Jurnal Elektronik Ilmu Komputer Udayana
Volume 13, No 1. August 2024

p-ISSN: 2301-5373
e-ISSN: 2654-5101

Pengolahan Big Data Dengan *Sharding Database* Dan *Kappa Architecture* Untuk *Data Time-Series*

Kompiang Gede Sukadharma[a1], I Putu Gede Hendra Suputra[a2], Ida Ayu Gde Suwiprabayanti Putra[a3], Luh Arida Ayu Rahning Putri[a4]

[a]Program Studi Informatika, Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Udayana Kuta Selatan, Badung, Bali, Indonesia
[1]kompiang.sukadharma@gmail.com
[2]hendra.suputra@unud.ac.id
[3]iagsuwiprabayantiputra@unud.ac.id
[4]rahningputri@unud.ac.id

Abstract

In the digital era, managing and processing Big Data presents challenges. Historical or time-series data is crucial for decision-making. Previous research has indicated that databases on a server become increasingly irrelevant over time for handling continuous and ever-growing data. Distributed database systems, specifically database sharding, efficiently distribute CPU load and memory usage. Kappa Architecture outperforms Lambda Architecture by 220% in terms of speed, though Lambda has 9% higher reliability rate than Kappa Architecture. This research integrates database sharding with Kappa Architecture using time-series data obtained from the Kaggle platform. In testing results using a load-testing method. Load testing showed 60.46% performance for data retrieval across distributed databases. Remarkably, the system's reliability reached 100%, even when one of the services failed, handling 10000 new data entries. Furthermore, with the same configuration, it was found that the utilization of Kappa Architecture and sharding database resulted in a 70.26% better performance compared to the system solely implementing sharding database.

Keywords: *Kappa Architecture, Time-Series Data, Sharding Database, Big Data Processing, Reliability, Performance*

1. Pendahuluan

Setiap detik, lebih dari 30000 gigabytes data baru dihasilkan, dan laju pertumbuhan produksi data ini diperkirakan akan terus meningkat [1]. Karena jumlah data yang tercipta sangat besar, sistem yang mampu mengolah data dengan volume dan kecepatan seperti ini seringkali dianggap sebagai sistem pengolah *big data*. Pemanfaatan data yang melimpah ini sangat berharga untuk menggali informasi yang berguna untuk mengoptimalkan strategi pengambilan keputusan [2]. Terdapat beberapa jenis data yang tersedia, *data time-series* menjadi salah satu sumber informasi kunci untuk mendukung pengambilan keputusan. Di era saat ini, salah satu tantangan besar dalam penelitian *big data* adalah mengolah *data time-series* yang jumlahnya sangat banyak. Ini karena *database* relasional konvensional sudah tidak lagi memadai untuk menangani *data time-series*, akibat dari keterbatasan dalam kecepatan pemrosesan saat volume data bertambah, redundansi data, dan kurangnya dukungan terhadap operasi pada data bertipe waktu. Hal ini menyebabkan proses pengolahan data menjadi tidak efisien [3].

Untuk menangani *big data*, yang dicirikan oleh volume, kecepatan, dan keragaman data yang besar, dibutuhkan sistem basis data khusus yang mampu menyimpan dan mengelola data dalam skala besar. Pendekatan penyimpanan data terpusat dapat diadaptasi untuk memenuhi kebutuhan skalabilitas, keandalan, dan pengelolaan data yang efisien. Namun, menghadapi tantangan untuk menciptakan sistem basis data yang efektif biaya dan berkinerja tinggi, solusi yang ditemukan adalah penerapan basis data terdistribusi melalui teknik *sharding database*. *Sharding database* memungkinkan pemecahan dan pendistribusian *database* menjadi segmen-segmen lebih kecil ke berbagai *node database* dalam sebuah kluster, dimana setiap segmen beroperasi secara mandiri, menawarkan solusi skalabilitas dan efisiensi [4].

Dalam pemrosesan *big data*, dua pendekatan utama yang digunakan adalah pemrosesan *batch* dan pemrosesan *stream*. Pemrosesan *batch* dikenal dengan kinerjanya yang solid namun memiliki latensi yang tinggi, yang membuatnya kurang ideal untuk aplikasi data *real-time*. Di sisi lain, pemrosesan *stream* menawarkan latensi yang rendah dan ketersediaan yang tinggi tetapi bisa berdampak pada penurunan tingkat akurasi. Karenanya, untuk mencapai hasil yang optimal dalam berbagai skenario, seringkali diperlukan gabungan dari kedua metode tersebut agar dapat memperoleh proses yang cepat namun tetap detail [5]. Dua arsitektur utama yang digunakan dalam implementasi metode pemrosesan ini adalah *lambda architecture* dan *kappa architecture*. *Lambda architecture* mendukung pemrosesan secara *real-time* dan *batch*, tetapi menimbulkan tantangan dalam hal biaya dan kompleksitas sistem karena memerlukan pemeliharaan dua teknologi yang berbeda [6]. Sebaliknya, *kappa architecture*, yang lebih sesuai untuk menangani data besar yang perlu diproses secara cepat dan *real-time* seperti data dari sensor, robot, dan jaringan, menonjolkan pemrosesan melalui jalur yang sama untuk *real-time* dan *micro batch*ing. Keuntungan menggunakan *kappa architecture* adalah hanya membutuhkan satu teknologi pemrosesan *real-time* untuk menangani kedua jenis pemrosesan data tersebut [7].

Oleh karena itu, sangat penting untuk melakukan penelitian yang lebih mendalam. Penelitian ini akan berfokus pada pengembangan teknik dan struktur yang mendukung keberlangsungan dan optimalisasi ekstraksi informasi dari *data time-series*. Hasil dari penelitian ini diharapkan dapat menyediakan wawasan terkait efisiensi pemrosesan *big data* dengan data berjenis *time-series* menggunakan teknik *sharding database* dan *kappa architecture*, yang keduanya berpotensi meningkatkan kinerja dan keandalan sistem. Kinerja, dalam konteks ini, merujuk pada kemampuan sistem untuk merespons permintaan.

2. Metodologi Penelitian

Penelitian ini penulis bagi ke dalam lima tahap, yaitu identifikasi masalah, pengumpulan data, perancangan kebutuhan sistem, dan perancangan pengujian dan evaluasi sistem.

2.1. Identifikasi Masalah

Isu yang dibahas dalam penelitian ini muncul dari kebutuhan sebuah perusahaan pengembang perangkat lunak yang ingin menghasilkan laporan berbasis waktu dengan cepat dan biaya yang efisien. Perusahaan ini menghadapi kendala dalam mengakses data dari *database* berukuran besar. Terutama, *data time-series* yang menjadi fokus dalam penelitian ini. Seiring waktu, *data time-series* akan bertambah banyak yang menyebabkan peningkatan waktu akses data serta konsumsi memori dan CPU pada server [4]. Hal ini tidak hanya memperlambat proses pengambilan data tapi juga berdampak pada performa *service* lain yang bergantung pada *database* yang sama. Penyebab utamanya adalah sistem dari operasi *database* pada server yang harus mengiterasi lebih banyak data, karena bertambahnya volume data, mengakibatkan penurunan kinerja server.

Berdasarkan analisis masalah ini, yang didukung oleh referensi penelitian lain, penulis memilih untuk mengeksplorasi penggunaan metode *sharding database* bersamaan dengan penerapan arsitektur kappa. Tujuannya adalah untuk mengolah *big data*, khususnya *data time-series*, melalui stream processing. Pendekatan ini diharapkan dapat meningkatkan kecepatan operasi *database* besar dan secara simultan menurunkan biaya operasional *database* bagi perusahaan.

2.2. Pengumpulan Data

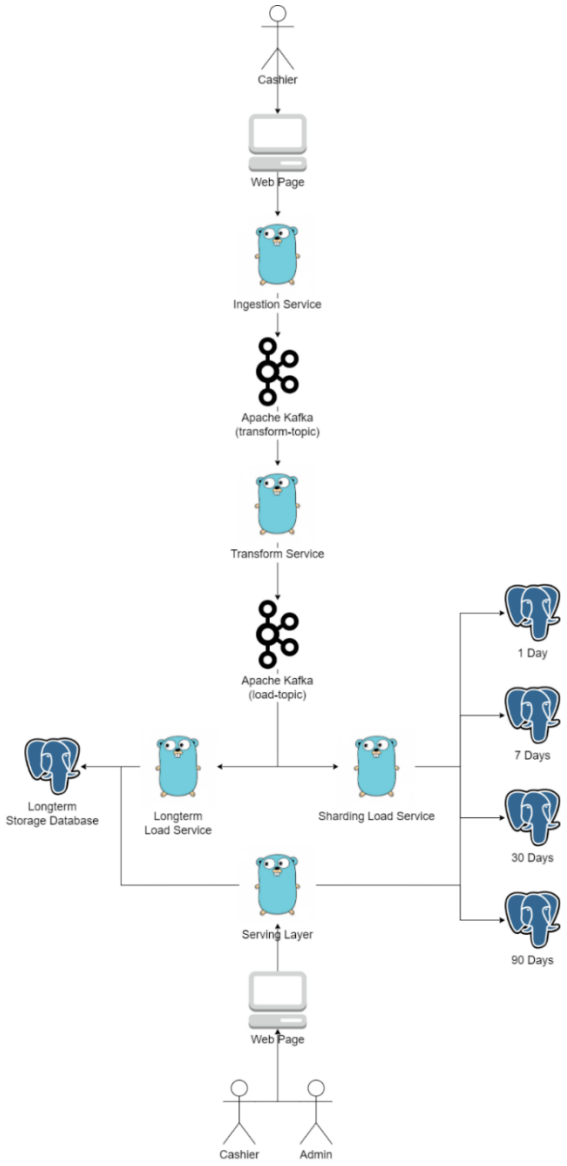
Penulis kemudian beralih ke tahap pengumpulan data untuk penelitian ini. Menggunakan *Dataset* dari Kaggle sebagai sumber data, penulis memilih pendekatan ini karena keterbatasan akses ke *data time-series* asli dari perusahaan yang bersifat rahasia dan tidak dapat dibagikan secara publik. *Dataset* yang dipilih mengandung sebuah tabel dengan 1 juta baris data yang berkaitan dengan *time-series*. Untuk lebih mendekati skenario *big data* yang sesungguhnya, data ini akan direplikasi sebanyak 10 kali, menggambarkan aspek volume yang besar, salah satu karakteristik penting dari *big data*. *Dataset* yang penulis peroleh berjudul “ECommerce Data Analysis” yang diunggah oleh M MOHAIMINUL ISLAM pada platform Kaggle. Data yang terdapat pada *dataset* ini adalah data dari *ecommerce* yang terletak pada negara Bangladesh. Oleh karena itu, data *time-series* yang penulis gunakan merupakan data order.

p-ISSN: 2301-5373
e-ISSN: 2654-5101

2.3. Perancangan Kebutuhann Sistem

2.3.1. Perancangan High-Level Kappa Architecture

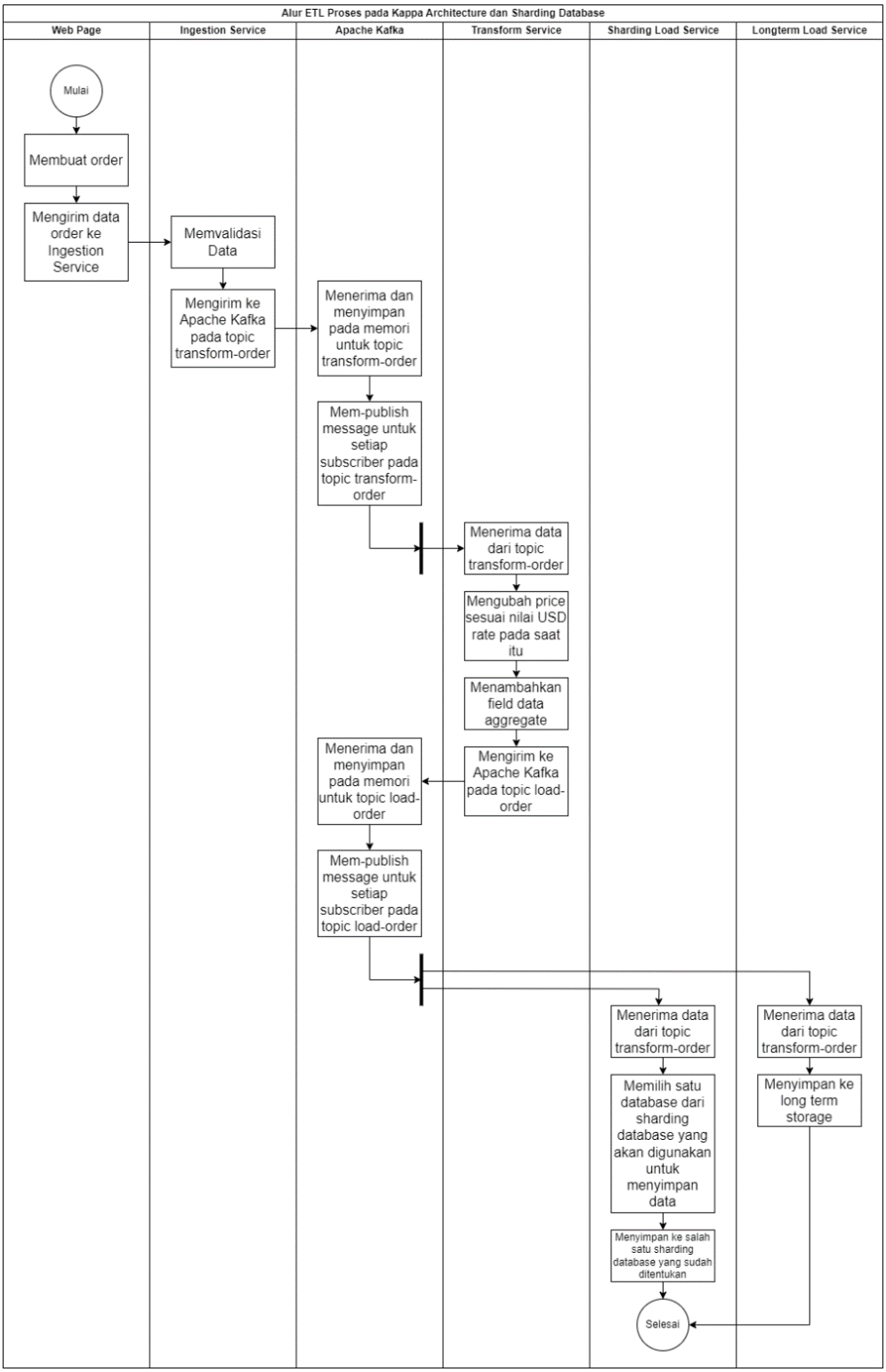
Perancangan *High-Level Arsitektur Kappa*, dapat dilihat pada Gambar 1. *Entrypoint* dari sistem ini terletak pada ingestion service, *service* ini akan mendapatkan data yang dikirim dari web page via API. Kemudian terdapat Apache Kafka yang memiliki peran sebagai *message broker* untuk setiap *publisher* dan *subscriber* yang ada pada sistem ini.



Gambar 1. *High-Level Kappa Architecture*

Kappa architecture merupakan sebuah pemrosesan *big data* dengan metode *stream processing* pada sebuah jalur dan juga implementasi *pipeline pattern*. Jadi, sistem ini merupakan sekumpulan proses yang akan mengirimkan data dari satu *service* ke *service* lainnya menggunakan *kappa architecture* untuk mengimplementasikan *event-driven microservices*. Pada *kappa architecture* yang penulis teliti juga terdapat Postgres yang menjadi SQL *database* dan menjadi *storage layer* dari arsitektur ini. Kemudian juga terdapat *serving layer* yang diimplementasikan menggunakan bahasa pemrograman Go dan menyediakan API yang dapat digunakan untuk mengambil data dari *database* yang tersedia.

2.3.2. Perancangan Algoritma dan Alur Untuk Melakukan Penulisan ke Storage Layer

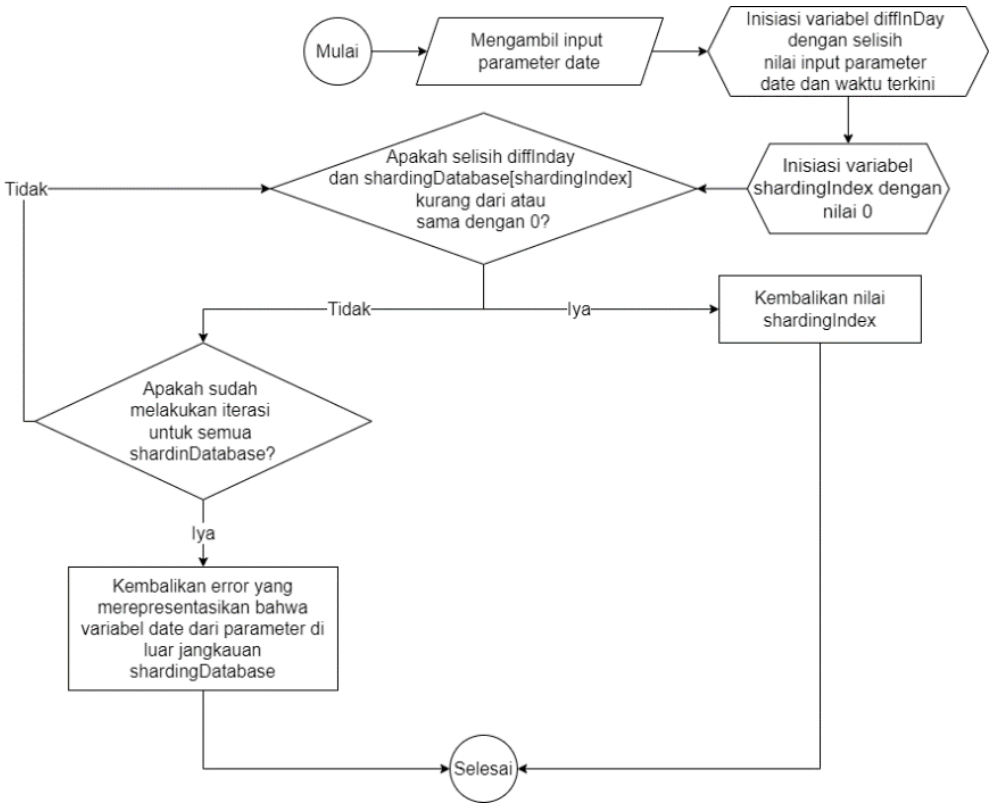


Gambar 2. *Activity Diagram* Alur Pembuatan Data Time-Series

Penulis menjelaskan alur proses pembuatan order dari data *order* yang masuk ke *ingestion service* hingga penyimpanan data pada *sharding load service* dan *longterm load service* dan ditunjukkan pada *activity diagram* pada Gambar 2. *Ingestion service* berperan dalam ekstraksi dan validasi data sebelum dipublikasikan ke Apache Kafka. *Transform service* menambahkan

p-ISSN: 2301-5373
e-ISSN: 2654-5101

nilai mata uang, melakukan konversi, dan melakukan agregasi data sebelum mempublikasikannya kembali ke Apache Kafka untuk disimpan pada *sharding load service* dan *longterm load service*. Proses *sharding load service* mencari *database* yang tepat untuk menyimpan data, sedangkan *longterm load service* langsung menyimpan data ke *database* tanpa proses tambahan. Algoritma yang dibuat untuk pemilihan *sharding database* mempertimbangkan rentang data retensi pada setiap *database*. Jika data tidak sesuai dengan rentang semua *sharding database*, data tersebut akan disimpan pada *longterm database*.

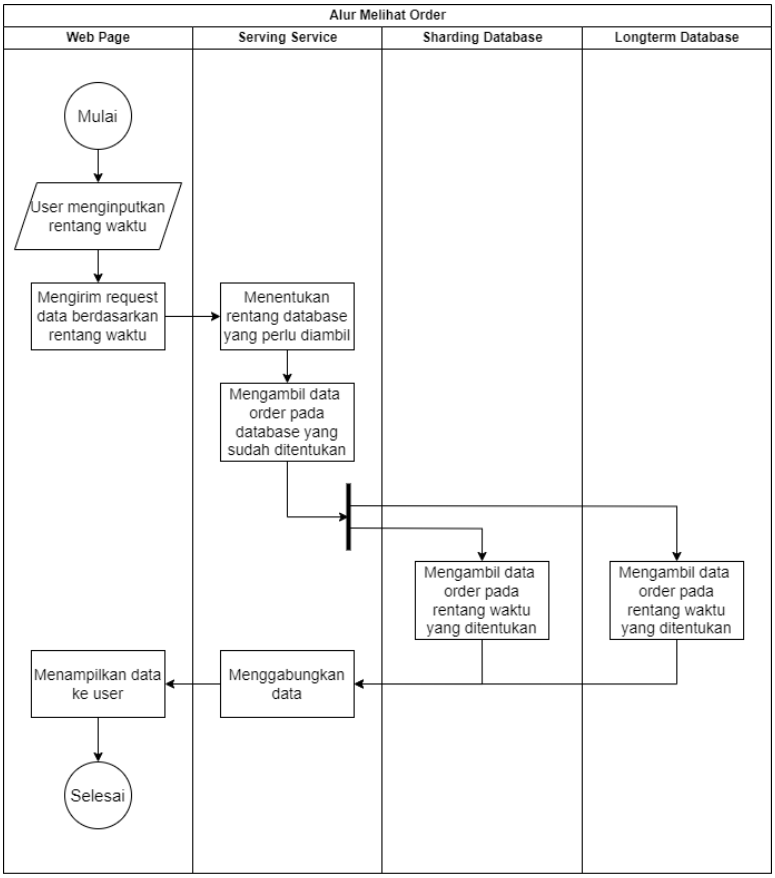


Gambar 3. Alur Algoritma Penentuan Sharding Database Untuk Sebuah Data Time-Series

Karena penelitian ini menggunakan *sharding database,* penulis perlu membuat sebuah algoritma yang akan menentukan *sharding database* indeks yang diperlukan. Algoritma pada Gambar 3 akan meminta sebuah *input parameter* yang merupakan sebuah data yang memiliki tipe data *date*. Setelah itu, nilai dari *parameter* tersebut akan dihitung selisih hari dengan waktu terkini. Kemudian, algoritma akan melakukan perulangan pada konfigurasi dari kumpulan *sharding database* yang merupakan sebuah *array.* Setiap perulangan, algoritma akan menghitung selisih data retensi dari *sharding database* tersebut dengan selisih nilai parameter dengan waktu terkini. Apabila selisih tersebut memiliki nilai 0 atau kurang dari 0, artinya data tersebut berada pada rentang *sharding database* tersebut. Setelah itu, algoritma akan mengembalikan nilai indeks dari konfigurasi *sharding database* yang akan digunakan untuk membuat koneksi ke *sharding database* tersebut. Namun, apabila sudah melakukan perulangan ke semua *sharding database* dan tidak ada selisih yang memiliki nilai 0 atau kurang dari 0, algoritma akan mengembalikan nilai *error* yang merepresentasikan kondisi data tersebut tidak berada dalam rentang dari semua *sharding database*.

2.3.3. Perancangan Algoritma dan Alur Untuk Melakukan Pembacaan Dari Serving Layer

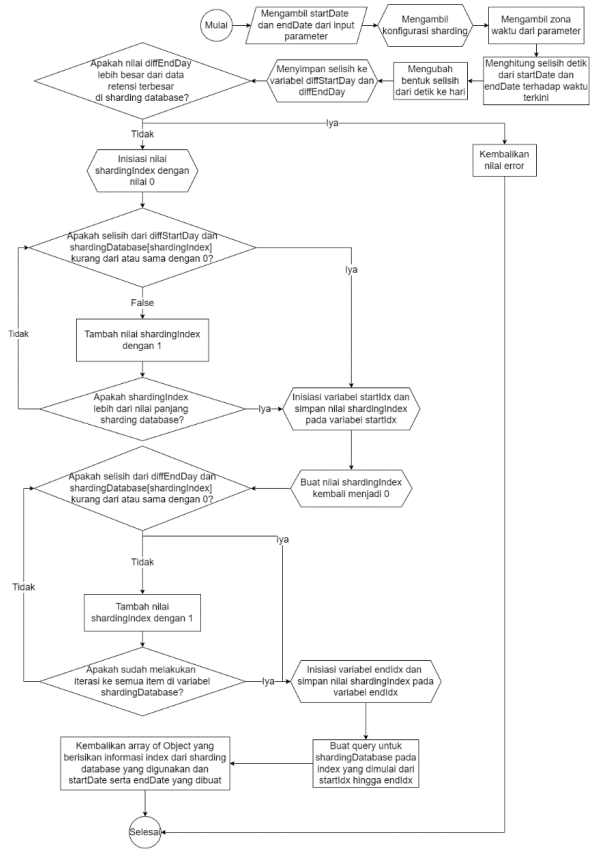
Alur dalam pembacaan data dapat dilihat pada Gambar 4. Secara umum, pengguna memasukkan rentang waktu pada *web page*, lalu request dikirim ke *serving service*. *Serving service* menggunakan sebuah algoritma untuk menentukan rentang *database*, kemudian mengambil data dari setiap rentang *database*, menggabungkannya, dan menampilkan hasilnya pada *web page*.



Gambar 4. *Activiy Diagram* Untuk Membaca Data Dari *Serving Layer*

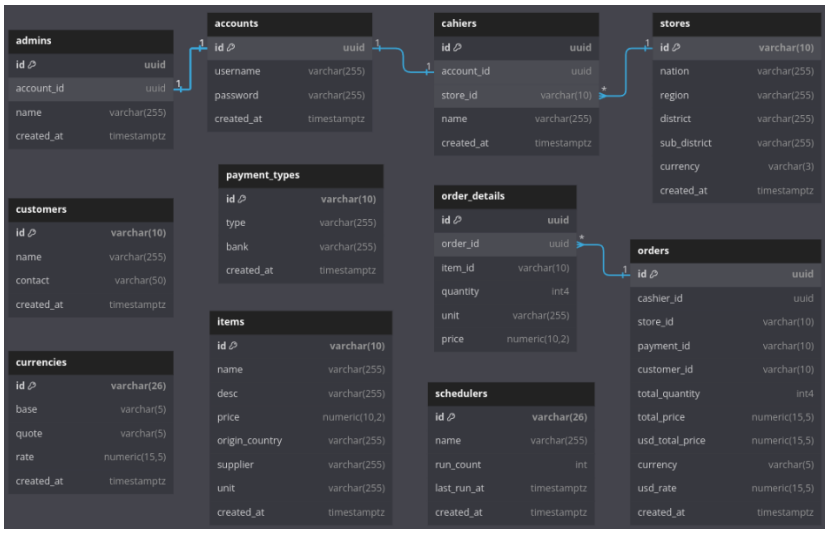
Perancangan alur dari algoritma untuk pembacaan data dari beberapa *sharding database* dapat dilihat pada Gambar 5. Dengan menggunakan algoritma tersebut, sistem dapat dengan efisien menentukan dan memilih *database* yang tepat untuk mengambil data yang diperlukan, sehingga meminimalkan beban dan meningkatkan kinerja secara keseluruhan. Algoritma tersebut beroperasi dengan langkah-langkah sebagai berikut, dimulai dengan validasi terhadap parameter rentang waktu yang dimasukkan oleh pengguna. Validasi ini penting untuk memastikan bahwa *query* yang akan dilakukan tidak mengakses semua *database* jika tidak diperlukan, sehingga menghindari beban yang tidak perlu. Setelah itu, algoritma memeriksa urutan tanggal awal dan akhir, dan jika tanggal akhir lebih awal dari tanggal awal, sistem akan memberikan pesan *error* untuk memperingatkan pengguna. Langkah selanjutnya adalah menyesuaikan zona waktu dari parameter *input* dengan konfigurasi sistem, sehingga *query* yang dihasilkan akan sesuai dengan zona waktu yang relevan. Algoritma kemudian menghitung perbedaan hari antara tanggal awal dan tanggal akhir, yang menjadi dasar untuk menentukan rentang *database* yang akan digunakan. Proses perulangan dilakukan untuk setiap *sharding database* yang terdaftar dalam konfigurasi sistem. Setiap iterasi perulangan mencari perbedaan antara data retensi dari *sharding database* dan selisih hari dari tanggal awal dan akhir dengan waktu saat ini. Jika perbedaannya kurang dari atau sama dengan nol, itu menandakan bahwa *sharding database* tersebut akan menjadi bagian dari rentang yang diperlukan. Proses ini diulangi untuk menentukan rentang akhir dari *sharding database* yang akan digunakan. Setelah rentang *sharding database* ditentukan, *query* yang sesuai dibuat untuk mengambil data dari rentang waktu yang ditentukan. Terakhir, algoritma mengembalikan *array* yang berisi informasi tentang indeks dari *sharding database* terpilih bersama dengan *query* yang dibuat, sehingga sistem dapat dengan mudah mengambil data yang diperlukan dari *database* yang sesuai dengan rentang waktu yang diminta. Dengan demikian, algoritma ini tidak hanya memastikan efisiensi dalam penggunaan sumber daya, tetapi juga memberikan fleksibilitas dan akurasi dalam pengambilan data yang sesuai dengan kebutuhan pengguna.

p-ISSN: 2301-5373
e-ISSN: 2654-5101



Gambar 5. Alur Algoritma Penentuan Rentang *Sharding Database*

2.3.4. Perancangan Skema Database



Gambar 6. Skema Database

Dalam penelitian yang dilakukan, sistem yang dirancang memanfaatkan tiga jenis *database* yang berbeda, yang masing-masing diadaptasi untuk fungsi tertentu. Pemisahan ini merupakan bagian dari penerapan metode *sharding database* dan arsitektur *microservices*, yang bertujuan untuk memaksimalkan efisiensi dan skalabilitas sistem. Oleh karena itu, perencanaan skema *database* menjadi langkah krusial dalam proses implementasi, di mana harus ditentukan bagaimana

masing-masing *database* akan diorganisir dan diintegrasikan dengan bagian lain dari sistem. Pendekatan ini memungkinkan untuk pemrosesan data yang lebih cepat dan lebih efisien, sekaligus memudahkan pengelolaan data yang berskala besar dan kompleks.

Pada gambar Gambar 6, tabel admins, accounts, cashiers, stores, schedulers, payment_types, items, customers, dan currencies akan berada pada general *database*. Hal ini dilakukan, karena tabel-tabel tersebut memiliki data yang bersifat non-transaksional. Untuk tabel orders dan order_details akan tersimpan pada tiap-tiap indeks pada *sharding database* dan juga tersimpan pada *longterm database*.

2.4. Perancangan Pengujian dan Evaluasi Sistem

Dalam fase evaluasi dari sistem, peneliti memutuskan untuk mengimplementasikan *load-testing* menggunakan k6 untuk mengukur performa dan reliabilitas sistem yang dikembangkan. Proses pengujian ini dibagi menjadi dua bagian utama untuk menguji aspek-aspek kritis dari sistem,

1. Pengujian Performa Pengambilan Data *Time-Series*:
   Bagian kedua berfokus pada efektivitas sistem dalam mengambil atau meng-query *data time-series* yang besar dari *database*. Ini bertujuan untuk menilai seberapa cepat sistem dapat mengakses dan menyajikan data yang diminta. Pengujian ini akan membandingkan sistem ketika tidak menggunakan *sharding database* dan menggunakan *sharding database*.
2. Pengujian Reliabilitas
   Bagian pertama dari pengujian fokus pada kemampuan sistem untuk menangani data yang masuk ketika salah satu *stream processor service* dalam kappa architecture mati, pada penelitian ini, penulis mematikan *transform service*. Pengujian ini bertujuan untuk melihat berapa banyak data yang hilang.
3. Pengujian Performa Penulisan Data *Time-Series*:
   Bagian ketiga berfokus pada efektivitas sistem dalam menulis data *time-series* yang memiliki kecepatan yang tinggi dan jumlah yang banyak. Pengujian ini bertujuan untuk menilai seberapa cepat sistem dalam menangani penulisan data.

Pengujian reliabilitas dan pengujian performa penulisan data *time-series* akan membandingkan sistem ketika menggunakan *kappa architecture* dan *sharding database* dengan sistem yang tidak menggunakan *kappa architecture*, tetapi menggunakan *sharding database*. Untuk sistem yang tidak menggunakan *kappa architecture* pada pengujian ini, setiap *service* akan saling berkomunikasi secara langsung menggunakan protokol HTTP dan tidak menggunakan Apache Kafka.

Hasil dari pengujian akan menjadi kunci untuk mengevaluasi performa keseluruhan dan keandalan dari sistem yang dibangun. Adapun beberapa metris yang digunakan untuk melakukan pengukuran dalam pengujian ini. Metris-metris yang digunakan dapat dilihat pada Tabel 1.

Tabel 1. Metris Pengujian

Nama Metris	Satuan Ukur	Deskripsi
http_req_waiting	milliseconds	Seberapa lama menunggu respon balik dari awal request. Dapat pula dikatakan dengan "time to first byte"
http_req_failed	persentase	Seberapa banyak failed rate atau rata - rata request yang gagal

Dalam analisis performa sistem menggunakan *load-testing*, metris *http_req_waiting* menjadi kunci untuk memahami waktu yang dibutuhkan untuk menunggu respons dari server. Untuk memastikan analisis yang akurat dan menghindari distorsi yang mungkin diakibatkan oleh *data outlier*, persentil ke-95 digunakan dalam evaluasi. Pendekatan ini membantu dalam menyaring data yang tidak representatif dengan mengesampingkan 5% latensi teratas, yang dapat memberikan gambaran lebih jelas mengenai performa sistem dalam kondisi normal. Terdapat penelitian sebelumnya yang menggunakan persentil ke-99 untuk mendapatkan gambaran yang lebih dekat dengan kondisi ekstrem dimana latensi tertinggi menandakan potensi masalah dalam sistem [8].

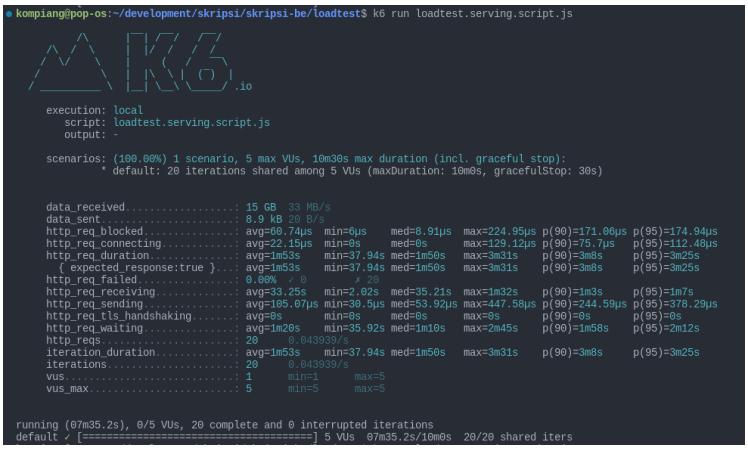
p-ISSN: 2301-5373
e-ISSN: 2654-5101

Sementara itu, metris *http_req_failed* mengukur keandalan sistem dengan menghitung jumlah *request* yang gagal, yang ditandai dengan *status code* HTTP kurang dari 300. Metris ini memberikan *insight* tentang seberapa sering pengguna mengalami kegagalan saat berinteraksi dengan sistem, sehingga dapat menjadi indikator langsung dari kualitas *service* yang diberikan. Menggunakan kedua metris ini bersama-sama memberikan gambaran komprehensif tentang performa dan reliabilitas sistem, mulai dari seberapa cepat sistem dapat merespons permintaan hingga seberapa sering permintaan tersebut gagal.

3. Hasil dan Diskusi

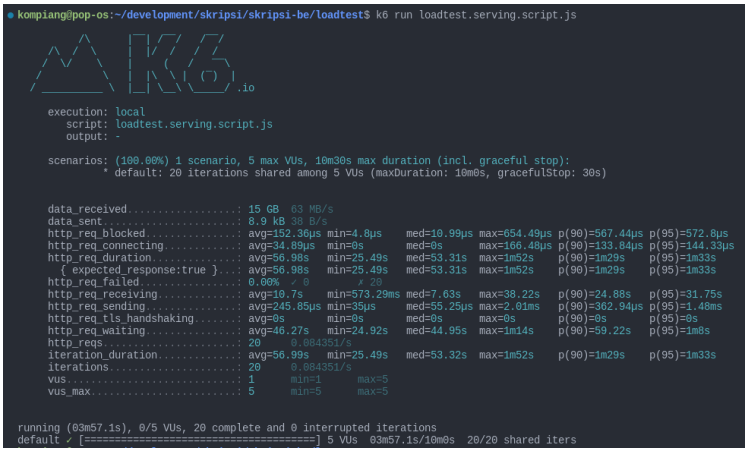
3.1. Hasil Pengujian Performa Pengambilan Data Time-Series

Dalam skenario pengujian yang tidak menggunakan metode *sharding* untuk pengambilan data yang dapat dilihat pada Gambar 7, pengujian dilakukan dengan 5 pengguna virtual dan 20 iterasi, memakan waktu total 7 menit dan 35 detik. Dari hasil pengujian, tercatat pada metris *http_req_duration* nilai tertinggi yang dicapai adalah 3 menit 31 detik, dengan nilai untuk persentil ke-95 adalah 3 menit 25 detik. Untuk metris *http_req_waiting*, nilai tertinggi yang tercatat adalah 2 menit 45 detik, sedangkan untuk persentil ke-95, nilai yang dicapai adalah 2 menit 12 detik. Adapun metris *http_req_failed*, menunjukkan nilai 0%, yang berarti tidak ada kegagalan yang terjadi selama proses pengujian. Ini mengindikasikan bahwa meskipun tanpa *sharding*, sistem mampu menangani permintaan tanpa mengalami kegagalan, namun dengan waktu tunggu yang signifikan.



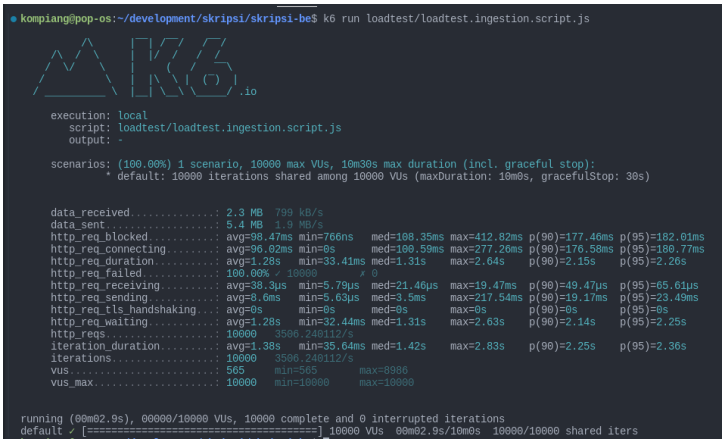
Gambar 7. Hasil Pengujian Performa Sistem Tanpa Menggunakan Metode *Sharding* Database

Hasil pengujian performa sistem menggunakan metode *sharding database* pada dilihat pada Gambar 8. Dengan penggunaan *sharding*, membutuhkan waktu total 3 menit 57 detik untuk 5 *virtual user* dan 20 iterasi. Metris *http_req_duration* menunjukkan nilai tertinggi 1 menit 52 detik dan pada persentil ke-95 adalah 1 menit 33 detik. Untuk *http_req_waiting*, nilai tertinggi adalah 1 menit 14 detik dan pada persentil ke-95 adalah 1 menit 8 detik. Sama seperti skenario pertama, metris *http_req_failed* tetap menunjukkan 0%. Perbandingan antara kedua skenario ini menunjukkan bahwa penggunaan *sharding database* secara signifikan meningkatkan kecepatan pengambilan data, dengan pengurangan waktu tunggu yang signifikan. Peningkatan kecepatan ini diperoleh dari pembagian *data time-series* ke berbagai *database*, memungkinkan proses pencarian dan pengambilan data menjadi lebih efisien karena setiap *database* hanya mengelola jumlah *row* yang lebih sedikit.



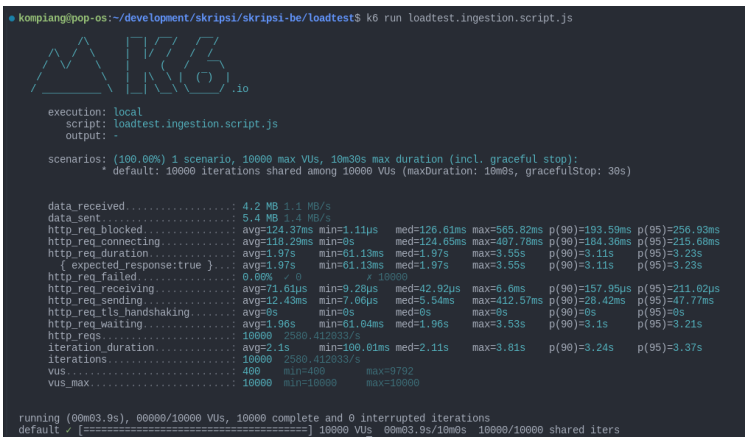
Gambar 8. Hasil Pengujian Performa Sistem Menggunakan Metode *Sharding* Database

3.2. Hasil Pengujian Reliabilitas Sistem



Gambar 9. Hasil Pengujian Reliabilitas Sistem yang Tidak Menggunakan *Kappa Architecture*

Dapat dilihat pada Gambar 9 yang merupakan hasil dari pengujian *reliabilitas* dari sistem yang tidak menggunakan *kappa architecture*. Ketika kita membuat order dengan kondisi salah satu *stream processor sevice* mati dan tidak menggunakan *kappa architecture*, menghasilkan nilai metris *http_req_failed* dengan nilai 100% atau semua *request* gagal dikirim dan tidak ada data yang berhasil diproses.

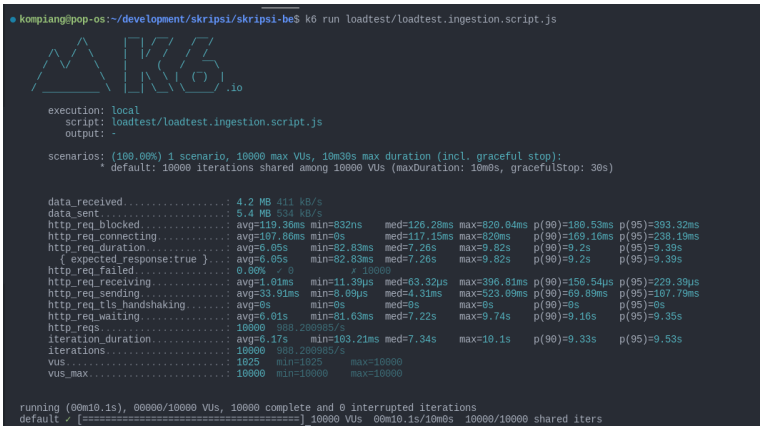


Gambar 10. Hasil Pengujian Reliabilitas Sistem yang Menggunakan *Kappa Architecture*

p-ISSN: 2301-5373
e-ISSN: 2654-5101

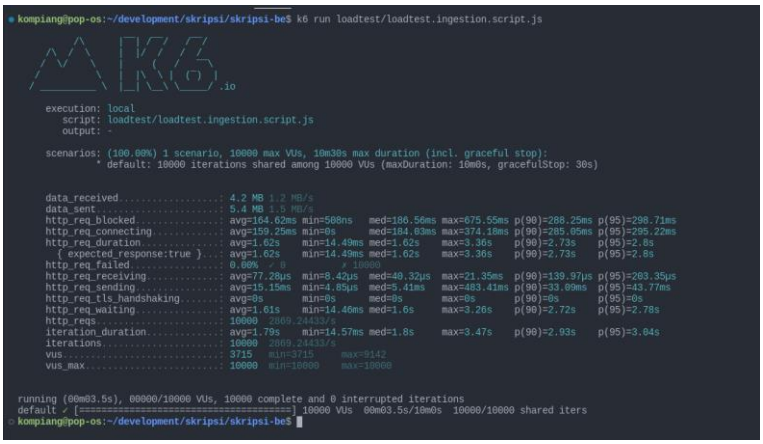
Dapat dilihat pada Gambar 10 pengiriman tidak mendapatkan *error* apapun. Hal ini dapat dilihat pada metris *http_req_failed* yang menghasilkan nilai 0% yang memberikan informasi, bahwa semua *request* sukses dikirimkan dan tidak ada *error*, meskipun *transform service* dalam kondisi mati. Hal ini sangat penting dalam keandalan sistem, karena kita tidak ingin pengguna mendapatkan masalah karena sistem tidak bisa melakukan pembuatan order. Hal ini juga dimungkinkan karena data disimpan secara sementara dalam *message broker*. Setelah *stream processor system* diaktifkan kembali, semua data yang tersimpan di *message broker* berhasil dipulihkan dan disimpan kembali ke *longterm database* dan *sharding database*, tanpa ada kehilangan data.

3.3. Hasil Pengujian Performa Penulisan Data Time-Series



Gambar 11. Hasil Pengujian Performa Penulisan Data Time-Series Dengan Tidak Menggunakan Kappa Architecture Tetapi Menggunakan Sharding Database

Hasil pengujian performa penulisan data *time-series* dengan tidak menggunakan *kappa architecture,* tetapi menggunakan *sharding database* dapat dilihat pada Gambar 11. Pada pengujian ini dilakukan pembuatan 10000 data oleh 10000 *virtual users*. Hasil dari *load testing* pada metris *http_req_waiting* menghasilkan nilai 9,35 detik pada persentil ke-95 dan metris *http_req_duration* memiliki nilai 0%, artinya semua *request* berhasil terkirim.



Gambar 12. Hasil Pengujian Performa Penulisan *Data Time-Series* Dengan Menggunakan *Kappa Architecture* dan Menggunakan *Sharding Database*

Hasil pengujian pada skenario pengujian performa penulisan data *time-series* menggunakan *kappa architecture* dan *sharding database* dapat dilihat pada Gambar 12. Pada skenario ini, dilakukan pembuatan 10000 data *time-series* oleh 10000 *virtual users* yang menggunakan *kappa*

architecture dan *sharding database* dalam pembuatan order. Hasil dari *load testing* pada skenario 5 pada metris *http_req_waiting* menghasilkan nilai 2,78 detik pada persentil ke-95 dan metris *http_req_duration* memiliki nilai 0%, artinya semua *request* berhasil terkirim.

4. Kesimpulan

Berdasarkan penelitian yang telah dilaksanakan dan juga hasil yang diperoleh dari penelitian, dapat ditarik beberapa kesimpulan sebagai berikut:

a. Dari hasil pengujian performa pengambilan data dalam skenario 5 *virtual users* dan 20 iterasi dengan rentang waktu 89 hari, terbukti bahwa *sharding database* meningkatkan kecepatan pemrosesan hingga 60,46%.
b. Dalam penelitian ini, ketidakaktifan *transform service* diuji dengan mengirimkan 10000 data melalui 10000 pengguna virtual. Hasilnya, sistem berhasil mengirimkan 100% data tersebut. Ini menunjukkan kemampuan arsitektur Kappa dalam menjaga kelangsungan proses dan data meskipun menghadapi gangguan pada salah satu *service*-nya.
c. Pada pengujian performa *kappa architecture* yang dikombinasikan dengan *sharding database*, dikirimkan 10000 data oleh 10000 *virtual users*. Sistem yang mengimplementasikan *kappa architecture* yang dikombinasikan dengan *sharding database* menunjukkan peningkatan kecepatan sebesar 70,26% dibandingkan dengan sistem yang hanya mengimplementasikan *sharding database*, tetapi tidak mengimplementasikan *kappa architecture*.

References

[1] N. Marz and J. (James O.) Warren, *Big data : principles and best practices of scalable real-time data systems*, vol. 1. 2015.

[2] F. N. Fote, S. Mahmoudi, A. Roukh, and S. A. Mahmoudi, “Big Data Storage and Analysis for Smart Farming,” in *Proceedings of 2020 5th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications, CloudTech 2020*, Institute of Electrical and Electronics Engineers Inc., Nov. 2020. doi: 10.1109/CloudTech49835.2020.9365869.

[3] M. Tahmassebpour, “A New Method for Time-Series Big Data Effective Storage,” *IEEE Access*, vol. 5, pp. 10694–10699, 2017, doi: 10.1109/ACCESS.2017.2708080.

[4] S. Samidi, R. Y. Suladi, and A. B. Lesmana, “Implementation of Database Distributed Sharding Horizontal Partition in MySQL. Case Study of Application of Food Serving On Kemkes,” *JURNAL SISFOTEK GLOBAL*, vol. 12, no. 1, p. 50, Mar. 2022, doi: 10.38101/sisfotek.v12i1.477.

[5] Y. Kumar, “Lambda Architecture-Realtime Data Processing,” 2020, doi: 10.13140/RG.2.2.19091.84004.

[6] J. B. Nkamla Penka, S. Mahmoudi, and O. Debauche, “A new Kappa Architecture for IoT Data Management in Smart Farming,” in *Procedia Computer Science*, Elsevier B.V., 2021, pp. 17–24. doi: 10.1016/j.procs.2021.07.006.

[7] O. Debauche *et al.*, “RAMi: A New Real-Time Internet of Medical Things Architecture for Elderly Patient Monitoring,” *Information (Switzerland)*, vol. 13, no. 9, Sep. 2022, doi: 10.3390/info13090423.

[8] N. A. N. Sobri *et al.*, “INTERNATIONAL JOURNAL ON INFORMATICS VISUALIZATION: Study of Database Connection Pool in Microservice Architecture,” pp. 566–571, 2022, [Online]. Available: www.joiv.org/index.php/joiv