Nama : Tito Ariffianto Miftahul Huda
NIM : 11221035
Sistem Paralel dan Terdistribusi

T1 (Bab 1): Jelaskan karakteristik utama sistem terdistribusi dan trade-off yang umum pada desain Pub-Sub log aggregator.
Jawaban :
Sistem terdistribusi dicirikan oleh:

- heterogeneity (beragam hardware/OS),
- concurrency (banyak proses paralel),
- lack of a global clock,
- partial failures (node/network failures tanpa kegagalan total), dan
- scalability requirements (horizontal scaling).

Pub-Sub log aggregator, memiliki desain yang harus menyeimbangkan trade-off utama berikut:

- latency vs durability (sinkronous flush ke disk menambah latency tetapi menjamin durability),
- consistency vs availability (pengaturan replikasi/ack policy; lebih strict consistency mengurangi availability saat partition), serta
- throughput vs ordering (memecah topic ke banyak partition menaikkan throughput tapi melemahkan global ordering). (van Steen & Tanenbaum, 2023, Chap. 1).

T2 (Bab 2): Bandingkan arsitektur client-server vs publish-subscribe untuk aggregator. Kapan memilih Pub-Sub? Berikan alasan teknis.
Jawaban:
Model Client Server, memiliki mekanisme, producer (client) melakukan request secara langsung ke aggregator (server). Mekanisme ini memiliki kelebihan yaitu

- Desain sederhana
- Mudah untuk implementasi synchronous request/response.
- Lebih mudah untuk menegakkan konsistensi dan autentikasi (auth) per-request

Kekurangan dari mekanisme adalah sebagai berikut :

- Kurang scalable untuk skenario high-fan-out (satu pesan banyak penerima).
- Tidak cocok untuk offline consumers (penerima yang tidak selalu aktif.
- Menyebabkan tight coupling (keterikatan erat) antara produsen dan konsumen.

Model publish-subscribe (Pub-sub)
Memiliki mekanisme dimana producer menerbitkan events ke broker (atau topic). Broker kemudian bertindak sebagai perantara menangani

- Decoupling (loose losing), produsen dan konsumen tidak perlu tahu satu sama lain.
- Buffering, menampung pesan semnentara.
- Fan-out, Distribusi pesan ke banyak customers.
- Retention/Replay : Menyimpan data dan memungkinkan pemutaran ulang.
- Horizontal Scale, Skalabilitas melalui partitioning.

Model Pub-Sub paling cocok digunakan ketika:

- Tingkat ingestion data sangat tinggi.
- Terdapat banyak downstream consumers, misalnya untuk analitik, monitoring, alerting.

- Adanya kebutuhan data untuk replay data atau late-joining consumers(konsumer baru bergabung).

Pub-Sub dipilih saat sistem membutuhkan :

- Asynchronous ingestion
- Horizontal scalability via partitioning.
- Durable retention (penyimpanan data tahan lama) untuk pemrosesan ulang.
- Consumer elasticity (jumlah konsumen bisa bertambah atau berkurang secara fleksibel).

Meskipun scalable, model ini membawa kompleksitas dalam hal penanganan ordering, delivery semantics (jaminan pengiriman) dan koordinasi broker seperti partition leader dan consumer groups.
(van Steen & Tanenbaum, 2023, Chap. 2).

T3 (Bab 3): Uraikan at-least-once vs exactly-once delivery semantics. Mengapa idempotent consumer krusial di presence of retries?

Jawaban:

Perbandingan Semantika Pengiriman

- At-least-once: Memberikan jaminan bahwa sebuah pesan akan terkirim satu kali atau lebih. Ini adalah jaminan yang umum, namun memiliki konsekuensi: duplikasi pesan mungkin terjadi.
- Exactly-once: Bertujuan agar setiap event memengaruhi state sistem tepat satu kali secara end-to-end. Implementasi ini sangat mahal dan kompleks, seringkali memerlukan broker transactions, two-phase commit (2PC), atau mekanisme deduplication/offset atomicity di sisi broker.

Pendekatan Praktis: At-least-once + Idempotency

Dalam praktiknya, mencapai exactly-once murni, terutama lintas sistem yang heterogen (heterogeneous systems), sangatlah mahal dan rapuh (fragile).

Oleh karena itu, arsitektur yang umum digunakan adalah at-least-once (yang mengizinkan retries) dikombinasikan dengan idempotent consumer untuk menangani duplikasi.

Peran Krusial Idempotent Consumer

Ketika terjadi retries (misalnya karena network glitches atau consumer crashes), duplikasi pesan akan muncul. Di sinilah idempotent consumer menjadi krusial. Pola desain ini merupakan sebuah operasi consumer bersifat idempoten jika eksekusi berulang kali dengan input (pesan) yang sama akan menghasilkan state akhir yang sama seolah-olah operasi itu hanya dieksekusi satu kali. Jika consumer bersifat idempotent, pemrosesan ulang (reprocessing) pesan yang duplikat menjadi aman dan tidak akan menyebabkan korupsi state atau data.

Contoh Teknik Implementasi Idempotency:

- Menggunakan operasi UPSERT (UPDATE or INSERT) berdasarkan event_id yang unik.
- Menggunakan operasi INCREMENT yang dijaga oleh unique constraint (misalnya, memproses event_id hanya jika belum ada di tabel processed_events).
- Menggunakan compare-and-set (CAS) yang memanfaatkan sequence number atau version.

T4 (Bab 4): Rancang skema penamaan untuk topic dan event_id (unik, collision-resistant). Jelaskan dampaknya terhadap dedup.

Jawaban:
Praktik yang direkomendasikan untuk skema penamaan adalah menggunakan namespace topic yang hierarkis dan human-readable, serta *event_id* yang deterministik dan collision-resistant. Untuk topic, struktur hierarkis seperti *org-{orgId}/svc-{serviceName}/{env}/{resource}* (sering menggunakan kebab-case) sangat disarankan karena mempermudah pengaturan routing dan Access Control Lists (ACL). Sementara itu, untuk *event_id*, resistansi terhadap kolisi dapat dicapai dengan mengkombinasikan *producer_id* (misalnya UUIDv4), timestamp presisi tinggi (ISO8601 dengan nanoseconds), dan producer_sequence (angka monotonik per-produsen). Alternatif lain adalah menggunakan content-hash (misalnya SHA-256) dari payload yang telah dikanonikalisasi, atau menggunakan UUID berbasis waktu (UUIDv1/v6) maupun UUIDv4 dengan entropy yang kuat.

Dampak utama dari skema ini terlihat pada proses deduplikasi (dedup). Penggunaan *event_id* yang deterministik dan kanonikalis memungkinkan lookup $O(1)$ yang efisien pada dedup store (seperti hash table atau key-value store persistent). ID yang collision-resistant juga meminimalkan terjadinya false positive (kesalahan deteksi duplikat). Tentu ada trade-off yang harus dipertimbangkan, seperti panjang ID (yang mempengaruhi biaya penyimpanan/indeks), isu privasi (jika menggunakan payload hashing), dan kompleksitas pengelolaan sequence monotonik per-produsen. Penting juga untuk mempertimbangkan Time-To-Live (TTL) atau sliding window untuk entri di dedup store agar state tidak bertumbuh tanpa batas.(van Steen & Tanenbaum, 2023, Chap. 4)

T5 (Bab 5): Bahas ordering: kapan total ordering tidak diperlukan? Usulkan pendekatan praktis (mis. event timestamp + monotonic counter) dan batasannya.
Jawaban :
Total ordering, atau pengurutan sekuensial global untuk semua events, tidak selalu diperlukan dalam sistem terdistribusi (T5, Bab 5). Jika operasi bersifat commute (dapat dieksekusi dalam urutan apa pun dan menghasilkan hasil yang sama, misalnya appends ke key yang berbeda), atau jika sistem dapat beroperasi dengan eventual consistency yang hanya memerlukan per-key causality, maka per-partition atau per-key ordering (pengurutan per partisi atau per kunci) sudah memadai.

Pendekatan praktis untuk mencapai pengurutan adalah dengan menggunakan kombinasi event timestamp dan monotonic counter (penghitung yang selalu naik) per produsen/partisi. Contohnya adalah tuple (ts_iso8601, producer_seq) atau penggunaan Hybrid Logical Clocks (HLC) yang menggabungkan waktu fisik (physical time) dengan logical monotonicity. Metode ini memberikan pandangan global perkiraan (approximate global view) dengan petunjuk kausal (causal hints). Namun, pendekatan ini memiliki batasan: wall-clock skew (perbedaan jam fisik antar mesin) dapat membuat pengurutan berbasis timestamp menjadi menyesatkan. Selain itu, monotonic counters memerlukan koordinasi di sisi produsen (atau sequence harus ditetapkan oleh partition leader). Ketergantungan kausal lintas partisi (cross-partition causal dependencies) jauh lebih rumit dan memerlukan vector clocks (yang memiliki overhead besar) atau metadata kausal yang eksplisit.

Jika strict total order (urutan total yang ketat) mutlak diperlukan untuk kebenaran sistem (contohnya, memastikan urutan transfer perbankan antar rekening), solusi praktis memerlukan sequencer (pengurut terpusat) atau mekanisme konsensus (seperti Raft/Paxos). Trade-off dari solusi ini adalah biaya performa: sequencer tunggal dapat menjadi bottleneck, sementara konsensus akan menambah latensi. Oleh karena

itu, strategi yang disarankan adalah memilih per-partition sequencing ditambah timestamps untuk skalabilitas, dan hanya beralih ke global order (menggunakan konsensus) jika semantik yang kuat benar-benar dibutuhkan.
(van Steen & Tanenbaum, 2023, Chap. 5)

T6 (Bab 6): Identifikasi failure modes (duplikasi, out-of-order, crash). Jelaskan strategi mitigasi (retry, backoff, durable dedup store).
Jawaban:
Sistem terdistribusi dapat mengalami beberapa mode kegagalan utama:

- Duplicates (Duplikasi): Terjadi akibat retries (upaya pengiriman ulang) atau crash pada producer.
- Out-of-Order: Pesan diterima tidak sesuai urutan, sering disebabkan oleh partitioning atau replication lag.
- Crash (Consumer/Producer): Kegagalan pada consumer atau producer yang menyebabkan pemrosesan sebagian (partial processing).
- Network Partitions (Partisi Jaringan): Menyebabkan kondisi split-brain, di mana sistem terpecah menjadi sub-sistem yang terisolasi.
- Message Loss (Kehilangan Pesan): Terjadi jika tidak ada jaminan durability (penyimpanan yang andal).

Strategi Mitigasi Praktis
Untuk mengatasi mode kegagalan tersebut, beberapa strategi mitigasi dapat diterapkan:
Menangani Duplikasi:
Idempotent Processing: Mendesain consumer agar pemrosesan pesan yang sama berulang kali menghasilkan state akhir yang sama.
Durable Dedup Store: Menyimpan event_id yang telah diproses secara persisten (dengan TTL) untuk mencegah pemrosesan ulang.

Menangani Crash & Pemrosesan Sebagian:
Commit-After-Process Pattern: Menyimpan hasil pemrosesan dan offset secara atomik (bersamaan) dalam satu transaksi.
Checkpointing and Snapshots: Menyimpan state secara berkala untuk pemulihan pasca-crash.

Mengelola Jaringan & Retry:
Retries with Exponential Backoff + Jitter: Melakukan retry dengan jeda waktu yang meningkat secara eksponensial (ditambah jitter/waktu acak) untuk menghindari thundering herd (lonjakan permintaan serentak).
Circuit Breaker & Circuit-Aware Routing: Mekanisme untuk mendeteksi kegagalan jaringan dan mengalihkan traffic sementara.

Menjamin Durability & Ketersediaan:
Replication: Menyalin data ke beberapa node.
Configurable Acks/Replication Factor: Mengatur jumlah konfirmasi (acks) dan faktor replikasi untuk menyeimbangkan durability dengan latency.

Manajemen Aliran & Urutan:
Backpressure and Buffering: Broker memberikan sinyal backpressure (menahan laju) atau buffering untuk melindungi consumer yang lambat.
Leader per Partition: Menggunakan leader untuk setiap partisi guna menjamin urutan di dalam partisi tersebut (in-partition sequence).

Monitoring:
Secara aktif memonitor consumer lag (keterlambatan consumer) dan menerapkan kebijakan rebalancing jika diperlukan.

T7 (Bab 7): Definisikan eventual consistency pada aggregator; jelaskan bagaimana idempotency + dedup membantu mencapai konsistensi.
Eventual consistency (konsistensi pada akhirnya) adalah sebuah model jaminan di mana sistem menjamin bahwa jika tidak ada input (pembaruan) baru, semua replika atau consumer pada akhirnya akan konvergen ke state (keadaan) yang sama dalam waktu yang terbatas (bounded/finite time).

Untuk log aggregator, ini berarti: meskipun consumer mungkin menerima events secara out-of-order (tidak berurutan) atau duplikat, sistem akan memiliki mekanisme di latar belakang—seperti rekonsiliasi, anti-entropy (re8pair), atau replay—yang akan membawa semua views (tampilan data) ke konsensus akhir.

Idempotency (Idempotensi): Memungkinkan aplikasi ulang dari event yang sama tanpa mengubah final state secara salah. Ini adalah kunci utama saat menangani retries (pengiriman ulang) atau replay (pemutaran ulang) event yang diperlukan untuk mencapai konsistensi.
Deduplication (Deduplikasi): Mencegah over-counting (penghitungan berlebih) dan mempercepat konvergensi dengan cara mengabaikan pengiriman event yang duplikat.

Gabungan dari event_id yang deterministik,Penyimpanan processed-ids (ID event yang telah diproses) secara durable, Semantik pembaruan (update) yang idempotent (contoh: UPSERT, last-write-wins dengan metadata vector/time, atau operasi CRDT yang komutatif), membuat aggregator dapat mencapai eventual consistency lebih cepat dan lebih aman.

(van Steen & Tanenbaum, 2023, Chap. 7)

T8 (Bab 1–7): Rumuskan metrik evaluasi sistem (throughput, latency, duplicate rate) dan kaitkan ke keputusan desain.
Jawaban:

Metrik Evaluasi	Definisi	Kaitan dengan keputusan desain
Throughput	Jumlah *event* per detik yang berhasil di-*ingest* & diproses (events/sec)	**Dinaikkan** oleh: Penambahan jumlah *partition* (meningkatkan paralelisme). **Diturunkan** oleh: Penambahan

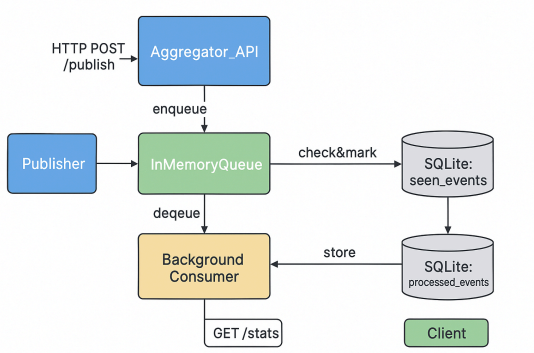
		replication factor atau pengaktifan semantik *exactly-once* (misalnya via transaksi). **SLA:** Sistem untuk analitik biasanya memprioritaskan *throughput* yang maksimal.
Latency	Waktu tunda *end-to-end*, diukur dari *publish* hingga *consumer acknowledgement* (ack). Dilaporkan dalam persentil (P50, P95, P99).	**Dinaikkan** (memburuk) oleh: Penambahan *replication factor* atau pengaktifan semantik *exactly-once*. **SLA:** Sistem *real-time alerting* harus meminimalkan P99 *latency*.
Duplicate Rate	Jumlah duplikat yang terdeteksi per Miliar (atau per Juta) *event*.	**Diturunkan** oleh: Pengaktifan semantik *exactly-once*. **Dikontrol** oleh: Kebijakan *dedup store* (ukuran, persistensi) yang menukar penggunaan memori/IO untuk *window* penekanan duplikat. **SLA:** Sistem *real-time alerting* harus meminimalkan *duplicate rate*.

Laporan: Event Aggregator Service (UTS)

Ringkasan Sistem dan Arsitektur

Deskripsi singkat
Layanan ini adalah *event aggregator* berbasis Python (FastAPI) yang menerima event dari publisher, memasukkannya ke antrian in-memory (*asyncio.Queue*), dan memprosesnya secara asinkron oleh sebuah background consumer. Event yang unik disimpan ke SQLite (persistensi untuk deduplikasi) dan event duplikat dibuang dengan logging yang jelas.

Diagram Arsitektur



Keputusan Desain

Idempotency & Dedup Store

- **Definisi idempotency**: event yang memiliki pasangan *(topic, event_id)* sama diproses maksimal sekali walau diterima berkali-kali.
- **Implementasi saat ini**: deduplikasi berbasis primary key komposit *(topic, event_id)* pada tabel *seen_events*.
- **Pilihan storage**: SQLite (local file) sebagai dedup store untuk memastikan keberlanjutan terhadap restart container. Pilihan ini sederhana, local-only, dan cukup untuk skala tugas.
- **Logging**: Setiap kali deteksi duplikat terjadi, sistem menuliskan log *Duplicate event dropped: <topic>/<event_id>*.

Ordering

- Sistem menjamin *FIFO ordering* untuk event yang masuk lewat satu request batch, karena *asyncio.Queue* adalah FIFO.

- **Total ordering antar pengirim berbeda** tidak dijamin dan dalam konteks aggregator (pemungutan log) tidak diwajibkan.
- Jika sistem butuh replay/strong-consistency ordering di masa depan, arsitektur perlu diganti ke sistem persisten queue (Kafka/RabbitMQ) dengan penandaan offset.

Retry & Reliability

- Publisher mensimulasikan *at-least-once delivery* (20% duplikat pada uji). Consumer mendeteksi duplikat menggunakan operasi atomik DB (INSERT dengan primary key) sehingga race condition teratasi.
- Untuk toleransi crash: SQLite yang disimpan di Docker Volume memastikan state deduplikasi bertahan setelah restart.
- Catatan: *in-memory queue* berarti ada kemungkinan message sedang diproses saat crash hilang dari queue—untuk hard reliability perlu persisted queue.

Analisis Performa & Metrik

Skenario uji

- Total event dikirim oleh publisher: **5,000**
- Persentase duplikat: **20% → 4,000 unique**, **1,000 duplicates**

Hasil observasi (dari run demo)

- received_total: 5,000
- unique_processed: 4,000
- duplicate_dropped: 1,000
- queue_size_current: 0 (setelah run selesai)

Analisis

- Endpoint *POST /publish* tetap sangat responsif karena hanya melakukan *queue.put()* (I/O minimal).
- Bottleneck utama adalah I/O write ke SQLite saat menyimpan event unik.
- *aiosqlite* membantu agar operasi DB tidak memblok event loop.
- Throughput yang diamati cukup tinggi (5000 events dikirim dan diproses dalam hitungan detik pada mesin pengujian lokal).

Tes & Validasi

Unit tests (pytest)

- Cakupan minimum: 5 test yang mencakup root health, publish single, publish batch, deduplication, dan GET /events.

- Semua test lulus pada lingkungan pengembang (`5 passed`).

Stress test

- Publisher mengirim batch 100 event per request sampai 5k total; konsistensi statistik dicek oleh skrip.

Observability & Logging

- Endpoint *GET /stats* menyajikan metrik operasional:” received_total, unique_processed, duplicate_dropped, queue_size_current, processed_topics, uptime_seconds”
- Logging mencakup: startup/shutdown, consumer processing, duplicate detection warnings, dan error stacktrace.

Kesimpulan & Rekomendasi

- Implementasi saat ini sesuai spesifikasi UTS: API, dedup, persistence lokal, Docker, dan tests.

Referensi

Van Steen, M., & Tanenbaum, A. S. (2023). *Distributed Systems* (4th ed.). Maarten Van Steen. https://distributed-systems.net/index.php/books/ds4/