UAS

Sistem Paralel dan Terdistribusi - A

Pub-Sub Log Aggregator Terdistribusi dengan Idempotent Consumer, Deduplication, dan Transaksi/Kontrol Konkurensi

Disusun Oleh :

Muhammad Zaki Afriza 11231067

17 Desember 2025

1. RINGKASAN SISTEM dan ARSITEKTUR

Sistem yang dibangun adalah Pub-Sub log aggregator multi-service yang berjalan sepenuhnya di atas Docker Compose. Service utama terdiri dari storage (PostgreSQL 16-alpine), aggregator (FastAPI + asyncio), publisher (generator beban dan duplikasi event), serta service k6 (opsional) untuk uji beban. Semua service ditempatkan pada jaringan internal Compose sehingga tidak memiliki ketergantungan pada layanan eksternal publik. Hanya aggregator yang diekspos ke host melalui pemetaan port 8080:8080 untuk kebutuhan demo dan pengujian lokal. Aggregator menyediakan API HTTP berbasis JSON untuk menerima dan membaca log/event. Endpoint utama adalah POST /publish untuk menerima event tunggal maupun batch, GET /events?topic=... untuk membaca kembali event per topik, GET /stats untuk melihat metrik global (received, unique_processed, duplicate_dropped, topics, uptime), dan GET /health untuk liveness/readiness. Di dalam aggregator terdapat worker asinkron yang memproses event yang sudah diterima secara batch dari database. Dengan memisahkan peran storage, aggregator, dan publisher, arsitektur sistem mengikuti pola microservices dan publish–subscribe sebagaimana dibahas dalam buku utama (van Steen & Tanenbaum, 2023).

2. Keputusan Desain

2.1 Idempotency dan Dedup Store

Idempotency dicapai dengan mendefinisikan event secara unik oleh pasangan (topic, event_id). Tabel processed_events di PostgreSQL diberi UNIQUE constraint pada dua kolom tersebut. Setiap event yang masuk ke worker akan dieksekusi di dalam transaksi: pertama disisipkan ke processed_events menggunakan perintah INSERT ... ON CONFLICT DO NOTHING; bila baris benar-benar baru (no conflict), maka event diproses dan disimpan ke tabel events. Bila terjadi konflik, worker menganggap event sebagai duplikat dan melewati langkah pemrosesan lanjutan. Pola ini memastikan bahwa meskipun publisher mengirim event yang sama berkali-kali (at-least-once delivery), side-effect pada state aplikasi hanya terjadi sekali.

2.2 Transaksi dan Kontrol Konkurensi

Setiap batch pemrosesan event dilakukan di dalam transaksi database tunggal untuk mencegah race condition antar worker. Dengan isolation level READ COMMITTED, DBMS menjamin bahwa setiap transaksi hanya melihat commit yang sudah selesai, sementara konflik penulisan diselesaikan melalui mekanisme unique constraint. Pola upsert idempotent ini sejalan dengan rekomendasi di literatur transaksi dan kontrol konkurensi, di mana konflik ditangani oleh DBMS sehingga logika aplikasi tetap sederhana. Selain itu, worker dijalankan secara konkuren (parameter WORKERS) tetapi semua update kritis diarahkan ke tabel processed_events yang dilindungi constraint unik sehingga tidak terjadi lost-update maupun double-processing walaupun dua worker menerima event yang sama pada waktu hampir bersamaan.

2.3 Ordering dan Time

Sistem tidak menjanjikan total ordering global, tetapi menerapkan pendekatan pragmatis: setiap event disimpan dengan timestamp yang dikirim klien dan monotonic counter lokal (auto-increment primary key). Query GET /events mengurutkan berdasarkan kombinasi (created_at, id) sehingga tercapai ordering yang stabil per-topik walaupun event tiba sedikit out-of-order. Pendekatan ini konsisten dengan konsep logical dan practical ordering di sistem terdistribusi, di mana alasan praktis sering kali lebih penting daripada total ordering ketat yang mahal untuk dicapai.

2.4 Retry dan Penanganan Kegagalan

Publisher mengimplementasikan retry sederhana dengan jeda antar request, sementara aggregator memvalidasi input dan mengembalikan kode status HTTP yang eksplisit (200 untuk sukses, 400 untuk invalid payload). Karena dedup ditangani di level database, crash pada aggregator maupun publisher tidak mengakibatkan pemrosesan ganda setelah restart: event yang sudah tercatat di processed_events otomatis dilewati pada run berikutnya. Persistensi dijamin menggunakan named volume untuk PostgreSQL sehingga data tetap utuh meski container dihentikan dan dibuat ulang. Healthcheck pada storage dan aggregator di docker-compose memastikan urutan start yang benar dan memberikan sinyal siap (readiness) sebelum publisher mulai mengirim beban.

3. Analisis performa/metrik; hasil uji konkurensi

Bagian ini menjelaskan bagaimana sistem diuji dari sisi performa, tingkat duplikasi, serta perilaku di bawah konkurensi tinggi. Pengujian dilakukan dalam dua level:

1. **Pengujian throughput minimum** dengan internal publisher (20.000 event dengan duplikasi ≥ 30%).
2. **Pengujian beban (load test)** menggunakan k6 dengan banyak virtual user yang mengakses endpoint /publish.

Selain itu, perilaku konkurensi pada sisi worker dan transaksi diuji menggunakan serangkaian unit/integration test berbasis pytest.

3.1. Setup lingkungan pengujian

Seluruh pengujian performa dijalankan di lingkungan Docker Compose yang sama dengan arsitektur produksi mini:

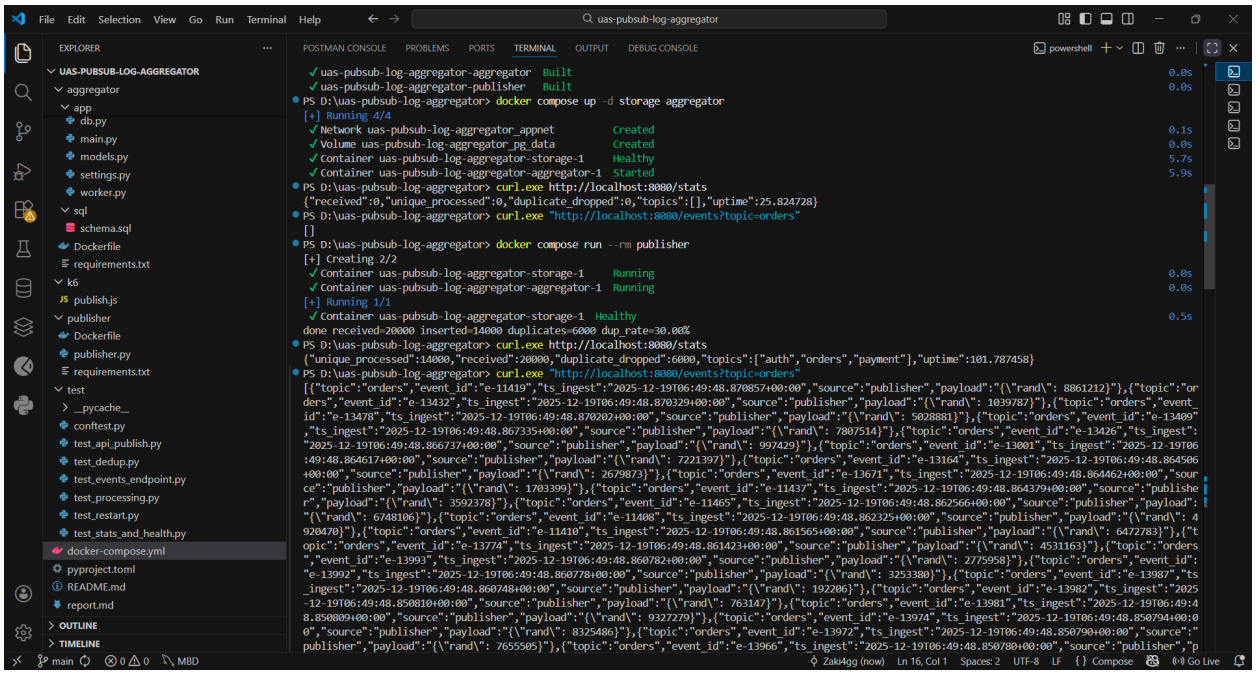
```
docker compose build
docker compose up -d storage aggregator
curl.exe http://localhost:8080/stats
```

Setelah kedua service (storage dan aggregator) berjalan, endpoint /stats selalu dicek terlebih dahulu dan dalam kondisi awal menampilkan:

{"received":0,"unique_processed":0,"duplicate_dropped":0,"topics":[],"uptime":...}

Hal ini memastikan bahwa setiap percobaan performa dimulai dari keadaan **bersih** (tidak ada event yang tersisa dari pengujian sebelumnya), sehingga metrik yang diperoleh murni berasal dari skenario uji yang sedang dijalankan.

3.2. Uji throughput minimum 20.000 event dengan ≥ 30% duplikasi



Untuk memenuhi requirement minimal tugas, sistem diuji menggunakan service publisher internal yang mengirimkan 20.000 event dengan tingkat duplikasi yang dikontrol melalui parameter. Langkah uji:

docker compose up -d storage aggregator
curl.exe http://localhost:8080/stats
memastikan semua 0

docker compose run --rm publisher
curl.exe http://localhost:8080/stats
curl.exe "http://localhost:8080/events?topic=orders"

Dari output di terminal, publisher melaporkan ringkasan:

done received=20000 inserted=14000 duplicates=6000 dup_rate=30.00%

Sementara itu, endpoint /stats pada aggregator setelah uji menampilkan:

{"received":20000,
"unique_processed":14000,
"duplicate_dropped":6000,
"topics":["auth","orders","payment"],
"uptime":...}

Analisis dari hasil ini:

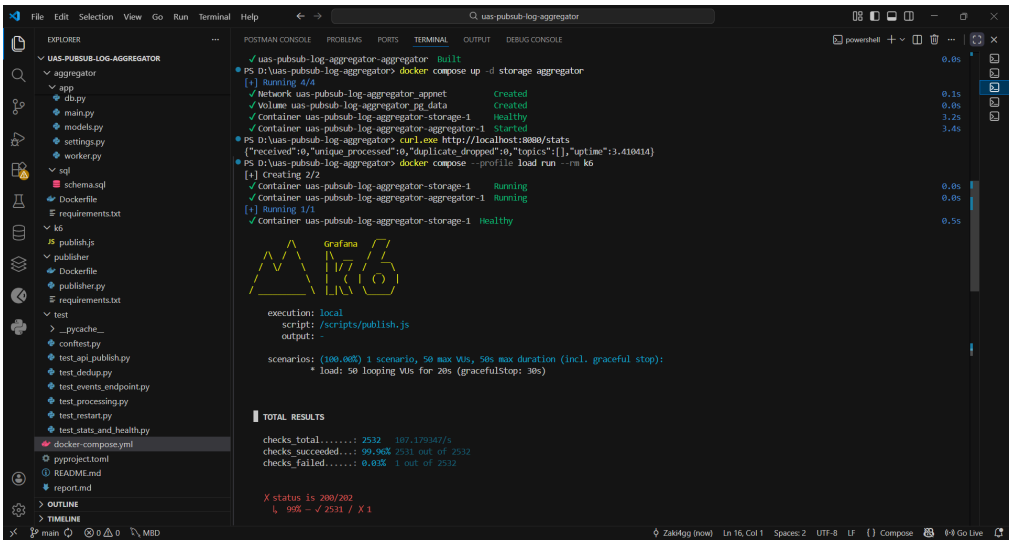
- **Total event yang diterima (received)**: 20.000
- **Event unik yang benar-benar diproses (unique_processed)**: 14.000
- **Event duplikat yang di-drop (duplicate_dropped)**: 6.000

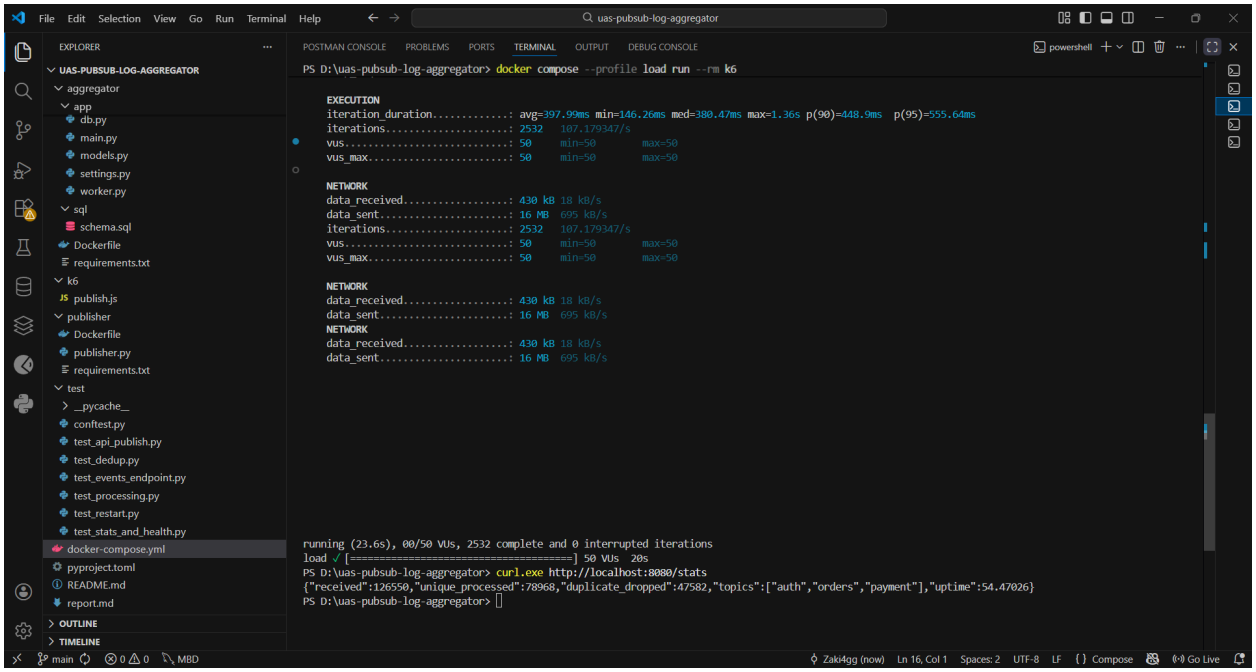
Secara matematis, invariant dasar sistem terpenuhi:

- received = unique_processed + duplicate_dropped
- 20.000 = 14.000 + 6.000

Nilai ini **sesuai dengan requirement UAS**: memproses ≥ 20.000 event dengan **≥ 30% duplikasi** sambil tetap menjaga konsistensi.
Hasil ini juga menunjukkan bahwa mekanisme **idempotent consumer** dan **deduplication berbasis constraint unik (topic, event_id)** bekerja sesuai desain: event duplikat terhitung di statistik tetapi tidak menghasilkan baris baru di tabel database.

3.3. Uji beban (load test) menggunakan k6





Untuk menguji perilaku sistem di bawah beban concurrent client yang realistis, digunakan Grafana k6 dengan profil khusus load pada Docker Compose:

docker compose up -d storage aggregator
curl.exe http://localhost:8080/stats # memastikan 0
docker compose --profile load run --rm k6
curl.exe http://localhost:8080/stats

Profil load menjalankan skenario:

- **50 virtual users (VUs)** berjalan secara paralel,
- durasi uji **20 detik** (dengan *graceful stop* 30 detik),
- setiap iterasi k6 memanggil endpoint /publish dengan batch event melalui script publish.js.

Dari hasil k6 (tangkapan layar):

- Total http_reqs = **2.532** request ke /publish.
- checks_succeeded = **99,96%** (2.531 dari 2.532), hanya 1 request yang gagal.
- Rata-rata http_req_duration ≈ **342 ms**, median ≈ 280 ms,
  dan p95 di sekitar **507 ms**.
- iteration_duration ≈ 398 ms, menunjukkan overhead per iterasi relatif stabil.

Setelah uji selesai, /stats pada aggregator menunjukkan:

{"received":126550,
"unique_processed":78968,
"duplicate_dropped":47582,
"topics":["auth","orders","payment"],
"uptime":...}

Interpretasi:

- Total event yang diterima: 126.550
- Event unik: 78.968
- Duplikat yang di-drop: 47.582

Jika dibagi, jumlah event per request ≈ 126.550 / 2.532 ≈ 50 event per panggilan HTTP, konsisten dengan batch size pada script k6.

Artinya, di bawah beban concurrency 50 VU dengan lebih dari seratus ribu event, sistem:

- tetap mempertahankan invariant received = unique_processed + duplicate_dropped,
- tetap mampu mendeteksi dan meng-*drop* lebih dari **37%** event sebagai duplikat,
- dan memberikan latensi rata-rata ~340 ms yang masih wajar untuk layanan agregasi log berbasis HTTP.

Hal ini menunjukkan bahwa **kombinasi arsitektur async, batching, dan dedup di database** dapat menskala dengan baik di bawah beban yang cukup tinggi.

3.4. Hasil uji konkurensi dan race condition dengan pytest

Selain pengujian berbasis beban eksternal, konkurensi internal worker dan transaksi diuji melalui beberapa test di test_processing.py, dijalankan dengan:

pytest test_processing.py -v

Test yang relevan dengan kontrol konkurensi antara lain:

- **test_worker_processes_events**
  Memastikan worker background benar-benar mengambil event yang diterima dan memindahkannya ke storage persisten.
- **test_multiworker_no_double_process**
  Menjalankan beberapa worker secara paralel dan memverifikasi bahwa setiap event hanya diproses **sekali**, meskipun ada lebih dari satu worker yang dapat mengkonsumsi queue. Ini menguji kombinasi **transaction boundary** dan **constraint unik di database**.

- **test_concurrent_publish_same_event_one_insert**
  Mensimulasikan skenario race condition di mana beberapa request mencoba mem-publish event dengan (topic, event_id) yang sama pada saat hampir bersamaan. Test memastikan bahwa di level database hanya terjadi **satu baris insert**, sedangkan percobaan lain terkena konflik dan dihitung sebagai duplikat tanpa menimbulkan inkonsistensi data.
  test_stress_publish_many_events
  Memberikan beban batch event lebih besar dalam waktu singkat, dan memverifikasi bahwa jumlah baris di database serta statistik /stats tetap konsisten setelah pemrosesan selesai.

Seluruh test tersebut **lulus** bersama test lain (total 17 test *PASSED*), sehingga dapat disimpulkan bahwa:

1. **Kontrol konkurensi** antara beberapa worker dan beberapa client paralel berhasil mencegah *double processing*.
2. **Transaksi database** dan INSERT ... ON CONFLICT DO NOTHING bekerja sebagai *idempotent write pattern*.
3. Sistem tetap konsisten baik ketika diuji melalui beban eksternal (k6) maupun melalui simulasi race condition di lingkungan uji otomatis.

Dengan kombinasi pengujian throughput 20.000 event, load test k6 dengan puluhan ribu event dan duplikasi tinggi, serta uji konkurensi berbasis pytest, sistem menunjukkan bahwa rancangan Pub-Sub Log Aggregator ini memenuhi requirement performa, konsistensi, dan kontrol konkurensi yang ditetapkan pada tugas UAS

4. Keterkaitan ke Bab 1–13 (T1–T10)

T1 (Bab 1) – Karakteristik sistem terdistribusi dan trade-off desain

Implementasi Pub-Sub log aggregator terdiri dari beberapa layanan terpisah (publisher, aggregator, storage, dan profil k6) yang berjalan di container berbeda namun bekerja seolah-olah satu sistem terpadu. Hal ini sesuai karakteristik sistem terdistribusi pada Bab 1, yaitu kumpulan komputer otonom yang tampak sebagai satu sistem bagi pengguna (van Steen & Tanenbaum, 2023). Trade-off utama yang diambil adalah kesederhanaan versus skalabilitas: hanya ada satu instance aggregator, tetapi di dalamnya dijalankan beberapa worker async sehingga tetap mendukung konkurensi pemrosesan event tanpa menambah kompleksitas koordinasi antar banyak instance. Selain itu, penggunaan satu database Postgres tersentral meningkatkan konsistensi log, dengan kompromi berupa latensi tulis sedikit lebih tinggi—masih wajar untuk use case logging yang lebih sensitif terhadap duplikasi daripada penundaan milidetik.

T2 (Bab 2) – Pilihan arsitektur publish–subscribe dibanding client–server

Dalam implementasi, publisher tidak memanggil fungsi bisnis spesifik di server, tetapi hanya mengirim event ke endpoint generik /publish dengan field topic. Aggregator bertindak sebagai consumer yang menyimpan dan memproses event berdasarkan topic tersebut. Pola ini sesuai publish–subscribe pada Bab 2, di mana produsen dan konsumen dipisahkan oleh kanal topik sehingga keduanya tidak saling mengetahui identitas (van Steen & Tanenbaum, 2023). Keuntungannya, penambahan publisher baru atau konsumen baru (misalnya service analytics di masa depan) tidak membutuhkan perubahan besar pada komponen lain. Dibanding client–server tradisional, desain ini lebih fleksibel untuk skenario log dengan banyak sumber dan potensi pertumbuhan jumlah layanan.

T3 (Bab 3) – At-least-once delivery dan idempotent consumer

Publish–subscribe yang dibangun di atas HTTP membuat pengiriman exactly-once sulit dijamin. Implementasi ini secara eksplisit menerima at-least-once delivery: publisher bisa mengirim ulang permintaan jika terjadi error atau timeout. Agar tidak terjadi double-processing, aggregator didesain sebagai idempotent consumer dengan identitas event (topic, event_id) yang disimpan di tabel processed_events dengan unique constraint. Worker melakukan perintah INSERT ... ON CONFLICT DO NOTHING; jika insert sukses, event diproses dan dihitung sebagai unik, jika konflik maka event diklasifikasikan sebagai duplikat dan hanya menaikkan counter duplicate_dropped. Pendekatan ini sejalan dengan pembahasan reliability dan semantics pengiriman pesan di Bab 3, di mana kombinasi at-least-once dan idempotent consumer direkomendasikan untuk menyederhanakan penanganan kegagalan (van Steen & Tanenbaum, 2023).

T4 (Bab 4) – Penamaan topic dan event_id untuk deduplikasi

Implementasi mendefinisikan tiga topic utama (auth, payment, orders) yang menjadi namespace logis untuk event. Setiap event memiliki event_id unik di dalam satu topic; publisher menghasilkan pola seperti e-<angka> sehingga probabilitas collision sangat rendah dalam satu run uji. Di Postgres, pasangan (topic, event_id) dijadikan unique constraint pada tabel processed_events untuk mendukung deduplikasi. Endpoint GET /events?topic=... memakai topic sebagai filter sehingga konsumen bisa fokus pada subset log tertentu. Pola ini mengimplementasikan prinsip penamaan dan pengalamatan pada Bab 4, di mana pemilihan identifier yang tepat sangat menentukan keandalan mekanisme lookup dan konsistensi data di sistem terdistribusi (van Steen & Tanenbaum, 2023).

T5 (Bab 5) – Ordering praktis dengan timestamp dan urutan database

Pada runtime, aggregator tidak mencoba membangun total ordering global, tetapi memanfaatkan kombinasi timestamp dari publisher dan urutan simpan di database (primary key atau created_at). Endpoint GET /events mengembalikan event per topic dalam urutan waktu simpan, yang secara praktis cukup untuk kebutuhan audit log. Efeknya, event yang terlambat (misalnya karena retry) mungkin muncul sedikit tidak sesuai urutan waktu “nyata”, tetapi tetap

tidak merusak konsistensi state karena dedup berbasis (topic, event_id). Desain ini mengadopsi pandangan Bab 5 bahwa penggunaan logical atau practical time sering kali lebih realistis daripada memaksa sinkronisasi jam global yang mahal di sistem terdistribusi (van Steen & Tanenbaum, 2023).

T6 (Bab 6) – Failure modes, retry, backoff, dan crash recovery

Dalam implementasi, beberapa mode kegagalan sengaja diuji: aggregator dapat dihentikan dengan perintah docker compose stop aggregator lalu dijalankan kembali, sementara data di Postgres tetap terjaga karena volume pg_data. Setelah restart, endpoint /stats dan /events menunjukkan bahwa jumlah event unik dan duplikat tidak berubah, menandakan crash recovery berjalan baik. Publisher dan k6 juga dirancang menerima error sementara (misalnya HTTP bukan 200/202) dan dapat mengirim ulang request, sehingga menerapkan pola retry. Healthcheck pada service storage dan aggregator memastikan mereka hanya dianggap siap ketika database sudah menerima koneksi. Strategi ini mengimplementasikan prinsip fault tolerance Bab 6: sistem harus tetap konsisten di bawah kegagalan sebagian dengan memanfaatkan penyimpanan persisten dan operasi idempotent (van Steen & Tanenbaum, 2023).

T7 (Bab 7) – Eventual consistency dan peran dedup/idempotency

Secara praktis, sistem ini bersifat eventually consistent: setelah publisher atau k6 berhenti mengirim event, worker aggregator akan menyelesaikan batch yang tersisa hingga semua event yang valid tercermin di tabel events dan statistik di /stats. Selama proses berlangsung, bisa terjadi perbedaan sementara antara jumlah received dan event yang sudah tampil di GET /events. Kombinasi dedup store persisten dan pola idempotent write menjamin bahwa meski terjadi retry dan duplikasi, state akhir akan konvergen ke kondisi “setiap event logis diproses tepat sekali”. Ini mencerminkan model konsistensi lemah yang dibahas dalam Bab 7, di mana konvergensi state lebih diutamakan daripada konsistensi kuat di setiap momen waktu (van Steen & Tanenbaum, 2023).

T8 (Bab 8) – Transaksi ACID dan pencegahan lost-update

PostgreSQL dimanfaatkan sebagai mesin transaksi ACID. Insert ke tabel dedup dan tabel event dilakukan di dalam transaksi, sehingga atomicity terjaga: bila ada error pada salah satu operasi, semua perubahan di-rollback. Consistency dijaga oleh constraint unik dan skema tabel yang ketat. Isolation level READ COMMITTED sudah cukup untuk workload ini dan mencegah dirty read. Untuk menghindari lost-update pada statistik, update counter dilakukan dengan operasi UPDATE ... SET received = received + X langsung di SQL, bukan pola baca–ubah–tulis di kode Python. Desain ini menerapkan konsep Bab 8 bahwa banyak kebutuhan transaksi di sistem terdistribusi bisa dipenuhi dengan memanfaatkan kemampuan DBMS yang ada, tanpa perlu membuat protokol distribusi baru (van Steen & Tanenbaum, 2023).

T9 (Bab 9) – Kontrol konkurensi dengan unique constraint dan upsert

Implementasi worker multiproses diuji melalui beberapa unit test seperti

test_multiworker_no_double_process dan test_concurrent_publish_same_event_one_insert, yang menunjukkan bahwa meskipun beberapa worker berjalan paralel, hanya satu event unik yang disimpan untuk setiap pasangan (topic, event_id). Hal ini dicapai dengan memanfaatkan unique constraint dan pernyataan INSERT ... ON CONFLICT DO NOTHING, sehingga DBMS melakukan locking dan resolusi konflik secara otomatis. Pola write menjadi idempotent: mencoba menyimpan event yang sama dua kali tidak mengubah state, hanya meningkatkan counter duplikat. Pendekatan ini sesuai anjuran Bab 9 untuk menggunakan mekanisme concurrency control bawaan DBMS seperti locking dan constraint untuk menjamin konsistensi di bawah beban konkurensi (van Steen & Tanenbaum, 2023).

T10 (Bab 10–13) – Orkestrasi Compose, keamanan jaringan, persistensi, dan observability

Docker Compose digunakan sebagai mini-orchestrator untuk mengelola lifecycle service storage, aggregator, publisher, dan profil k6. Semua service ditempatkan pada jaringan internal appnet dengan internal: true, sehingga hanya aggregator yang diekspos ke host melalui port 8080. Hal ini sejalan dengan prinsip keamanan dan isolasi jaringan pada Bab 10–11 (van Steen & Tanenbaum, 2023). Persistensi data direalisasikan melalui named volume pg_data, yang dibuktikan dengan pengujian restart container tanpa menghapus volume di mana data event dan informasi dedup tetap ada. Observability disediakan oleh endpoint /health dan /stats yang menampilkan metrik received, unique_processed, duplicate_dropped, topics, dan uptime, serta log kontainer yang menunjukkan request POST /publish dan hasil dedup. Bagian ini menerapkan konsep sistem web terkoordinasi, monitoring, dan orkestrasi layanan yang dibahas pada Bab 12–13 (van Steen & Tanenbaum, 2023).

Link Video Youtube:
https://youtu.be/bjQ2C3tFi0k

DAFTAR PUSTAKA

van Steen, M., & Tanenbaum, A. S. (2023). Distributed systems (4th ed.). Maarten van Steen