbr. 1.



Gbr. 1 Diagram alur penelitian.

A. Perencanaan dan Perancangan Sistem

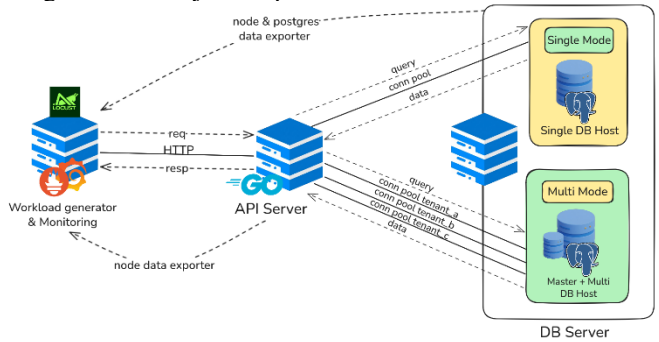
Tahap perencanaan diawali dengan analisis kebutuhan fungsional (manajemen produk, transaksi penjualan dan pembelian, laporan stok, serta riwayat transaksi) dan non-fungsional (performa, skalabilitas, efisiensi sumber daya, serta konsistensi transaksi ACID) agar lingkungan eksperimen dapat mendukung tujuan penelitian secara konsisten dan dapat direplikasi. Pada tahap desain arsitektur, kedua model multi-tenancy dirancang mengacu pada konsep umum shared-database shared-schema dan separate database per-tenant [5], [6], sebagaimana diilustrasikan pada Gbr.2 di mana pendekatan shared-schema unggul dalam efisiensi sumber daya namun memiliki isolasi data yang lebih rendah, sedangkan separate database per-tenant memberikan isolasi lebih tinggi dengan konsekuensi konsumsi sumber daya yang lebih besar [3]. Sistem uji kemudian diimplementasikan berbasis layered architecture menggunakan bahasa pemrograman Go, dengan basis data PostgreSQL yang mendukung kedua model tersebut. Lapisan API terdiri atas routes, handler, service, dan repository layer; mekanisme multi-tenancy diimplementasikan sepenuhnya pada repository layer. Pada arsitektur single-database, setiap tabel inti memiliki kolom tenant_id sebagai filter query melalui satu connection pool terpusat, sedangkan pada arsitektur multi-database setiap tenant memiliki basis data terpisah dengan skema identik, dan API memetakan koneksi tenant melalui master database yang berfungsi sebagai directory service agar koneksi tidak dibuat berulang.



Gbr. 2 Perbandingan pendekatan arsitektur basis data multi-tenant yang menjadi acuan perancangan sistem: separate database per-tenant, shared-database separate-schema, dan shared-database shared-schema.

B. Pengujian dan Pengumpulan Data

Pengujian dilakukan dengan memisahkan komponen workload generator dan monitoring, API server, serta database server pada mesin yang berbeda untuk menghindari bias performa akibat resource sharing. Locust digunakan sebagai workload generator, sedangkan Prometheus dan Grafana digunakan untuk memantau metrik CPU, memori, dan disk I/O pada API server dan database server secara real-time (Gbr. 3); seluruh komponen dijalankan pada lingkungan Docker terisolasi untuk menjaga konsistensi eksperimen. Beban kerja dirancang mengadaptasi prinsip dasar benchmark TPC-C [10], yaitu variasi jenis transaksi OLTP yang berjalan konkuren dengan proporsi write lebih dominan dibanding read, serta penerapan keying time (0,5-1 detik) dan think time (1-10 detik) untuk meniru perilaku operator, dengan proporsi transaksi sebagaimana ditunjukkan pada Tabel I.



Gbr. 3 Topologi pengujian dan pemantauan system.

TABEL I
RASIO WORKLOAD PENGUJIAN

Jenis Transaksi	Endpoint	Proporsi
Transaksi Penjualan	POST /sale	43%
Manajemen Stok	POST /transfer	22%
Restock	POST /purchase	18%
Pengecekan Stok	GET /inventory	4%
Analisis & Laporan	GET /reports, /history	13%

Skenario pengujian divariasikan dari skala small (5 tenant, 50 virtual user) hingga extreme (150 tenant, 1.500 virtual user), dengan warehouse per tenant sebanyak 3 dan branch per warehouse sebanyak 3, mengikuti struktur hierarkis serupa TPC-C. Setiap skenario dijalankan selama 10-15 menit dan

diulang minimal tiga kali untuk mengurangi pengaruh fluktuasi sistem, dengan isolation level database read committed, dan hasilnya dicatat sebagai data mentah untuk tahap pengolahan berikutnya.

C. Pengolahan dan Analisis Data

Data mentah hasil pengujian diolah menjadi metrik performa aplikasi (response time rata-rata, P95, P99, throughput, dan error rate) dari sisi Locust, serta metrik resource utilization (CPU, memori, dan disk I/O) dari API server dan database server melalui Prometheus [7]. Skalabilitas selanjutnya dianalisis dengan mengacu pada konsep scale-up, scale-out, dan elastic scalability [8] melalui rasio throughput terhadap peningkatan beban (Scale_load) dan terhadap peningkatan jumlah tenant (Scale_tenant), sebagaimana persamaan (1) dan (2), sedangkan efisiensi sumber daya dianalisis melalui rasio throughput terhadap konsumsi CPU dan memori rata-rata [11]. Seluruh metrik dari kedua arsitektur kemudian dibandingkan secara berdampingan pada setiap skala pengujian untuk menarik kesimpulan mengenai keunggulan dan keterbatasan masing-masing pendekatan.

Skalabilitas beban (Scale_load) mengukur seberapa proporsional throughput sistem meningkat ketika jumlah virtual user dinaikkan terhadap kondisi acuan, dirumuskan dalam (1).

$$Scale_{load} = ThroughputVU_n / ThroughputVU_{baseline} \quad (1)$$

dengan ThroughputVUn merupakan throughput sistem yang diukur pada faktor beban virtual user ke-n, dan ThroughputVUbaseline merupakan throughput sistem pada skenario acuan, yaitu skala small (50 virtual user). Nilai Scale_load kemudian dibandingkan terhadap faktor kenaikan virtual user itu sendiri, sehingga menghasilkan tiga kemungkinan zona skalabilitas: sub-linear apabila Scale_load lebih kecil dari faktor beban, linear apabila keduanya sama, dan super-linear apabila Scale_load melebihi faktor beban.

Skalabilitas tenant (Scale_tenant) selanjutnya digunakan untuk mengukur dampak penambahan jumlah tenant terhadap stabilitas throughput rata-rata yang diperoleh setiap tenant, sebagaimana dirumuskan dalam (2).

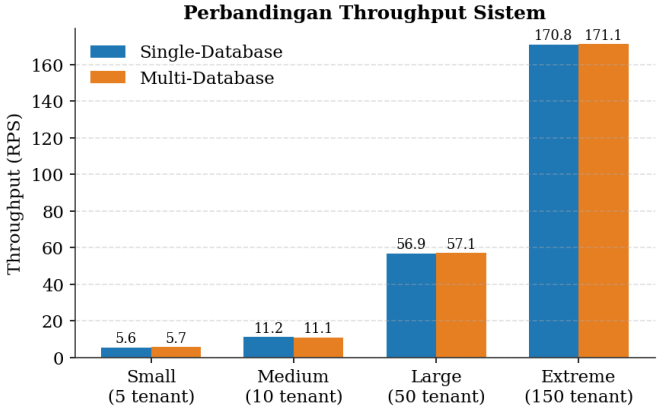
$$Scale_{tenant} = Throughput_{N\ Tenant} / N \times Throughput_{1\ Tenant} \quad (2)$$

dengan ThroughputN-tenant merupakan throughput total sistem yang tercatat pada skenario dengan N tenant, N merupakan jumlah tenant pada skenario tersebut, dan Throughput1-tenant merupakan throughput rata-rata satu tenant pada skenario acuan. Nilai Scale_tenant yang mendekati 1,0 mengindikasikan bahwa penambahan jumlah tenant tidak menyebabkan degradasi throughput individual, sedangkan nilai yang jauh di bawah 1,0 mengindikasikan adanya kontensi sumber daya antartenant.

III. Hasil dan Pembahasan

A. Performa Aplikasi

Pada skala tertinggi (extreme), sistem memproses sekitar 97.000 permintaan dalam 10 menit. Kedua arsitektur mencatatkan throughput yang nyaris identik: single-database mencapai rata-rata 170,81 RPS dan multi-database 171,11 RPS, dengan error rate 0,00% pada seluruh skenario dan tanpa satu pun kegagalan HTTP 500 maupun deadlock/rollback (Gbr. 4). Rata-rata response time keduanya konsisten di bawah 12 ms, dengan selisih di bawah 1,5 ms pada seluruh skala, dan nilai P99 tidak pernah melampaui 42 ms bahkan pada skenario extreme (Tabel II). Selisih tipis yang muncul pada beberapa skala lebih dipengaruhi oleh variasi normal lalu lintas jaringan dan pemrosesan sistem operasi (margin of error), bukan indikasi superioritas performa salah satu arsitektur, sehingga secara praktis kedua arsitektur setara dari sudut pandang pengguna akhir.



Gbr. 4 Grafik Throughput.

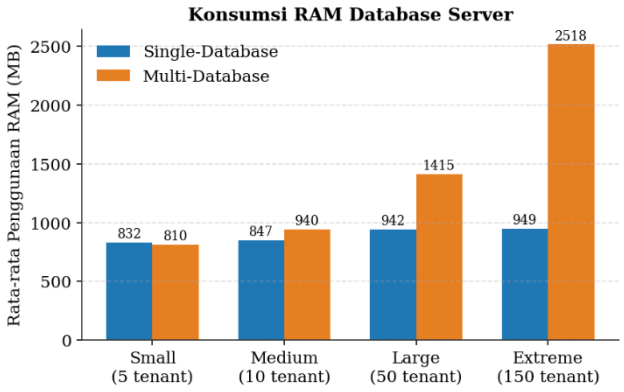
Tabel II
Response Time dan Persentil

Skala	Single-DB	Multi-DB	P99 Single	P99 Multi
Small	10,34	9,48	24	21
Medium	10,63	9,50	42	19
Large	10,78	9,80	21	21
Extreme	11,58	10,41	30	24

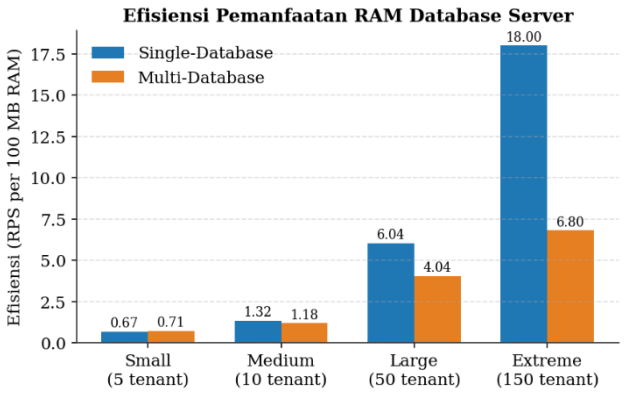
B. Efisiensi Sumber Daya Memori

Perbedaan paling menonjol terjadi pada efisiensi memori database server. Arsitektur single-database hanya membutuhkan rata-rata 949,1 MB RAM pada skala extreme (naik 14,1% dari baseline), sedangkan multi-database melonjak hingga 2.518 MB (naik 210,9%) dengan beban puncak menyentuh 3.082 MB (Gbr. 5). Dari sisi rasio efisiensi, single-database menghasilkan 18,00 RPS per 100 MB RAM database dibanding 6,80 RPS pada multi-database, atau sekitar 2,65 kali lebih hemat (Gbr. 6). Pada lapisan API server, single-database juga sekitar 2,44 kali lebih hemat memori. Tren pertumbuhan CPU dan RAM pada kedua lapisan server dari skala small hingga extreme dirangkup pada Tabel III, yang menegaskan bahwa lonjakan konsumsi sumber daya paling signifikan justru terjadi pada RAM database server arsitektur multi-database. Pola serupa juga terjadi pada lapisan API server, di mana

single-database mencatatkan 24,72 RPS per 10 MB RAM dibanding hanya 10,14 RPS pada multi-database, atau sekitar 2,44 kali lebih hemat.



Gbr. 5 Konsumsi RAM database server pada setiap skala pengujian.



Gbr. 6 Efisiensi pemanfaatan RAM database server (RPS per 100 MB).

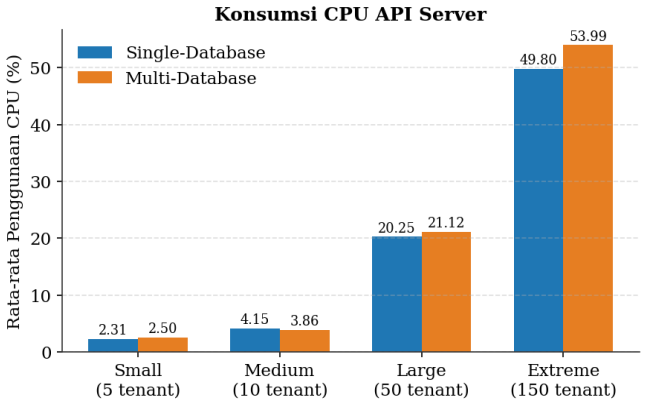
Tingginya konsumsi RAM pada multi-database disebabkan oleh dua faktor utama, yaitu fragmentasi memori akibat pengelolaan 150 basis data fisik beserta proses internalnya secara terpisah, serta penerapan strategi persistent connection pooling pada layer API berbasis pgxpool yang menahan (cache) rata-rata sekitar 525 koneksi idle di dalam memori database pada skala extreme. Meskipun memakan kapasitas RAM yang besar, pendekatan ini terbukti krusial untuk menjaga latensi tetap rendah karena sistem terhindar dari proses inisialisasi koneksi (forking dan TCP handshake) yang berulang kali, sehingga tingginya penggunaan RAM pada multi-database merupakan kompromi (trade-off) yang sulit dihindari demi mempertahankan stabilitas performa.

TABEL III
PERTUMBUHAN KONSUMSI SUMBER DAYA

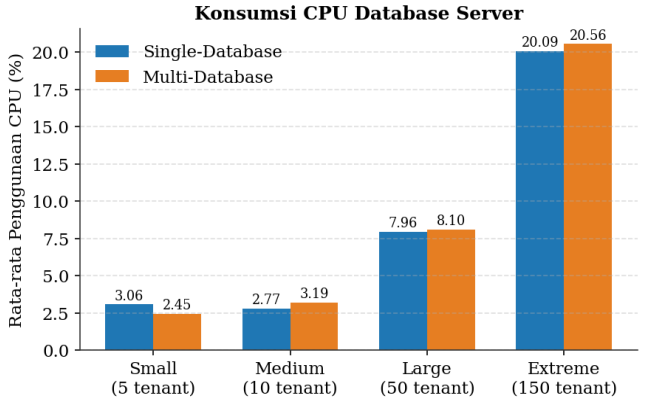
Metrik	Single-DB	Multi-DB
CPU API	2,31% → 49,80%	2,25% → 53,99%
RAM API (MB)	15,83 → 69,11	16,60 → 168,73
CPU DB	3,06% → 20,09%	2,45% → 20,56%
RAM DB (MB)	831,8 → 949,1	809,8 → 2.518,0

C. *Efisiensi Sumber Daya CPU*

Berbeda dengan pola RAM, konsumsi CPU pada kedua arsitektur relatif seimbang di kedua lapisan server. Pada API server, penggunaan CPU meningkat proporsional terhadap virtual user, dari rata-rata di bawah 5% pada skala small-medium hingga mendekati 50-54% pada skala extreme, dengan multi-database sedikit lebih tinggi karena overhead pengelolaan connection registry (Gbr. 7). Pada database server, kedua arsitektur juga tumbuh searah dari sekitar 2-3% menjadi sekitar 20% pada skala extreme (Gbr. 8), dan single-database bahkan sedikit lebih unggul dalam efisiensi CPU database pada beban puncak, yaitu mampu menghasilkan 8,50 RPS berbanding 8,32 RPS pada multi-database untuk setiap 1% pemakaian CPU. Hal ini menegaskan bahwa perbedaan utama antara kedua arsitektur terletak pada efisiensi memori, bukan pada efisiensi komputasi CPU.



Gbr. 7 Konsumsi CPU API server pada setiap skala pengujian.

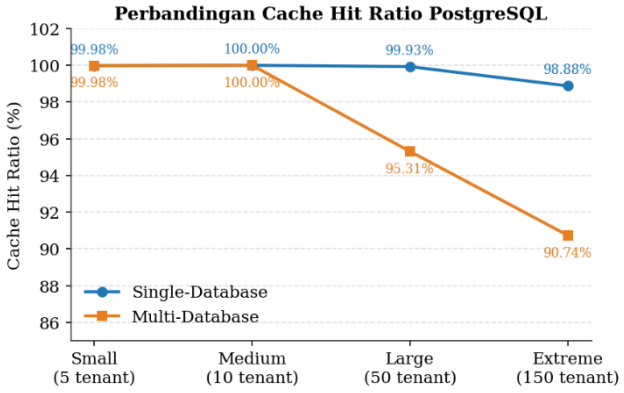


Gbr. 8 Konsumsi CPU database server pada setiap skala pengujian.

D. *Kesehatan Internal Basis Data*

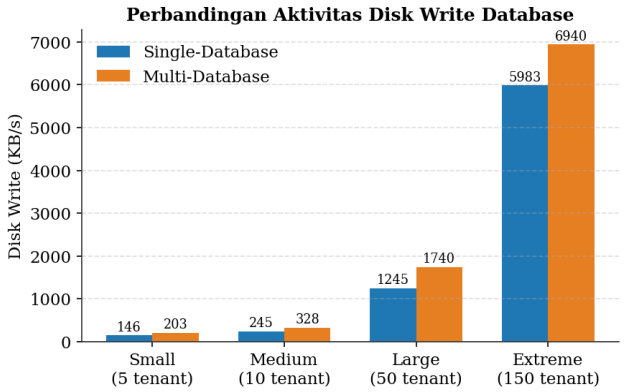
Metrik operasional internal basis data mencakup cache hit ratio serta aktivitas baca/tulis disk (disk I/O). Sepanjang pengujian, tidak ditemukan satu pun insiden deadlock maupun rollback pada kedua arsitektur, yang mengonfirmasi bahwa mekanisme isolasi transaksi berjalan aman. Cache hit ratio PostgreSQL pada multi-database menurun menjadi 90,74% pada skala extreme (dibanding 98,88% pada single-database) akibat fragmentasi shared_buffers ke 150 basis data terpisah, sehingga setiap tenant mendapat porsi buffer yang lebih kecil

dan cache eviction lebih sering terjadi saat beban memuncak [12] (Gbr. 9). Meskipun demikian, efektivitas OS Page Cache berhasil menahan cache miss tersebut sehingga disk read tetap minimal (0,01 KB/s pada skala extreme untuk kedua arsitektur), sehingga penurunan cache hit ratio pada multi-database tidak berdampak signifikan pada latensi aktual.



Gbr. 9 Perbandingan cache hit ratio PostgreSQL pada setiap skala pengujian.

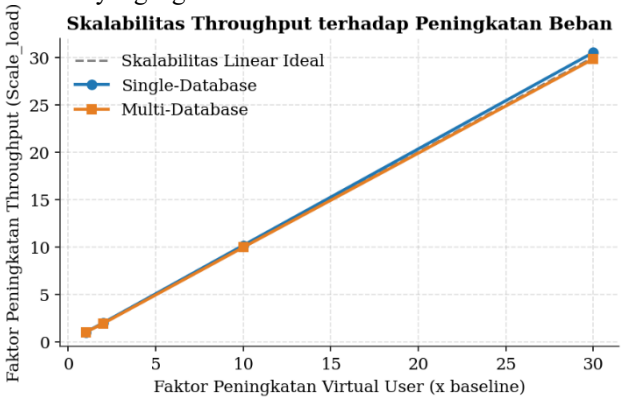
Berbeda dengan disk read, metrik disk write menunjukkan bahwa multi-database secara konsisten menghasilkan beban penulisan 16%-40% lebih tinggi dibanding single-database pada seluruh skala, mencapai 6.940,1 KB/s berbanding 5.982,6 KB/s pada skala extreme (Gbr. 10). Peningkatan ini merupakan konsekuensi logis dari pemisahan fisik basis data, karena setiap dari 150 basis data tenant memiliki mekanisme Write-Ahead Log (WAL) tersendiri yang harus disinkronisasikan (flush) ke disk secara terpisah, berbeda dengan single-database yang menikmati efisiensi batching penulisan WAL secara terpusat.



Gbr. 10 Perbandingan aktivitas disk write database pada setiap skala pengujian.

E. Analisis Skalabilitas

Skalabilitas beban (Scale_load) mengukur seberapa proporsional throughput meningkat ketika jumlah virtual user dikalikan terhadap baseline skala small, dan secara teoretis terbagi menjadi tiga zona: sub-linear (Scale_load < faktor beban), linear (Scale_load = faktor beban), dan super-linear (Scale_load > faktor beban) [8]. Kedua arsitektur mencapai skalabilitas beban yang mendekati linear ideal terhadap peningkatan virtual user; single-database bahkan menunjukkan skalabilitas super-linear pada transisi ke large dan extreme (Scale_load 10,16 dan 30,50 dibanding faktor beban 10x dan 30x), sedangkan multi-database sedikit di bawahnya namun tetap sangat baik (9,95 dan 29,81) akibat overhead pengelolaan koneksi ke 150 basis data (Gbr. 11). Temuan ini membuktikan bahwa arsitektur API server berbasis Go beserta mekanisme connection pool (pgxpool) mampu menangani peningkatan beban hingga 30 kali lipat tanpa mengalami saturasi maupun bottleneck yang signifikan.

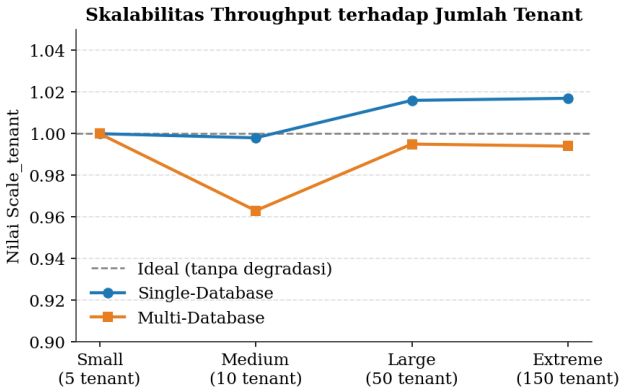


Gbr. 11 Sakalabilitas throughput terhadap peningkatan virtual user, dibandingkan dengan garis linear ideal.

Berbeda dengan pengujian beban yang berfokus pada volume pengguna secara umum, pengujian skalabilitas tenant (Scale_tenant) secara khusus mengukur pengaruh isolasi data terhadap stabilitas throughput rata-rata per tenant ketika jumlah tenant bertambah progresif dari 5 hingga 150. Tabel IV menunjukkan bahwa nilai Scale_tenant kedua arsitektur berada pada rentang 0,963-1,017, membuktikan bahwa lonjakan jumlah tenant tidak menyebabkan degradasi throughput individual yang berarti (Gbr. 12). Single-database bahkan mencatatkan nilai sedikit di atas 1,0 pada skala large dan extreme (1,016 dan 1,017), mengindikasikan efek positif dari connection pool tunggal serta cache PostgreSQL yang semakin optimal (warm cache) seiring tingginya intensitas kueri yang terpusat. Multi-database mengalami penurunan minor pada skala extreme (0,994) akibat overhead komputasi pengelolaan 150 connection pool independen, namun penurunan sebesar 0,6% dari baseline ini tergolong sangat tidak signifikan secara operasional.

TABEL IV
SKALABILITAS TENANT

Skala	RPS/Tenant Single	Scale_tenant Single	RPS/Tenant Multi	Scale_tenant Multi
Small	1,120	1,000	1,148	1,000
Medium	1,118	0,998	1,105	0,963
Large	1,137	1,016	1,142	0,995
Extreme	1,139	1,017	1,141	0,994



Gbr. 12 Nilai Scale_tenant pada setiap skala pengujian dibandingkan dengan garis ideal tanpa degradasi.

F. Evaluasi Trade-Off dan Rekomendasi

Hasil pengujian menunjukkan bahwa pemilihan arsitektur multi-tenant merupakan kompromi antara efisiensi biaya infrastruktur dan isolasi keamanan data. Arsitektur single-database unggul dalam efisiensi sumber daya karena hanya mengelola satu skema dan satu shared_buffers global, namun jaminan privasi data sepenuhnya bergantung pada validitas logika tenant_id di lapisan backend sehingga berisiko terjadi kebocoran data lintas tenant apabila terdapat celah pada kode aplikasi. Skalabilitas lanjutannya juga lebih sesuai melalui pendekatan scale-up karena pemecahan tabel komunal (sharding) membutuhkan rekayasa yang kompleks. Sebaliknya, arsitektur multi-database menawarkan isolasi fisik yang mengeliminasi risiko kebocoran data lintas tenant dan mempermudah backup/restore per tenant, namun menuntut alokasi memori yang jauh lebih besar dan lebih sesuai dipadukan dengan strategi scale-out, yaitu mendistribusikan basis data tenant ke beberapa node server ketika kapasitas satu server telah mencapai batasnya.

Berdasarkan temuan tersebut, arsitektur single-database direkomendasikan bagi platform POS SaaS pada fase awal (startup) atau segmen UMKM dengan keterbatasan modal infrastruktur, dengan syarat penerapan standardisasi pengujian kode (unit testing dan automated query validation) yang ketat untuk mencegah kebocoran data. Sebaliknya, arsitektur multi-database direkomendasikan bagi platform yang menargetkan segmen korporasi (enterprise), waralaba nasional, atau industri dengan regulasi kepatuhan privasi data yang ketat, dengan konsekuensi penyediaan kapasitas RAM server basis data yang jauh lebih besar (8-16 GB) sejak awal operasional.

IV. KESIMPULAN

Penelitian ini membandingkan performa dan skalabilitas arsitektur single-database (shared-schema) dan multi-database pada sistem backend POS multi-tenant menggunakan beban kerja yang mengadaptasi pola TPC-C. Hasil pengujian menunjukkan kedua arsitektur memberikan performa aplikasi yang setara dari sisi pengguna akhir, dengan throughput rata-rata 170 RPS, latensi di bawah 12 ms, dan error rate 0,00% pada beban ekstrem (150 tenant, 1.500 virtual user). Perbedaan mendasar terletak pada efisiensi sumber daya database server, di mana single-database sekitar 2,65 kali lebih hemat memori (949,1 MB dibanding 2.518 MB pada skala extreme) dan mampu mempertahankan cache hit ratio 98,88% dibanding 90,74% pada multi-database, meskipun multi-database tetap stabil berkat efektivitas OS page cache. Kedua arsitektur juga terbukti skalabel hampir linear terhadap peningkatan beban maupun jumlah tenant. Berdasarkan temuan tersebut, arsitektur single-database direkomendasikan bagi platform POS SaaS skala startup dan UMKM yang mengutamakan efisiensi biaya infrastruktur, sedangkan arsitektur multi-database lebih sesuai bagi platform berskala enterprise yang mewajibkan isolasi dan kepatuhan privasi data secara mutlak, dengan konsekuensi investasi kapasitas RAM server yang jauh lebih besar. Penelitian lanjutan dapat mengeksplorasi pendekatan hybrid (shared-database separate-schema) serta strategi optimasi connection pooling untuk menekan overhead memori pada arsitektur multi-database.

REFERENSI

[1] H. Asrani, S. Vishwakarma, D. Asrani, and Dr. K. Asrani, “Point of Sale Systems,” Innovative Research Publication, Apr. 2024, pp. 358–363. doi: 10.55524/csistw.2024.12.1.63.

[2] I. Jaka Perdana, E. Prayitno, E. Iskandar, and A. A. Subagyo, “Prosiding Seminar Nasional Aplikasi Sains & Teknologi (SNAST) 2024 Yogyakarta,” 2024.

[3] R. K. Ritesh Kumar, “Multi-Tenant SaaS Architectures: Design Principles and Security Considerations,” *Journal of Software Engineering and Simulation*, vol. 6, no. 5, pp. 28–41, May 2020, doi: 10.35629/3795-06052841.

[4] J. Ni, G. Li, L. Wang, J. Feng, J. Zhang, and L. Li, “Adaptive Database Schema Design for Multi-Tenant Data Management,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 9, pp. 2079–2093, Sep. 2014, doi: 10.1109/TKDE.2013.94.

[5] S. K. Pippal, S. Kumar, and R. Rani, “Optimizing multi-tenant database architecture for efficient software as a service delivery,” *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 22, no. 5, pp. 1128–1137, Oct. 2024, doi: 10.12928/TELKOMNIKA.v22i5.26385.

[6] E. José Pereira de Sousa, F. Araújo, and P. Furtado, “Analysis of Multi-Tenancy in Data Warehouse Databases,” 2023.

[7] T. Taipalus, “Database management system performance comparisons: A systematic literature review,” *Journal of Systems and Software*, vol. 208, Feb. 2024, doi: 10.1016/j.jss.2023.111872.

[8] P. Lake and P. Crowther, “Database Scalability,” in *Concise Guide to Databases*, London: Springer, 2013, ch. Database Scalability, pp. 201–219. doi: 10.1007/978-1-4471-5601-7_9.

[9] S. K. Agrawal and T. Aswini, “Multi-Tenant Low Latency Scalable Architectures for Large-Scale Customer Data Processing,” *International Journal for Research Publication and Seminar*, vol. 16, pp. 2278–6848, 2025, doi: 10.36676/jrps.v16.i1.1641.

[10] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, “OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases,” 2013. [Online]. Available: http://dag.wieers.com/home-made/dstat/

[11] G. B. Pallavi and P. Jayarekha, “An efficient resource sharing technique for multi-tenant databases,” *International Journal of Electrical and Computer Engineering*, vol. 10, no. 3, pp. 3216–3226, 2020, doi: 10.11591/ijece.v10i3.pp3216-3226.

[12] A. Shah and N. Bhatt, “A systematic review of in-memory database over multi-tenancy,” 2024, *Institute of Advanced Engineering and Science*. doi: 10.11591/ijece.v14i2.pp1720-1729.