Pertemuan 10 — Spark Skala Besar: *HPC Data*

Catalyst dan AQE, *join* dan *skew*, *partition*, *tuning*, lalu membuktikan *scaling* dengan angka

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Dari *job* yang jalan ke *job* yang cepat
2. Catalyst, *physical plan*, dan AQE
3. Strategi *join*, *data skew*, *salting*
4. *Partitioning* dan ukuran *file*
5. Menulis ke Iceberg/Delta: MERGE dan *compaction*
6. *Tuning*: memori, Kryo, *spill*, *dynamic allocation*
7. Menjalankan Spark: *standalone*, Kubernetes, *event log*
8. *HPC data*: *strong* dan *weak scaling*
9. Spark vs *baseline* satu mesin dari UTS
10. *When to use* dan *best practice*
11. Praktikum dan menuju UAS

Sasaran pertemuan ini

Anda dapat membuat *pipeline* Spark berjalan efisien di *cluster*, menjelaskan setiap perbaikan dengan bukti dari *physical plan* dan Spark UI, dan membuktikan dengan pengukuran **seberapa baik** *pipeline* Anda menskala, bahan inti UAS.

Dari *job* yang jalan ke *job* yang cepat

Empat angka yang selalu dicatat

Angka	Di mana melihatnya
Durasi *job* dan tiap *stage*	Spark UI: Jobs, Stages
Shuffle read/write (byte)	Spark UI: Stages
Spill memori dan disk	Spark UI: Stages, detail *task*
Cost (*core* × jam)	Jumlah *executor* × durasi

Siklus perbaikan

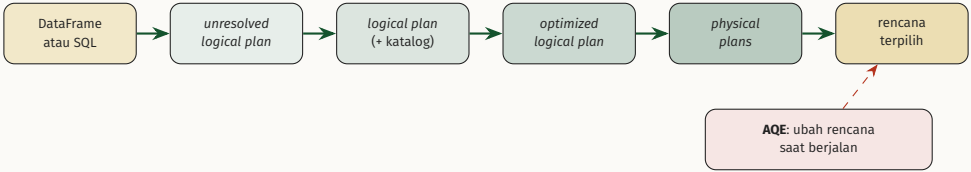
1. Ukur keadaan awal (*baseline*)
2. Baca `explain()` dan Spark UI: cari *stage* terlama
3. Ubah **satu** hal
4. Ukur ulang dengan data dan *cluster* yang sama
5. Simpan atau buang perubahan berdasarkan angka

Ringkasan

Mengubah lima *configuration* sekaligus lalu *job* terasa lebih cepat adalah tebakan. *Tuning* berarti mengubah satu hal, lalu mengukur hasilnya.

Catalyst, *physical plan*, dan AQE

Catalyst: dari *query* ke *physical plan*



Rule-based optimization yang penting dikenali

- ***Predicate pushdown***: filter didorong ke pembaca Parquet; *row group* yang tidak cocok dilewati (P2).
- ***Column pruning***: hanya kolom yang dipakai yang dibaca.
- ***Constant folding***, penyederhanaan ekspresi, urutan *join*.

Adaptive Query Execution (AQE)

- Aktif secara *default* sejak Spark 3.2 (cek versi)
- Menyatukan *partition* kecil setelah *shuffle*
- Mengganti *sort-merge join* menjadi *broadcast join* bila ternyata satu sisi kecil
- Memecah *partition* yang *skewed*

Sumber: Armbrust dkk., Spark SQL (SIGMOD 2015); dokumentasi Spark, *Performance Tuning*.

AQE lebih dekat: tiga hal yang diubah saat berjalan

- *Optimizer* biasa memutuskan rencana **sebelum** data dibaca, berdasarkan perkiraan. AQE memakai **statistik nyata** di setiap batas *stage*, lalu menyusun ulang sisa rencana.
- ***Coalesce partitions***: 200 *shuffle partition* yang kecil-kecil digabung menjadi lebih sedikit *partition* berukuran wajar.
- ***Switch join strategy***: *sort-merge join* diganti *broadcast join* bila salah satu sisi ternyata kecil setelah difilter.
- ***Skew join***: *partition* yang jauh lebih besar dari median dipecah menjadi beberapa *task*.

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set(
  "spark.sql.adaptive.coalescePartitions.enabled",
  "true")
spark.conf.set(
  "spark.sql.adaptive.skewJoin.enabled", "true")
spark.conf.set(
  "spark.sql.adaptive.advisoryPartitionSizeInBytes",
  "128m")
```

Di `explain()` terlihat sebagai `AdaptiveSparkPlan isFinalPlan=true` setelah *job* selesai; rencana akhir berbeda dari rencana awal.

Sumber: Dokumentasi Spark, *Adaptive Query Execution*. Nama dan nilai *default configuration*: cek versi Spark yang dipakai.

Membaca *physical plan*

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[tanggal, cabang], functions=[sum(nominal)])
   +- Exchange hashpartitioning(tanggal, cabang, 200)
      +- HashAggregate(keys=[tanggal, cabang],
                       functions=[partial_sum(nominal)])
         +- Project [tanggal, nominal, cabang]
            +- BroadcastHashJoin [id_nasabah], [id_nasabah]
               :- Filter (tanggal >= 2026-10-01)
               :  +- FileScan parquet [id_nasabah,tanggal,nominal]
               :     PushedFilters: [GreaterThanOrEqual(tanggal,...)]
               +- BroadcastExchange HashedRelationBroadcastMode
                  +- FileScan parquet [id_nasabah,cabang]
```

Output `harian.explain()`, diringkas.

- Baca **dari bawah ke atas**.
- `FileScan … [3 kolom]`: *column pruning* bekerja.
- `PushedFilters`: filter tanggal didorong ke Parquet.
- `BroadcastHashJoin`: tabel nasabah cukup kecil, tidak ada *shuffle* untuk *join*.
- `Exchange`: **satu-satunya** *shuffle*, untuk *groupBy*.
- `partial_sum` sebelum `Exchange`: *aggregation* sebagian mengurangi data yang dikocok.

Strategi *join* dan *data skew*

Tiga strategi *join* di Spark

Strategi	Cara kerja	Cocok bila
Broadcast hash join	Tabel kecil dikirim utuh ke semua *executor*; tabel besar **tidak** dikocok	Satu sisi kecil (dimensi cabang, produk akad). Ambang otomatis `autoBroadcastJoinThreshold` (*default* sekitar 10 MB, cek versi)
Shuffle hash join	Kedua sisi dikocok per *key*, sisi lebih kecil dibuat *hash table* per *partition*	Satu sisi jauh lebih kecil tetapi tidak muat untuk di-*broadcast*
Sort-merge join	Kedua sisi dikocok dan diurutkan per *key*, lalu digabung berurutan	Dua tabel besar (transaksi × mutasi rekening). Pilihan *default* untuk *equi-join* besar

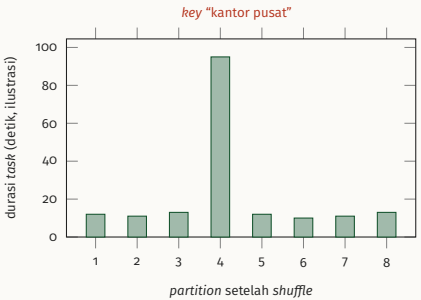
Memaksa *broadcast* ketika Anda tahu tabelnya kecil:

```
trx.join(F.broadcast(cabang), "id_cabang")
```

Jebakan

Mem-*broadcast* tabel yang ternyata besar membuat *driver* atau *executor* kehabisan memori. Ukur dulu ukurannya.

Data skew: satu *task* menahan semuanya



- ***Skew***: sebagian kecil *key* memuat sebagian besar baris. Contoh: transaksi QRIS satu *merchant* besar, atau rekening penampung kantor pusat.
- *Stage* baru selesai ketika *task* paling lambat selesai.
- Gejala di Spark UI: median durasi *task* detik, maksimum menit.

Penanganan

- Biarkan AQE memecah *skewed partition* (`skewJoin.enabled`)
- ***Salting***: tambah akhiran acak pada *key* berat, *join* dengan dimensi yang digandakan sebanyak akhiran itu
- Proses *key* berat secara terpisah

Salting dalam beberapa baris

```
N = 16  # jumlah garam; pilih dari distribusi kunci, bukan tebakan
trx_s = trx.withColumn("garam", (F.rand() * N).cast("int"))
mch_s = (merchant
         .withColumn("garam", F.explode(F.array([F.lit(i) for i in range(N)]))))

hasil = (trx_s.join(mch_s, ["id_merchant", "garam"])
              .groupBy("id_merchant").agg(F.sum("nominal").alias("total")))
```

Satu *key* berat kini tersebar ke 16 *partition*. *Cost*-nya: tabel dimensi menjadi 16 kali lebih besar, sebab itu hanya dipakai bila dimensinya kecil.

Ringkasan

Coba AQE dulu. *Salting* manual adalah alat terakhir, dan selalu ukur sebelum dan sesudahnya.

Partitioning dan ukuran *file*

Partition output dan ukuran *file*

Small files problem

200 *shuffle partition* × 30 nilai `tanggal` bisa menghasilkan ribuan *file* kecil per hari. Akibatnya: *listing object storage* lambat, metadata membengkak, pembaca berikutnya membuka terlalu banyak *file*.

Pedoman umum

- Ukuran *file* Parquet di kisaran ratusan MB (pedoman praktik, bukan aturan)
- `repartition("tanggal")` sebelum `partitionBy("tanggal")`
- *Partition* menurut kolom yang sering difilter, dengan kardinalitas rendah

Jangan *partition* menurut `id_nasabah`

Sejuta nasabah = sejuta folder kecil. *Partition* menurut tanggal atau bulan; kolom berkardinalitas tinggi lebih cocok untuk *sorting* atau *clustering* di dalam *file*.

Iceberg dan Delta (P4) punya *compaction* dan *hidden partitioning* untuk mengatasi masalah ini secara sistematis, lihat bagian berikutnya.

Berapa jumlah *partition* yang pas?

Dua batas

- **Terlalu sedikit**: setiap *task* besar, memori tidak cukup, *spill*, dan sebagian *core* menganggur.
- **Terlalu banyak**: *cost* penjadwalan per *task*, *file output* kecil-kecil.

Pedoman awal (bukan aturan)

- Ukuran *shuffle partition* di kisaran ratusan MB
- Jumlah *partition* minimal beberapa kali jumlah total *core*, agar *task* lambat tidak menahan semuanya
- Biarkan AQE menyesuaikan, lalu periksa hasilnya

```
# repartition: shuffle penuh, bisa menambah
#              atau mengurangi; bisa per kolom
gold = gold.repartition("tanggal")
# coalesce: hanya mengurangi, tanpa shuffle
kecil = laporan.coalesce(1)
# bucketing: data sudah terkelompok per key,
# join berikutnya pada key itu tanpa shuffle
(trx.write.bucketBy(64, "id_nasabah")
    .sortBy("id_nasabah")
    .saveAsTable("silver.trx_bucketed"))
```

Bucketing berguna untuk *join* berulang pada *key* yang sama; *cost*-nya penulisan yang lebih berat. Ukur sebelum memakainya.

Menulis ke Iceberg dan Delta

MERGE: *upsert* ke *layer* silver dan gold

```sql
-- data koreksi transaksi dari bronze hari ini
MERGE INTO lake.silver.transaksi t
USING koreksi_hari_ini k
ON t.id_transaksi = k.id_transaksi
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *;

-- gold: timpa hanya partisi tanggal yang diproses
INSERT OVERWRITE lake.gold.harian_cabang
SELECT tanggal, cabang, SUM(nominal) AS total
FROM lake.silver.transaksi
WHERE tanggal = DATE '2026-12-06'
GROUP BY tanggal, cabang;
```

- Iceberg dan Delta (P4) memberi ACID di atas *object storage*; Spark menjadi mesin tulisnya untuk data besar.
- `MERGE` membuat *pipeline idempotent*: menjalankan ulang hari yang sama tidak menggandakan baris.
- `INSERT OVERWRITE` per *partition* (*dynamic partition overwrite*) adalah pola *backfill* yang aman (P6).
- *Copy-on-write* vs *merge-on-read* menentukan apakah *cost* dibayar saat tulis atau saat baca.

Merawat tabel: *compaction* dan *snapshot expiration*

```
-- gabungkan berkas kecil hasil streaming/MERGE
CALL lake.system.rewrite_data_files(
  table => 'silver.transaksi',
  options => map('target-file-size-bytes',
                 '268435456'));

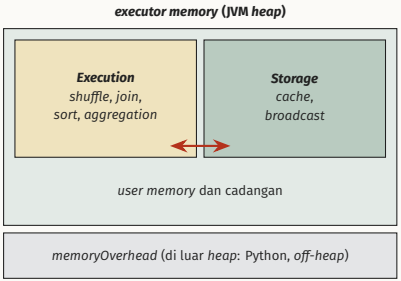
-- buang snapshot lama (time travel terbatas)
CALL lake.system.expire_snapshots(
  table => 'silver.transaksi',
  older_than => TIMESTAMP '2026-11-01 00:00:00');
```

Sintaks prosedur Iceberg di Spark SQL; Delta memakai `OPTIMIZE` dan `VACUUM`. Cek versi.

- *Streaming* (P11) dan MERGE yang sering menghasilkan banyak *file* kecil; *compaction* mengembalikan ukuran *file* ke kisaran wajar.
- *Snapshot expiration* membuang metadata dan *file* yang tidak lagi dirujuk.
- Tanpa *expiration*, data yang sudah di-DELETE masih ada di *snapshot* lama, penting untuk *right to erasure* UU PDP (P15).
- Jadwalkan keduanya sebagai *job* perawatan di Airflow/Dagster (P6).

Tuning: memori, *serialization*, *spill*

Model memori *executor*



Execution dan *storage* berbagi satu wilayah (*unified memory*); batasnya bergeser sesuai kebutuhan.

- Bila *execution* kekurangan memori, data ***spill*** ke disk: benar, tetapi lambat. Terlihat di Spark UI sebagai *spill (memory/disk)*.
- *Executor* terlalu besar: jeda GC panjang (lihat Sistem Terdistribusi P10). Terlalu kecil: *broadcast* dan *shuffle* tidak muat.
- Praktik umum: beberapa *core* per *executor* (bukan satu, bukan puluhan), memori seimbang dengan jumlah *core*.
- PySpark dengan UDF butuh *memoryOverhead* lebih besar karena ada proses Python.

Configuration yang paling sering disentuh

Configuration	Gunanya
`spark.executor.memory`, `.cores`	Ukuran tiap *executor*
`spark.executor.memoryOverhead`	Memori di luar *heap* (Python, *off-heap*, *container*)
`spark.sql.shuffle.partitions`	Jumlah *shuffle partition* awal (AQE dapat menyesuaikan)
`spark.sql.autoBroadcastJoinThreshold`	Batas ukuran tabel yang otomatis di-*broadcast*
`spark.serializer` = `KryoSerializer`	*Serialization* lebih ringkas untuk RDD dan *shuffle* objek
`spark.dynamicAllocation.enabled`	Menambah/mengurangi *executor* sesuai *queue task*
`spark.speculation`	Menjalankan salinan *task* lambat (*straggler*, P9)
`spark.eventLog.enabled`, `.dir`	Menyimpan riwayat untuk *history server*

Untuk DataFrame dan SQL, Spark memakai format biner internal (Tungsten), sehingga Kryo paling terasa pada kode RDD dan objek kustom.

Nilai *default* berbeda antar versi dan antar *cluster manager*. Selalu cek dokumentasi versi Anda dan catat *configuration* di kode.

Menjalankan Spark: *standalone*, Kubernetes, *event log*

Dari laptop ke *cluster* sungguhan

Mode	Kapan dipakai
`local[*]`	Pengembangan, data kecil, uji unit
Standalone	Lab, *cluster* kecil khusus Spark (P9)
Kubernetes	Platform modern: *executor* sebagai *pod*, berbagi *cluster* dengan layanan lain
YARN	*Cluster* Hadoop yang sudah ada
Terkelola	EMR, Dataproc, Databricks: operasi diserahkan ke penyedia

```
spark-submit \
  --master k8s://https://api.klaster:6443 \
  --deploy-mode cluster \
  --conf spark.kubernetes.container.image=\
registry.kampus/spark-bank:1.0 \
  --conf spark.executor.instances=4 \
  --conf spark.eventLog.enabled=true \
  --conf spark.eventLog.dir=s3a://logs/spark/ \
  local:///opt/job/gold_harian.py
```

Di Kubernetes, *driver* dan *executor* berjalan sebagai *pod*; Spark Operator membantu mengelolanya secara deklaratif.

Ringkasan

Spark UI hilang ketika aplikasi selesai. Simpan *event log* dan jalankan *history server* agar pengukuran bisa diperiksa ulang, termasuk saat penilaian UAS.

HPC data: membuktikan *scaling*

Strong dan *weak scaling*

Strong scaling

Data **tetap**, jumlah *executor* dinaikkan. Pertanyaannya: bila *resource* digandakan, apakah waktu menjadi setengah?

$$S(p) = \frac{T(1)}{T(p)}, \qquad E(p) = \frac{S(p)}{p}$$

Weak scaling

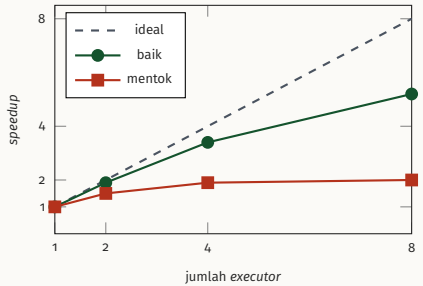
Data **per *executor* tetap**: data dan *executor* dinaikkan bersama. Pertanyaannya: bila data dan *resource* sama-sama digandakan, apakah waktu tetap?

$$E_w(p) = \frac{T(1, n)}{T(p, p \cdot n)}$$

Mengapa jarang ideal: bagian serial (membaca metadata, *driver*), *shuffle* yang tumbuh dengan jumlah *partition*, *skew*, dan *cost* penjadwalan, hukum Amdahl dan Gustafson yang dibahas di Sistem Terdistribusi P9.

Sumber: $T(p)$: waktu dengan p *executor* (atau *core*); n: ukuran data per *executor*; $S(p)$: *speedup*; $E(p)$: efisiensi *strong scaling*; $E_w(p)$: efisiensi *weak scaling*. Nilai ideal $S(p) = p$ dan $E = 1$.

Bentuk kurva yang akan Anda temui



Angka ilustrasi, bukan hasil pengukuran.

- **Mendekati ideal**: pekerjaan didominasi *scan* dan *aggregation* lokal, *shuffle* kecil.
- **Melengkung lalu datar**: bagian serial atau *shuffle* mendominasi; menambah *executor* hanya menambah *cost*.
- **Turun**: *executor* terlalu banyak untuk data sekecil itu, *cost* koordinasi lebih besar daripada kerja.
- Selalu laporkan **efisiensi** dan ***cost*** (*core* $\times$ jam), bukan hanya waktu tercepat.

Protokol pengukuran yang jujur

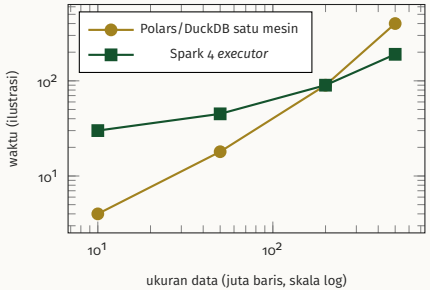
1. **Tetapkan beban**: *query*, versi kode, dan data yang sama untuk semua percobaan.
2. **Pemanasan**: satu kali jalan dibuang (JVM, *cache file*).
3. **Ulangi**: minimal tiga kali, laporkan median dan rentang.
4. **Ubah satu variabel**: jumlah *executor atau* ukuran data *atau* satu *configuration*.
5. **Catat konteks**: ukuran mesin, *core*, memori, versi Spark, *configuration*.
6. **Simpan bukti**: *event log*, tangkapan Spark UI, `explain()`.

Jebakan di laptop

Semua *worker* Docker berbagi CPU, memori, dan disk yang sama. Menambah *worker* melebihi jumlah *core* fisik tidak menambah tenaga. Batasi `SPARK_WORKER_CORES` sesuai jumlah *core* yang benar-benar ada, dan jelaskan batas ini di laporan.

Bila tersedia, ulangi percobaan di beberapa mesin lab atau di *cloud* kecil untuk melihat *scaling* lintas mesin.

Spark vs *baseline* satu mesin dari UTS



Bentuk ilustrasi; titik silangnya bergantung pada data, *query*, dan *hardware* Anda.

- Untuk data kecil, satu mesin menang jauh: Spark membayar *cost* tetap (JVM, penjadwalan, *shuffle* lewat jaringan).
- Seiring data membesar, satu mesin mulai *spill* atau kehabisan memori; Spark menyusul karena bebannya terbagi.
- Pertanyaan UAS: **di mana titik silangnya** untuk *pipeline* Anda, dan apa buktinya?
- Mengapa satu mesin bisa secepat itu (*vectorized execution*, SIMD, *cache*) dijelaskan di Sistem Terdistribusi P10, Sabtu 5 Des.

Akselerasi GPU: RAPIDS Accelerator for Apache Spark

- *Plugin* yang menjalankan operasi DataFrame dan SQL Spark di GPU tanpa mengubah kode aplikasi.
- Operasi yang belum didukung otomatis kembali ke CPU; rencana fisik menunjukkan operator mana yang berjalan di GPU.
- Paling terasa untuk *scan*, *join*, dan *aggregation* besar atas format *columnar*; sedikit manfaatnya bila *job* didominasi I/O jaringan.

Bukan jalan pintas

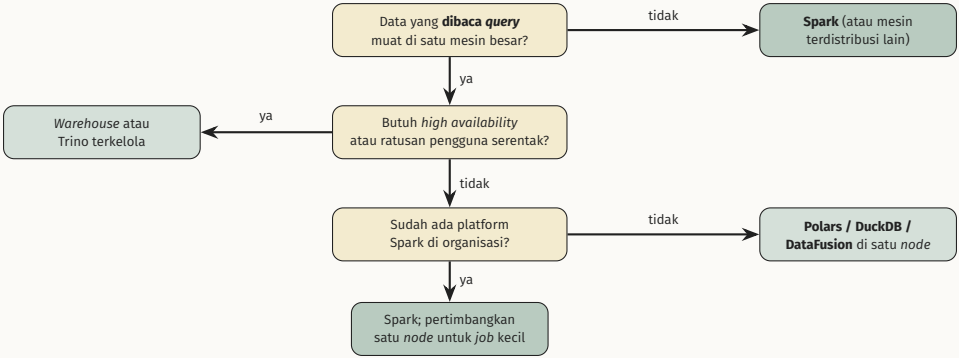
GPU mahal dan tidak otomatis lebih cepat. Sama seperti menambah *executor*: ukur dulu, bandingkan *cost* per *job*, baru putuskan.

Opsional untuk UAS bila kelompok punya akses GPU; tidak wajib.

Sumber: Dokumentasi NVIDIA RAPIDS Accelerator for Apache Spark; cek kompatibilitas versi Spark.

When to use dan *best practice*

Satu mesin atau *cluster*: alur keputusan



Alur ini penyederhanaan untuk memulai diskusi, bukan aturan baku. Faktor lain: kemampuan tim, kebutuhan *streaming*, kebijakan keamanan, *license cost*.

When to use: mesin *batch*

Mesin	Kekuatan	Hati-hati
Spark (Scala)	Skala *cluster*, ekosistem luas, *connector* lengkap, *Dataset* bertipe	*Cost* JVM dan operasi; berlebihan untuk data kecil
PySpark	Sama dengan Spark untuk DataFrame dan SQL, mudah bagi analis	UDF Python biasa lambat; *debugging* melewati dua *runtime*
Polars	Sangat cepat di satu mesin, hemat memori, API bersih	Satu mesin; API masih bergerak cepat antar versi
DuckDB	SQL analitik tanpa server, membaca Parquet dan Iceberg langsung	Tidak untuk banyak penulis serentak atau OLTP
DataFusion	Fondasi untuk membangun mesin atau produk sendiri; Rust dan Arrow	Sebagai alat analis, ekosistemnya lebih kecil dari DuckDB/Polars
Ballista	Eksekusi terdistribusi tanpa JVM	Belum sematang Spark untuk produksi

Best practice industri untuk *batch*

1. **Mulai dari satu mesin.** Pindah ke *cluster* hanya bila ukuran data, *concurrency*, atau *availability* menuntutnya, dan buktikan dengan angka.
2. **Pakai fungsi bawaan dan SQL** sebelum UDF.
3. **Filter dan pangkas kolom sedini mungkin**; biarkan pushdown bekerja.
4. **Baca rencana** (`explain`) untuk setiap *job* penting.
5. ***Broadcast* dimensi kecil**, waspadai *skew* pada *key* besar.
6. ***Partition output* menurut waktu**, jaga ukuran *file* wajar.
7. **Tulis Parquet**, bukan CSV, di antara tahap.
8. ***Job idempotent***: menjalankan ulang satu tanggal menimpa *partition* itu, tidak menggandakan (P6).
9. **Simpan *configuration* di kode**, bukan di klik konsol.
10. **Catat ukuran input, waktu, dan *cost*** tiap *run* untuk melihat tren.

Praktikum

Praktikum: *pipeline* bronze–silver–gold skala besar

1. Bangkitkan transaksi sintetis bank syariah dengan generator dari P9 pada tiga ukuran (misalnya 50, 200, 500 juta baris; sesuaikan kemampuan laptop).
2. Jalankan *pipeline* bronze → silver (dedup, *tokenization* PII sederhana) → gold (agregat harian per cabang) dengan Spark ke Iceberg/Delta di MinIO.
3. Eksperimen *join*: *broadcast* vs *sort-merge* (`autoBroadcastJoinThreshold=-1` untuk memaksa).
4. Eksperimen *skew*: *merchant* QRIS "berat"; AQE *skew join* aktif vs nonaktif, lalu *salting*.
5. *Strong scaling*: data 200 juta, *executor* 1, 2, 4. *Weak scaling*: 50 juta per *executor*.
6. *Baseline*: *query* gold yang sama dengan Polars/DuckDB (dari UTS).

Executor	Waktu	*Speedup*	Efisiensi	*Shuffle*
1	…	1,00	1,00	…
2	…	…	…	…
4	…	…	…	…
Polars/DuckDB	…	—	—	—

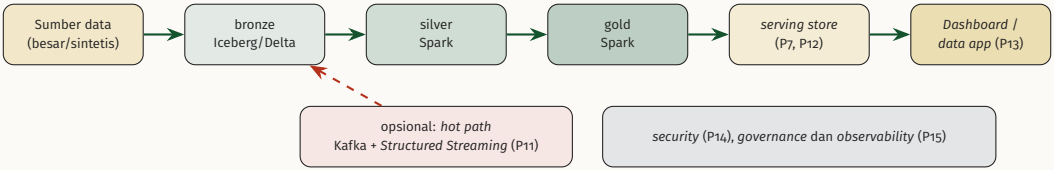
Isi dengan median tiga kali jalan, sertakan *event log*. Jelaskan **mengapa** kurvanya berbentuk begitu, dengan bukti dari `explain()` dan Spark UI.

Data sintetis

Nama, NIK, dan nomor rekening dibangkitkan acak. Jangan memakai data sungguhan.

Menuju UAS

Menuju UAS: *pipeline* skala besar sampai *dashboard* atau aplikasi



Yang dibawa dari hari ini

- *Pipeline* Spark bronze–silver–gold yang berjalan di beberapa *worker*
- Tabel dan grafik *strong*/*weak scaling* beserta *event log*
- Perbandingan dengan *baseline* satu mesin dari UTS

Yang masih akan ditambahkan

- *Serving* dan *dashboard* atau *data app* (P12–P13)
- *Tokenization* PII dan *role* (P14); katalog, *lineage*, *data contract* (P15)
- Rancangan arsitektur dinilai di P12; presentasi progres di P15

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum: tabel dan grafik *strong* dan *weak scaling*, perbandingan *join* dan *skew* (sebelum/sesudah), dan titik silang Spark vs Polars/DuckDB, semuanya dengan bukti `explain()` dan Spark UI.
2. Pilih *dataset* dan domain tugas besar UAS; tulis satu paragraf: berapa besar datanya dan mengapa butuh Spark.
3. Baca Shapira dkk., *Kafka: The Definitive Guide*, Bab 1–4, dan Akidau dkk., *Streaming Systems*, Bab 1–2.

Pertemuan depan

Hot path: Kafka, Redpanda, dan NATS JetStream, *change data capture* dengan Debezium, lalu Spark *Structured Streaming* dengan *window* dan *watermark*, jalur yang menulis ke *layer* bronze yang sama.

Terima kasih

Pertanyaan?