Pertemuan 11 — Kafka, NATS, CDC, dan *Stream Processing*

Membangun *hot path*: dari setiap transaksi di *database* ke keputusan dalam hitungan detik

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Dari *batch* ke *event*; log sebagai abstraksi
2. Kafka dari dekat: *partition*, *acks*, *consumer group*, ISR, KRaft
3. *Retention*, *compaction*, *tiered storage*, UU PDP
4. *Schema registry*, CDC, *outbox*, *delivery guarantee*
5. Kafka, Redpanda, NATS, atau JetStream?
6. *Event time*, *window*, *watermark*, *late data*
7. *State*, *checkpoint*, *exactly-once*
8. *When to use*: Spark, Flink, Kafka Streams, Beam
9. Praktikum: CDC → Redpanda → Spark *Structured Streaming* → Iceberg dan ClickHouse

Sasaran pertemuan ini

Anda dapat mengalirkan setiap transaksi dari *database* inti bank ke Kafka lewat CDC, lalu mengolahnya dengan Spark *Structured Streaming* (*window*, *watermark*, *checkpoint*) menjadi *fraud alert* di *hot store* sekaligus tabel *bronze*, dan menjelaskan jaminan apa yang benar-benar Anda dapat.

Dari *batch* ke *event*

Keputusan yang tidak bisa menunggu esok hari

Sampai minggu lalu, data kita mengalir **berkala**: *batch* Spark dari *bronze* ke *gold*, *dashboard* pagi. Itu cukup untuk laporan, tetapi tidak untuk:

- memblokir transfer mencurigakan **sebelum** dana berpindah,
- notifikasi “transaksi Rp 5 juta dari perangkat baru” ke nasabah,
- saldo dan limit harian yang harus konsisten di semua kanal,
- *monitoring queue* QRIS saat jam makan siang.

Dua cara mengambil data dari *database*

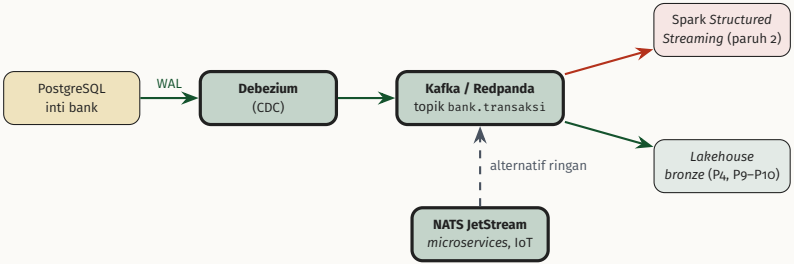
***Batch* (*pull* berkala)**: `SELECT * WHERE updated_at > :kemarin`. Sederhana, tetapi terlambat, membebani *database*, dan **kehilangan penghapusan**.

***Event* (*push* tiap perubahan)**: setiap `INSERT`, `UPDATE`, `DELETE` menjadi satu *event* dalam hitungan detik.

Ingat Sister P5

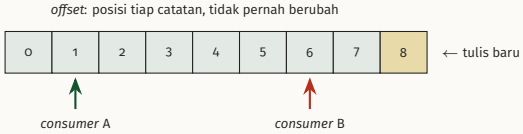
Model *pub-sub*, *message queue* vs log, dan *event-driven architecture* sudah Anda pelajari Sabtu, 31 Oktober. Hari ini kita **memakainya** untuk data.

Posisi pertemuan ini dalam arsitektur semester



Paruh pertama hari ini membangun *stream* (bergaris tebal); paruh kedua mengolahnya dengan Spark *Structured Streaming*, mesin yang sudah Anda kenal dari P9–P10.

Log: urutan catatan yang hanya ditambah di ujung



- ***Append-only***: catatan tidak diubah, hanya ditambah.
- **Terurut**: setiap catatan punya *offset* yang naik.
- **Dibaca berkali-kali**: tiap pembaca menyimpan posisinya sendiri; membaca tidak menghapus.
- **Dapat diputar ulang**: pembaca baru bisa mulai dari awal.

Mengapa ini penting untuk data

Database sendiri dibangun di atas log (*write-ahead log*). Jay Kreps menyebut log sebagai abstraksi pemersatu data *real-time*: bila semua perubahan tercatat terurut di satu log, sistem mana pun dapat membangun ulang keadaannya dengan membaca log itu.

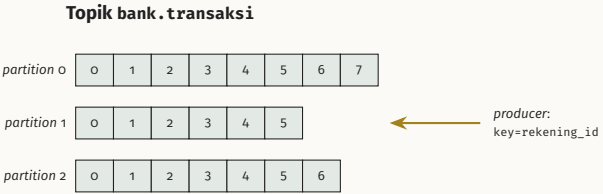
Ringkasan

Queue menghapus *message* setelah diambil; **log menyimpannya**. Itulah perbedaan Kafka dengan RabbitMQ yang Anda lihat di Sister P5.

Sumber: Kreps, *The Log*, 2013; Kleppmann, DDIA Bab 11.

Kafka dari dekat

Topik dipecah menjadi *partition*



- ***Partition*** adalah satuan paralelisme dan satuan urutan.
- Urutan dijamin **hanya di dalam satu *partition***, tidak antar *partition*.
- *Message* dengan ***key*** yang sama masuk ke *partition* yang sama (hash(*key*) mod *N* secara konsep, *N*: jumlah *partition*, mod: sisa bagi).
- *Partition* tersebar ke beberapa ***broker***.

Memilih *key* pada kasus bank

`key = rekening_id`: semua transaksi satu rekening terurut, cocok untuk menghitung saldo berjalan dan pola *fraud* per rekening.

`key = null`: tersebar rata, tetapi urutan per rekening hilang.

Jebakan *hot partition*

Satu rekening korporat dengan jutaan transaksi per hari membuat satu *partition* jauh lebih sibuk daripada yang lain, masalah *skew* yang sama seperti di Spark (P10).

acks: kapan *producer* menganggap *message* terkirim

Nilai	Arti	*Latency*	Risiko kehilangan
`acks=0`	Tidak menunggu jawaban *broker* sama sekali	paling rendah	tinggi: *message* bisa hilang tanpa diketahui
`acks=1`	Menunggu *leader* menulis ke lognya	rendah	ada: bila *leader* mati sebelum disalin
`acks=all`	Menunggu semua *replica* dalam ISR	lebih tinggi	rendah, bila `min.insync.replicas` ≥ 2

Untuk transaksi bank

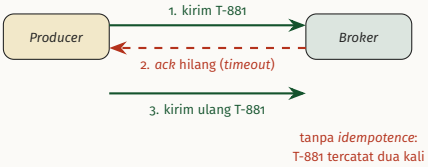
`acks=all`, *replication factor* 3, `min.insync.replicas=2`. Artinya satu *broker* boleh mati tanpa *data loss* dan tanpa menghentikan penulisan.

Untuk log klik aplikasi

`acks=1` sering cukup: kehilangan segelintir klik tidak mengubah keputusan, dan *latency*-nya lebih rendah.

Pilihan *acks* adalah keputusan bisnis: **berapa harga satu *message* yang hilang?**

Retry menimbulkan duplikat; *idempotent producer* mencegahnya

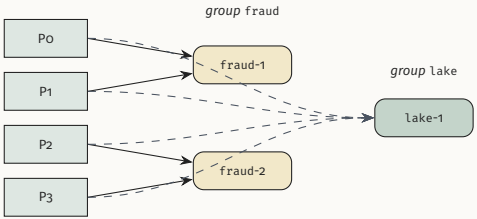


- ***Idempotent producer***: tiap *producer* punya ID dan tiap *message* per *partition* punya nomor urut. *Broker* menolak nomor urut yang sudah pernah diterima.
- Aktif secara *default* pada Kafka versi modern (`enable.idempotence=true`); tetap tuliskan eksplisit di *configuration*.
- Hanya mencegah duplikat akibat *retry* di **satu sesi** *producer*, per *partition*.
- **Transaksi Kafka** memperluasnya: tulis ke beberapa *partition* dan simpan *offset consumer* secara atomik (*read-process-write*).

Ringkasan

Duplikat di jaringan pasti terjadi. Rancang agar duplikat **tidak berbahaya**.

Satu *partition*, satu *consumer* per *group*



- Di dalam satu *group*, tiap *partition* dibaca **tepat satu** *consumer*: kerja terbagi.
- Antar *group*, tiap *group* membaca **semua** *message*: data yang sama dipakai banyak sistem.

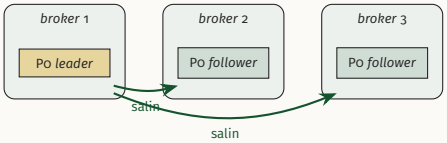
Offset commit

Consumer menyimpan posisi terakhir yang **sudah selesai diproses** ke topik internal `__consumer_offsets`. Bila mati, penggantinya melanjutkan dari situ.

Kapan *commit*?

Commit sebelum memproses → *message* bisa hilang. *Commit* sesudah memproses → *message* bisa diproses dua kali. Pilihan *default* yang aman: sesudah, ditambah pemrosesan yang idempoten.

Partition replication: *leader*, *follower*, ISR



- Tiap *partition* punya satu ***leader*** yang melayani tulis (dan, secara *default*, baca) serta beberapa ***follower***.
- **ISR** (*in-sync replicas*): *replica* yang tertinggal tidak terlalu jauh dari *leader*.
- Bila *leader* mati, salah satu anggota ISR dipilih menjadi *leader* baru.

Unclean leader election

Bila semua anggota ISR mati, Kafka bisa diizinkan memilih *replica* yang tertinggal (data terakhir hilang) atau menunggu (*partition* tidak tersedia). Untuk data keuangan, `unclean.leader.election.enable=false`: lebih baik berhenti sebentar daripada kehilangan transaksi.

Ini pilihan **C vs A** pada CAP yang dibedah di Sister P7, di sini muncul sebagai satu baris *configuration*.

KRaft: Kafka tanpa ZooKeeper

Dulu

Metadata *cluster* (daftar topik, *partition*, siapa *leader*, *configuration*) disimpan di **ZooKeeper**, sistem terpisah dengan operasinya sendiri.

Sekarang

Metadata disimpan di **log metadata** di dalam Kafka sendiri, direplikasi oleh sekelompok *controller* dengan protokol **KRaft**, turunan *consensus* Raft. Seri Kafka terbaru hanya mendukung mode KRaft.

Kaitannya dengan Sister P7

- *Controller quorum* memilih *leader* dengan *term* dan suara mayoritas, persis pemilihan *leader* Raft.
- Dengan 3 *controller*, *cluster* tahan 1 kegagalan; dengan 5, tahan 2 ($\lfloor (n-1)/2 \rfloor$, n: jumlah *controller*, $\lfloor \cdot \rfloor$: pembulatan ke bawah).
- Perhatikan: *replication* **data *partition*** memakai ISR, bukan Raft; Raft hanya untuk metadata. Redpanda memakai Raft untuk keduanya.

Ringkasan

Satu sistem lebih sedikit untuk dioperasikan, dan Anda sudah tahu cara kerjanya.

Berapa lama *message* disimpan?

Retensi waktu/ukuran

`retention.ms` atau `retention.bytes`. *Segment* log yang lebih tua dihapus utuh.

Contoh: topik transaksi 7 hari, cukup untuk memutar ulang *pipeline* yang rusak sepekan.

Log compaction

`cleanup.policy=compact`: untuk tiap *key*, hanya nilai **terakhir** yang dipertahankan. *Message* dengan nilai `null` (*tombstone*) menghapus *key*.

Contoh: topik `saldo` per rekening, profil nasabah.

Tiered storage

Segment lama dipindah ke *object storage*; hanya bagian terbaru di disk lokal *broker*. Retensi panjang jadi murah, konsep *hot/cold* dari P3, di dalam Kafka.



Retensi dan UU PDP: jangan lupa log

Topik Kafka adalah **salinan data** juga. Bila berisi data pribadi nasabah:

- Retensi panjang berarti data pribadi tersimpan lebih lama, harus ada dasarnya.
- Menghapus satu nasabah dari log *append-only* tidak mudah; *tombstone* hanya bekerja pada topik *compacted*.
- *Tiered storage* memindahkan data ke *bucket*: ikut kebijakan akses dan *bucket encryption* itu.

Pola yang dipakai industri

- Kirim **ID atau *token***, bukan NIK dan nama lengkap, di topik umum.
- Pisahkan topik berisi PII dengan ACL ketat.
- *Crypto-shredding*: *encryption* per nasabah, hapus kuncinya untuk "menghapus" datanya (dibahas di P14).

Sumber: Dibahas lengkap pada P14 (*security*) dan P15 (GRC).

Schema, CDC, dan *delivery guarantee*

Message tanpa *schema* adalah utang

Tim A mengganti `nominal` dari angka menjadi teks "`2.500.000`". Malam itu semua *consumer* gagal. Siapa yang salah?

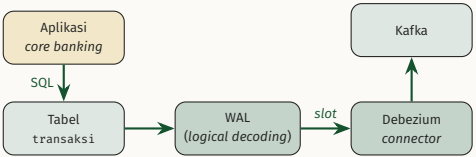
Schema registry

Layanan yang menyimpan versi *schema* tiap topik (Avro, Protobuf, atau JSON *Schema*). *Producer* mendaftarkan *schema*; *message* hanya membawa **ID *schema***; *consumer* mengambil *schema* dari *registry*. *Schema* baru ditolak bila melanggar aturan kompatibilitas.

```
# Skema Avro event transaksi (versi 2)
{
  "type": "record", "name": "Transaksi",
  "fields": [
    {"name": "trx_id",       "type": "string"},
    {"name": "rekening_id", "type": "string"},
    {"name": "nominal",      "type": "long"},
    {"name": "kanal",        "type": "string"},
    {"name": "akad",  "type": ["null", "string"],
     "default": null}      # kolom baru: opsional
  ]
}
```

Protobuf dan Avro sudah Anda temui di Sister P4 (gRPC); di sini keduanya menjadi *data contract* antar-tim.

CDC berbasis log: membaca WAL, bukan tabel



- PostgreSQL mencatat setiap perubahan di **WAL** sebelum menulis ke tabel.
- Dengan `wal_level=logical`, perubahan itu dapat dibaca sebagai baris (*logical decoding*, *plugin* `pgoutput`).
- Debezium membaca lewat ***replication slot*** dan menulis satu *event* per perubahan baris ke Kafka.

Dibanding *query* berkala

- Menangkap `DELETE` dan semua perubahan antara, bukan hanya keadaan akhir.
- Hampir tidak membebani tabel: tidak ada `SELECT` besar.
- Urutan perubahan per baris terjaga.
- Tidak butuh kolom `updated_at` yang disiplin.

Hati-hati

Replication slot menahan WAL sampai dibaca. Bila Debezium mati lama, disk *database* produksi bisa penuh. Pantau ukuran *slot*.

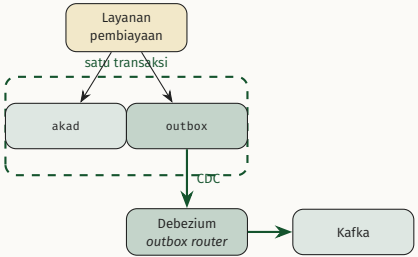
Outbox pattern: mengirim *event* tanpa *dual write*

Masalah *dual write*

Layanan pembiayaan menyimpan akad murabahah ke *database*, lalu mengirim *event* ke Kafka. Bila program mati di antara keduanya, *database* dan Kafka tidak sepakat, dan tidak ada yang tahu.

Solusi *outbox*

Dalam **satu transaksi *database***: simpan akad **dan** satu baris di tabel `outbox`. Debezium membaca `outbox` lewat CDC dan menerbitkannya ke Kafka. Atomisitas dijamin *database*.



Debezium menyediakan *transformation* khusus (*outbox event router*) untuk memetakan baris `outbox` ke topik tujuan.

At-most-once, *at-least-once*, *exactly-once*

Jaminan	Cara mendapatkannya	Akibat gagal	Cocok untuk
At-most-once	*Commit offset* sebelum memproses; `acks=0`	*message* hilang	*metric* kasar, telemetri yang boleh bolong
At-least-once	*Commit* sesudah memproses; *retry*	*message* diproses ulang (duplikat)	*default* yang wajar; aman bila pemrosesan idempoten
Exactly-once	*Idempotent producer* + transaksi Kafka, atau *sink* idempoten	—	saldo, *ledger*, agregat keuangan

Exactly-once punya batas

Transaksi Kafka menjamin *exactly-once* **di dalam** Kafka (baca-proses-tulis antar topik). Begitu efeknya keluar, mengirim SMS, memanggil API bank lain, jaminannya kembali ke *at-least-once*.

Praktik industri

Jalankan *at-least-once*, lalu buat penerima **idempoten**: simpan `trx_id` yang sudah diproses, atau tulis dengan `UPSERT` berdasarkan kunci unik. Hasil akhirnya setara *exactly-once*.

Kafka, Redpanda, NATS, atau JetStream?

Empat pilihan, empat karakter

	Kafka	**Redpanda**	**NATS *core***	**NATS JetStream**
Model	Log terdistribusi	Log, API Kafka	*Pub-sub*, *request-reply*	*Stream* persisten di atas NATS
Persistensi	Ya, ke disk	Ya, ke disk	**Tidak**	Ya (*file* atau memori)
Replay	Ya, dari *offset*	Ya	Tidak	Ya, per *consumer*
Jaminan	s.d. *exactly-once* (transaksi)	seperti Kafka	*at-most-once*	*at-least-once*; de-duplikasi via `Nats-Msg-Id`
Implementasi	JVM (Java, Scala)	C++, *thread-per-core*	Go, satu biner kecil	Go, biner yang sama
Replication	ISR (data), KRaft (metadata)	Raft per *partition*	—	Raft per *stream*
Ekosistem	Terluas: Connect, Debezium, Flink, Spark	Kompatibel klien Kafka	Klien banyak bahasa	Konektor data lebih sedikit

Sumber: Dokumentasi resmi Apache Kafka, Redpanda, NATS/JetStream. Karakter umum; ukur di beban Anda sendiri.

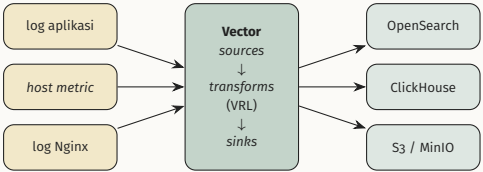
Kriteria memilih

1. **Perlu diputar ulang?** Bila *consumer* baru harus membaca sejarah (membangun ulang *lakehouse*, melatih ulang model), butuh log: Kafka, Redpanda, atau JetStream. NATS *core* tidak.
2. **Ekosistem konektor.** CDC dengan Debezium, *sink* ke Iceberg, integrasi Flink dan Spark: Kafka (dan Redpanda lewat API Kafka) paling lengkap.
3. ***Throughput* dan retensi.** Ratusan ribu *event*/detik dengan retensi berhari-hari: Kafka atau Redpanda.
4. **Jejak *resource*.** *Edge*, *gateway* IoT, cabang dengan server kecil: NATS sangat ringan.
5. **Pola komunikasi.** *Request-reply* antar *microservice* dan perintah: NATS alami. *Data stream* analitik: log.
6. **Kemampuan tim dan operasi.** Kafka butuh pengalaman operasi JVM dan *tuning*; layanan terkelola mengurangi beban itu.
7. **Lisensi dan dukungan.** Periksa lisensi edisi yang dipakai (Kafka: Apache 2.0; Redpanda: cek lisensi edisi *community*; NATS: Apache 2.0).

Ringkasan

Banyak organisasi memakai **keduanya**: NATS untuk *messaging* antar layanan dan di *edge*, Kafka sebagai jalur utama data yang dijembatani dari NATS.

Vector: memindahkan log dan *metric*, bukan transaksi



- Ditulis dengan Rust: jejak memori kecil, cocok sebagai agen di setiap server.
- Mengambil, mengubah (*parse*, menyaring, menyamarkan), dan merutekan log, *metric*, dan *trace*.
- Bisa menulis ke Kafka bila log perlu masuk *stream* yang sama.

Bukan pengganti CDC

Log aplikasi boleh hilang sebagian; transaksi tidak. Untuk data bisnis, pakai CDC dan Kafka dengan jaminan yang dibahas tadi.

Stream processing: waktu dan *window*

Bounded dan *unbounded*

Batch (P5, P6, P9–P10)

Data ***bounded***: *file* transaksi tanggal 30 November punya awal dan akhir. *Job* membaca semuanya, menghitung, selesai. Pertanyaan "total hari ini" punya satu jawaban final.

Streaming (hari ini)

Data ***unbounded***: topik `bank.transaksi` terus bertambah. *Job* berjalan berbulan-bulan. "Total hari ini" adalah jawaban yang **terus diperbarui**, dan bisa berubah ketika data terlambat datang.

Tiga hal yang membuat *streaming* sulit

1. **Waktu**: kapan sebuah *event* "terjadi" berbeda dari kapan ia "sampai".
2. **Keadaan** (*state*): menghitung jumlah per rekening berarti mengingat sesuatu di antara *event*.
3. **Kegagalan**: *job* yang mati di tengah jalan tidak boleh menghitung dua kali atau melewatkan *event*.

Ringkasan

Batch adalah kasus khusus *streaming*: *stream* yang kebetulan berakhir.

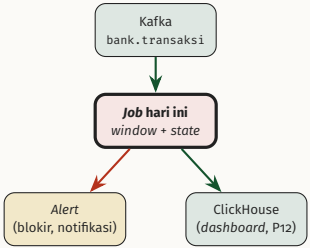
Kasus hari ini: aturan *fraud* sederhana

Aturan (ilustrasi)

Tandai rekening bila dalam **satu menit** terdapat **lebih dari lima** transaksi **dari lebih dari satu perangkat**.

Kedengarannya mudah: `GROUP BY rekening, menit`. Tetapi:

- "Satu menit" menurut jam siapa? Jam ponsel nasabah, jam server, atau jam *job*?
- Transaksi dari ATM yang sempat *offline* 3 menit masuk ke menit yang mana?
- Kapan kita boleh menyatakan "menit 10:01 sudah lengkap" dan mengirim *alert*?
- Bila *job* mati pukul 10:01:30, apakah hitungan menit itu hilang?



Sumber *stream*: topik CDC yang kita bangun di paruh pertama.

Empat pertanyaan Akidau

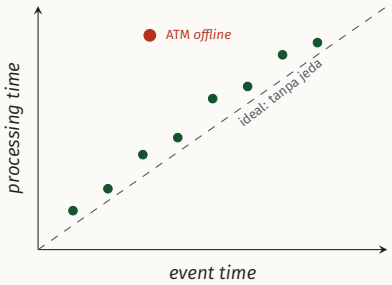
Pertanyaan	Dijawab oleh	Pada aturan *fraud* kita
What: apa yang dihitung?	*Transformation* dan *aggregation*	Jumlah transaksi dan jumlah perangkat berbeda per rekening
Where: di rentang *event time* mana?	*Window*	*Tumbling window* 1 menit menurut waktu transaksi
When: kapan hasil dikeluarkan?	*Watermark* dan *trigger*	Saat *watermark* melewati akhir menit; perbarui bila ada data terlambat yang masih diterima
How: bagaimana hasil yang diperbarui saling berhubungan?	Mode akumulasi	Hasil baru menggantikan hasil lama (*update*), bukan ditambahkan

Ringkasan

Setiap *job streaming* harus bisa menjawab keempat pertanyaan ini; bila tidak, hasilnya hampir pasti keliru saat data terlambat atau *job* dimulai ulang.

Sumber: Akidau dkk., *The Dataflow Model*, VLDB 2015; *Streaming Systems*, Bab 1–2.

Dua jam yang berbeda

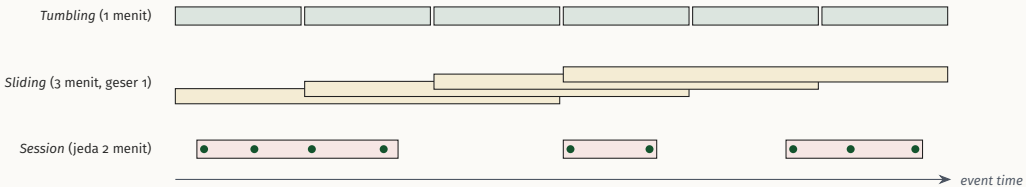


- ***Event time***: kapan transaksi benar-benar terjadi (stempel waktu di *event*).
- ***Processing time***: kapan *event* itu diproses *job*.
- Jaraknya (*skew*) tidak tetap: jaringan, *queue*, *retry*, perangkat *offline*.
- *Event* bisa datang **tidak berurutan** (*out-of-order*).

Mengapa penting

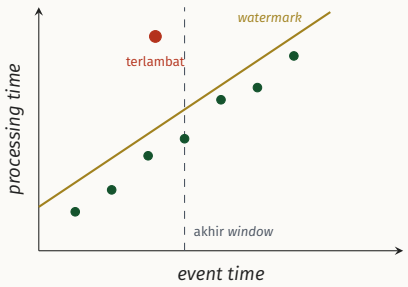
Menghitung per menit dengan *processing time* membuat hasil bergantung pada kecepatan jaringan hari itu. Menjalankan ulang *job* atas data yang sama akan memberi hasil berbeda. Untuk audit, itu tidak dapat diterima.

Tiga jenis *window*



Window	Sifat	Contoh di bank
Tumbling	Panjang tetap, tidak tumpang tindih; tiap *event* masuk tepat satu *window*	Jumlah transaksi QRIS per menit per *merchant*
Sliding	Panjang tetap, bergeser lebih pendek dari panjangnya; *event* masuk beberapa *window*	Nilai transfer 1 jam terakhir, diperbarui tiap 5 menit
Session	Dibatasi oleh jeda tanpa aktivitas; panjang berbeda per kunci	Satu sesi penggunaan aplikasi nasabah

Watermark: “sampai sini, saya kira sudah lengkap”



- **Watermark** W adalah pernyataan: “saya tidak berharap lagi menerima *event* dengan *event time* $< W$”.
- Heuristik umum: $W =$ *event time* terbesar yang sudah terlihat dikurangi batas jeda (misalnya 2 menit).
- Saat W melewati akhir sebuah *window*, *window* itu dianggap lengkap dan hasilnya boleh dikeluarkan.
- *Event* di bawah W disebut ***late* data**.

Trade-off

Batas jeda kecil: hasil cepat, lebih banyak data terlambat terbuang. Batas besar: hasil lengkap, tetapi *fraud alert* lebih lambat dan *state* lebih besar.

Apa yang dilakukan pada data terlambat?

Pilihan	Cara	Cocok bila
Buang	Data di bawah *watermark* diabaikan	Hasil cepat lebih penting daripada lengkap; dicatat sebagai *metric*
Allowed lateness	*Window* tetap dibuka beberapa saat setelah *watermark*; hasil diperbarui	*Dashboard* yang boleh berubah sedikit
Side output	Data terlambat dialirkan ke topik terpisah	Rekonsiliasi: *batch* (P6, P9–P10) memperbaiki angka final

Trigger

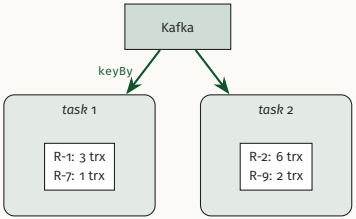
Kapan hasil dikeluarkan: saat *watermark* melewati akhir *window* (*on-time*), lebih awal secara berkala (*early*, hasil sementara), dan setiap ada data terlambat yang diterima (*late*).

Pola bank

Peringatan *fraud* memakai hasil *on-time* (cepat). Angka final untuk laporan dihitung ulang oleh *cold path* dari *lakehouse* yang lengkap. Inilah benih arsitektur *hot/cold* di P12.

State, *checkpoint*, dan *exactly-once*

Keyed state: ingatan per kunci



- *Stream* dipartisi berdasarkan kunci (`rekening_id`); semua *event* satu rekening sampai di *task* yang sama, seperti *partition* Kafka tadi.
- Tiap *task* menyimpan *state* hanya untuk kunci miliknya.

State backend

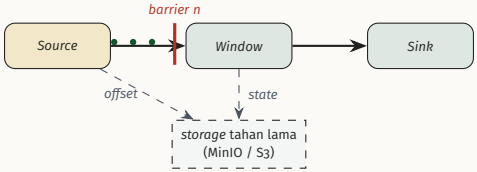
Di memori (*heap*): tercepat, dibatasi RAM, terganggu *garbage collection* bila besar.

RocksDB (LSM-tree, P7): *state* di disk lokal, bisa jauh melebihi RAM, sedikit lebih lambat. Pilihan umum untuk *state* besar di Flink, Kafka Streams, dan Spark.

State yang terus tumbuh

Kunci yang tidak pernah muncul lagi tetap disimpan selamanya kecuali diberi *TTL* atau dibersihkan oleh *watermark*. Pantau ukuran *state*.

Checkpoint: memotret seluruh *job* tanpa menghentikannya



1. *Source* menyisipkan ***barrier*** bernomor ke *stream* dan menyimpan *offset* Kafka-nya.
2. Setiap operator yang menerima *barrier* menyimpan *state*-nya, lalu meneruskan *barrier*.
3. Saat semua operator selesai, *checkpoint n* dinyatakan lengkap.

Kaitan dengan Sister P7

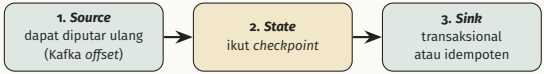
Ini turunan algoritma *snapshot* terdistribusi Chandy-Lamport: memotret keadaan global yang konsisten tanpa jam bersama, dengan menyisipkan penanda ke saluran komunikasi. Flink menyebut versinya *asynchronous barrier snapshotting*.

Saat *job* mati

State dipulihkan dari *checkpoint* terakhir dan *source* kembali ke *offset* yang tersimpan. *Event* sesudahnya dibaca ulang, itu sebabnya *source* harus bisa diputar ulang (Kafka).

Sumber: Carbone dkk., 2015; Hueske dan Kalavri, Bab 3.

Tiga syarat *exactly-once*



- Bila salah satu tidak terpenuhi, jaminannya turun menjadi *at-least-once*.
- ***Sink* transaksional**: hasil ditulis dalam transaksi yang baru di-*commit* ketika *checkpoint* selesai (*two-phase commit*); contoh: Kafka transaksional sebagai *sink*.
- ***Sink* idempoten**: tulis dengan kunci unik (`UPSERT`), sehingga menulis ulang hasil yang sama tidak menggandakan.

Contoh *sink* idempoten

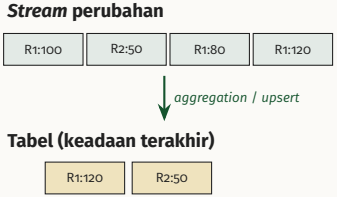
Hasil *window* ditulis ke tabel dengan kunci `(rekening_id, window_start)`. Ditulis dua kali → tetap satu baris dengan nilai yang sama.

Efek samping di luar sistem

Mengirim SMS "rekening Anda diblokir" tidak bisa di-*rollback*. Simpan ID *alert* yang sudah dikirim dan periksa sebelum mengirim lagi.

Sumber: Lanjutan pembahasan *delivery guarantee* di paruh pertama.

Stream dan tabel adalah dua wajah data yang sama



- *Stream* → tabel: menerapkan setiap perubahan (topik *compacted*, paruh pertama).
- Tabel → *stream*: setiap perubahan tabel diterbitkan (CDC, paruh pertama).

Ragam *join*

- ***Stream*–table (*enrichment*)**: transaksi digabung dengan profil nasabah terbaru dari topik CDC `nasabah`. Paling umum.
- ***Stream*–stream (*window join*)**: permintaan OTP dan transaksi dari rekening yang sama dalam 5 menit. *State* kedua sisi disimpan selama *window*.
- ***Temporal join***: gabungkan dengan versi tabel **pada saat** transaksi terjadi, misalnya nisbah bagi hasil yang berlaku hari itu.

When to use: mesin *stream processing*

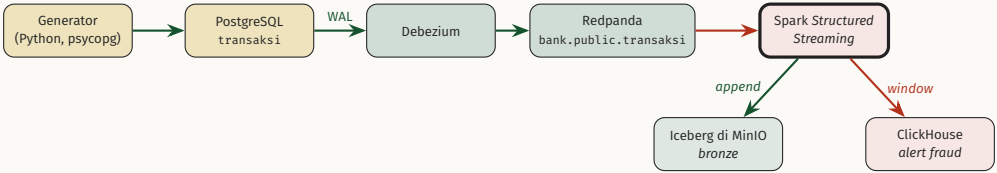
When to use: mesin *stream processing*

Mesin	Model eksekusi	Pakai bila	Pertimbangkan lain bila
Apache Flink	*Record-at-a-time*, *stateful*; Java, Scala, PyFlink, SQL	*Latency* sub-detik, *state* besar, *event time* rumit, CEP	Tim kecil tanpa pengalaman operasi *cluster* JVM
Spark Structured Streaming	*Micro-batch* (*latency* orde ratusan ms sampai detik)	Tim sudah memakai Spark untuk *batch*; satu API untuk keduanya	Butuh *latency* sangat rendah atau *window* yang sangat rumit
Kafka Streams	Pustaka Java di dalam aplikasi; *state* RocksDB + topik *changelog*	Logika *streaming* di dalam *microservice*, tanpa *cluster* terpisah	Sumber bukan Kafka; butuh Python
Apache Beam	Model pemrograman; dijalankan oleh *runner* (Flink, Spark, Dataflow)	Ingin satu kode di beberapa *runner*; memakai Google Cloud Dataflow	Hanya satu mesin; butuh *feature* khas mesin tertentu

Sumber: Karakter umum dari dokumentasi resmi; *latency* bergantung *configuration* dan beban, ukur sendiri.

Praktikum: CDC sampai *hot store*

Alur praktikum hari ini



1. Nyalakan infrastruktur (`docker compose up -d`).
2. Daftarkan *connector* Debezium; amati *event* `c`, `u`, `d`.
3. Jalankan *job* Spark: satu *query* ke *bronze*, satu ke ClickHouse.
4. Jalankan generator: pola curiga, *late event*, *event* di bawah *watermark*.
5. Matikan *job* di tengah, nyalakan lagi: periksa duplikat.
6. (Opsional) bandingkan *throughput* Redpanda dengan NATS JetStream.

Langkah 1: infrastruktur

```
services:
  postgres:
    image: postgres:16
    command: ["postgres", "-c", "wal_level=logical"]
    environment:
      POSTGRES_PASSWORD: rahasia_lab
      POSTGRES_DB: bank
    ports: ["5432:5432"]
  redpanda:
    image: docker.redpanda.com/redpandadata/redpanda
    command:
      - redpanda
      - start
      - --mode=dev-container
      - --kafka-addr=in://0.0.0.0:9092,out://0.0.0.0:19092
      - --advertise-kafka-addr=in://redpanda:9092,out://localhost:19092
    ports: ["19092:19092"]
  connect:
    image: quay.io/debezium/connect
    environment:
      BOOTSTRAP_SERVERS: redpanda:9092
      GROUP_ID: "1"
      CONFIG_STORAGE_TOPIC: cfg
      OFFSET_STORAGE_TOPIC: off
      STATUS_STORAGE_TOPIC: sts
    ports: ["8083:8083"]
```

```
  minio:
    image: minio/minio
    command: server /data --console-address ":9001"
    environment:
      MINIO_ROOT_USER: lab
      MINIO_ROOT_PASSWORD: rahasia_lab
    ports: ["9000:9000", "9001:9001"]
  clickhouse:
    image: clickhouse/clickhouse-server
    environment:
      CLICKHOUSE_PASSWORD: rahasia_lab
    ports: ["8123:8123"]
  nats:                          # opsional
    image: nats
    command: ["-js"]
    ports: ["4222:4222"]
```

Spark dijalankan dari laptop (`pip install pyspark`) dengan paket *connector* lewat `–packages`: Kafka untuk Spark, Iceberg *runtime*, dan `hadoop-aws`. Samakan versi paket dengan versi Spark Anda.

Kata sandi hanya untuk lab. Pada tugas besar, kunci versi *image*.

Langkah 2: mendaftarkan *connector* Debezium

```
import requests

cfg = {
  "name": "bank-cdc",
  "config": {
    "connector.class":
      "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "rahasia_lab",
    "database.dbname": "bank",
    "topic.prefix": "bank",
    "plugin.name": "pgoutput",
    "table.include.list": "public.transaksi",
    "value.converter":
      "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false"
  }
}
r = requests.post("http://localhost:8083/connectors", json=cfg)
print(r.status_code, r.json())
```

- Topik yang terbentuk: `bank.public.transaksi`.
- `schemas.enable=false`: *value* berisi *envelope* `before`, `after`, `op`, `ts_ms` tanpa salinan *schema* di tiap *message*. Di produksi, pakai *schema registry*.
- Amati: `rpk topic consume bank.public.transaksi`, lalu lakukan `UPDATE` dan `DELETE`.
- Lihat *slot*: `SELECT * FROM pg_replication_slots;`

Pertanyaan

Matikan `connect` 5 menit sambil terus menulis. Apa yang terjadi pada ukuran WAL, dan saat dinyalakan lagi?

Langkah 3a: membaca CDC dan menulis *bronze*

```
from pyspark.sql import SparkSession, functions as F, types as T

spark = (SparkSession.builder.appName("hot-path")
  .config("spark.sql.catalog.lake",
          "org.apache.iceberg.spark.SparkCatalog")
  # ... konfigurasi katalog dan S3 MinIO (dibagikan) ...
  .getOrCreate())

AFTER = T.StructType([
  T.StructField("trx_id", T.StringType()),
  T.StructField("rekening_id", T.StringType()),
  T.StructField("device_id", T.StringType()),
  T.StructField("nominal", T.LongType()),
  T.StructField("waktu", T.LongType())])   # mikrodetik
ENV = T.StructType([T.StructField("after", AFTER),
                    T.StructField("op", T.StringType()),
                    T.StructField("ts_ms", T.LongType())])

raw = (spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", "localhost:19092")
  .option("subscribe", "bank.public.transaksi")
  .option("startingOffsets", "earliest").load())
```

```
trx = (raw
  .select(F.from_json(F.col("value").cast("string"),
                      ENV).alias("e"))
  .where(F.col("e.op").isin("c", "r"))
  .select("e.after.*", "e.ts_ms")
  .withColumn("event_time",
              F.expr("timestamp_micros(waktu)")))

bronze = (trx.writeStream
  .format("iceberg").outputMode("append")
  .option("checkpointLocation",
          "s3a://lake/ckpt/bronze_trx")
  .toTable("lake.bronze.transaksi_cdc"))
```

- `op` `c` = *insert*, `r` = *snapshot* awal. `u` dan `d` ditangani terpisah (latihan).
- Kolom `TIMESTAMP` PostgreSQL dikirim Debezium sebagai mikrodetik *epoch*; cek tipe di *event* Anda.
- Setiap *query* punya *checkpoint* sendiri.

Langkah 3b: *window* ke ClickHouse yang *idempotent*

```
import clickhouse_connect

agregat = (trx
  .withWatermark("event_time", "2 minutes")
  .groupBy(F.window("event_time", "1 minute"),
           "rekening_id")
  .agg(F.count("*").alias("n_trx"),
       F.sum("nominal").alias("total"),
       F.approx_count_distinct("device_id")
        .alias("n_perangkat"))
  .where("n_trx > 5 AND n_perangkat > 1"))

def ke_clickhouse(batch_df, batch_id):
    pdf = (batch_df.select(
        F.col("window.start").alias("window_start"),
        "rekening_id", "n_trx", "total", "n_perangkat")
        .toPandas())
    if len(pdf):
        ch = clickhouse_connect.get_client(
            host="localhost", password="rahasia_lab")
        ch.insert_df("peringatan_fraud", pdf)

hot = (agregat.writeStream.outputMode("update")
  .foreachBatch(ke_clickhouse)
  .option("checkpointLocation",
          "s3a://lake/ckpt/fraud_hot")
  .start())
spark.streams.awaitAnyTermination()
```

```
CREATE TABLE peringatan_fraud (
  window_start DateTime,
  rekening_id  String,
  n_trx        UInt32,
  total        Int64,
  n_perangkat  UInt32
) ENGINE = ReplacingMergeTree
ORDER BY (rekening_id, window_start);
```

- Mode `update`: satu *window* bisa dikirim beberapa kali saat datanya bertambah.
- `ReplacingMergeTree` dengan kunci (rekening, *window*) membuat kiriman ulang menimpa, bukan menggandakan: *sink* menjadi *idempotent*. *Query* dengan `FINAL` untuk hasil yang sudah dirapikan.
- `count(distinct)` tidak didukung pada *aggregation streaming*; pakai `approx_count_distinct`.
- `toPandas()` cukup untuk lab; di produksi pakai *connector* JDBC atau penulis khusus.

Langkah 4: generator dengan *late event*

```python
import random, uuid
from datetime import datetime, timedelta, timezone
import psycopg

con = psycopg.connect("postgresql://postgres:rahasia_lab"
                      "@localhost:5432/bank", autocommit=True)

def tulis(rek, dev, menit_lalu=0):
    t = datetime.now(timezone.utc) - timedelta(minutes=menit_lalu)
    con.execute(
        "INSERT INTO transaksi (trx_id, rekening_id, device_id,"
        " nominal, waktu) VALUES (%s, %s, %s, %s, %s)",
        (str(uuid.uuid4()), rek, dev,
         random.randint(10, 500) * 1000, t.replace(tzinfo=None)))

for i in range(7):                          # pola curiga
    tulis("R-10023", f"HP-{i % 2}")
tulis("R-10023", "HP-9", menit_lalu=1)   # terlambat, masih diterima
tulis("R-10023", "HP-9", menit_lalu=5)   # di bawah watermark
```

Skenario uji

1. 7 transaksi dari dua perangkat: muncul satu *alert* di ClickHouse.
2. *Event* terlambat 1 menit: *window* diperbarui?
3. *Event* terlambat 5 menit: masuk *bronze*, tetapi diabaikan oleh *aggregation*? Buktikan.
4. Matikan *job* di tengah, nyalakan lagi: *alert* ganda atau hilang? Baris *bronze* ganda?

Kolom `waktu` adalah *event time*: waktu transaksi terjadi, bukan waktu baris ditulis.

Opsional: *throughput* Redpanda vs NATS JetStream

```python
import time, json
from confluent_kafka import Producer

def ukur(acks, n=200_000, topik="uji"):
    p = Producer({"bootstrap.servers":
                  "localhost:19092", "acks": acks,
                  "linger.ms": 5})
    msg = json.dumps({"nominal": 1}).encode()
    t0 = time.perf_counter()
    for i in range(n):
        p.produce(topik, key=str(i % 1000), value=msg)
        p.poll(0)
    p.flush()
    return n / (time.perf_counter() - t0)

for a in ["0", "1", "all"]:
    print(a, round(ukur(a)), "pesan/detik")
```

```python
import asyncio, nats

async def main():
    nc = await nats.connect("nats://localhost:4222")
    js = nc.jetstream()
    await js.add_stream(name="TRX", subjects=["trx.>"])
    await js.publish("trx.qris", b'{"nominal":1}',
        headers={"Nats-Msg-Id": "T-881"})  # dedup
    sub = await js.pull_subscribe("trx.qris",
                                  durable="fraud")
    for m in await sub.fetch(10, timeout=2):
        await m.ack()
    await nc.close()

asyncio.run(main())
```

Ukur juga dengan 1, 3, dan 6 *partition* (`rpk topic create uji -p 6`) dan bandingkan memori tiap *container* (`docker stats`). Kirim *message* dengan `Nats-Msg-Id` yang sama dua kali: berapa yang tersimpan?

Yang dilaporkan

Bukti

- Tangkapan *event* CDC `c`, `u`, `d` dan jawaban pertanyaan *replication slot*.
- Isi `peringatan_fraud` untuk keempat skenario, dengan penjelasan.
- Jumlah baris *bronze* dan jumlah *snapshot* Iceberg sebelum dan sesudah *restart*.
- Jumlah *event* yang dibuang karena terlambat (Spark UI: *metric state operator*).
- (Opsional) tabel *throughput acks* $\times$ *partition*, median 3 kali.

Pertanyaan analisis

1. Ubah *watermark* dari 2 menjadi 10 menit. Apa yang berubah pada *latency alert* dan ukuran *state*?
2. Mengapa *bronze* tidak ganda setelah *restart*, padahal Kafka hanya *at-least-once*? (Petunjuk: *checkpoint* + *commit* Iceberg.)
3. Apa yang terjadi bila `peringatan_fraud` memakai `MergeTree` biasa?

Kejujuran pengukuran

Satu laptop, satu *broker*, tanpa *replication*: angka Anda **bukan** angka produksi.

Penutup

Best practice hot path yang kita pegang

1. ***Key* sesuai urutan bisnis**: `rekening_id`, bukan acak.
2. **Data penting**: `acks=all`, *replication* 3, `min.insync.replicas=2`, *idempotence* aktif.
3. **CDC atau *outbox***, bukan *dual write*; *schema* terdaftar.
4. **Minimalkan PII di topik**; kirim ID atau *token*. Retensi sesuai kebutuhan *replay* dan aturan privasi.
5. **Pantau *consumer lag*** dan ukuran *replication slot*.
6. ***Event time*, bukan *processing time***, untuk hasil yang harus dapat diulang.
7. ***Watermark* dari data**: ukur sebaran keterlambatan dulu.
8. ***Late data* tidak dibuang diam-diam**: hitung, atau simpan di *bronze*.
9. ***Checkpoint* ke *storage* tahan lama** dan *sink idempotent*, diuji dengan *restart* sungguhan.
10. **Angka final dari *cold path***; *hot path* untuk keputusan cepat.

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum sesuai *slide* “Yang dilaporkan”.
2. Tangani `op = u` dan `op = d` di *bronze* (simpan sebagai *event*, jangan ditimpa), lalu jelaskan dampaknya pada *silver*.
3. Tambahkan satu aturan kedua (lonjakan nominal terhadap rata-rata 30 hari dari *gold*) dan jelaskan *state* atau *join* yang dibutuhkan.
4. Baca Shapira dkk., *Kafka: The Definitive Guide*, Bab 3–4; Akidau dkk., *Streaming Systems*, Bab 1–3; Chambers dan Zaharia, Bab 20–22.
5. Tugas besar UAS: *hot path* adalah nilai tambah. Bila memilihnya, tentukan sumber *event*, *broker*, dan empat pertanyaan Dataflow untuk *job* Anda.

Pertemuan depan

Menyatukan semuanya: arsitektur *hot* dan *cold* di industri, Lambda vs Kappa, *serving layer* (ClickHouse, OpenSearch, Redis), SLO kesegaran data, dan *cost*-nya, serta penilaian rancangan arsitektur tugas besar UAS.

Terima kasih

Pertanyaan?