Pertemuan 5 — ETL di Satu Mesin

Apache Arrow, Polars, DuckDB, dan DataFusion: mengisi bronze, silver, gold tanpa *cluster*

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

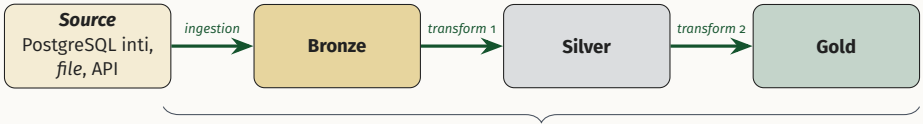
1. Posisi ETL di antara bronze, silver, gold
2. *Ingestion*: *full* vs *incremental load*
3. ETL vs ELT, dan alat *ingestion*
4. *Scale-up* dulu: perlukah *cluster*?
5. Apache Arrow dan *vectorized execution*
6. Polars, DuckDB, dan DataFusion
7. *Pipeline* bronze → silver → gold
8. Menulis secara *idempotent*
9. *When to use* dan *best practice*
10. Praktikum: *pipeline* + *benchmark*

Sasaran pertemuan ini

Anda dapat membangun *transformation* bronze → silver → gold yang cepat di satu laptop dengan Polars dan DuckDB, menjelaskan mengapa Arrow membuatnya cepat, dan memilih *engine* berdasarkan ukuran data dan pola aksesnya.

Posisi ETL di *pipeline*

Dari P4 ke hari ini



Yang sudah ada

P2: *star schema* dan Parquet. P3: MinIO dan *partition* tanggal. P4: aturan tiap *layer* dan tabel Iceberg/Delta.

Yang ditambah hari ini

Cara memindahkan data ke bronze, lalu **kode *transformation*** yang mengisi silver dan gold dengan cepat. P6 menjadwalkan dan mengujinya.

Ingestion: full vs incremental

Tiga cara menarik data dari *source*

Cara	Bagaimana	Kelebihan	Kekurangan
Full load	Salin seluruh tabel setiap kali	Sederhana, selalu lengkap	Mahal untuk tabel besar; membebani *source*
Incremental (*high-water mark*)	Ambil baris dengan `updated_at` > nilai terakhir	Hemat; cocok untuk tabel besar yang terus tumbuh	Baris yang dihapus tidak terlihat; jam *source* harus jujur
CDC	Baca *log* perubahan *database*	Lengkap termasuk DELETE, hampir *real-time*	Butuh akses *log*, lebih rumit (P11)

Di bank syariah kita

Tabel cabang dan produk (kecil, jarang berubah): *full load*. Transaksi dan angsuran (besar): *incremental*. Saldo dan transaksi untuk deteksi *fraud*: CDC di P11.

Jebakan *incremental*

Transaksi yang tercatat terlambat (*late arriving*) dengan `updated_at` lama akan terlewat. Ambil jendela tumpang tindih (misalnya mundur satu jam) lalu *deduplicate* di silver.

Incremental load dengan *high-water mark*

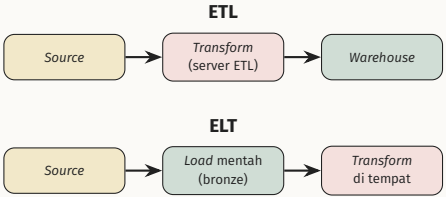
```
import duckdb, json, pathlib
st = pathlib.Path("state/trx_hwm.json")
hwm = json.loads(st.read_text())["hwm"] \
      if st.exists() else "1970-01-01"

con = duckdb.connect()
con.sql("INSTALL postgres; LOAD postgres;")
con.sql("ATTACH 'dbname=core host=localhost "
        "user=etl' AS core (TYPE postgres, READ_ONLY)")
batch = con.sql(f"""
  SELECT *, now() AS _ingest_ts,
         'core.transaksi' AS _source
  FROM core.public.transaksi
  WHERE updated_at > TIMESTAMP '{hwm}'
        - INTERVAL 1 HOUR      -- jendela tumpang tindih
""")
batch.write_parquet(f"bronze/trx/batch={run_id}.parquet")
new = con.sql("SELECT max(updated_at) "
              "FROM batch").fetchone()[0]
st.write_text(json.dumps({"hwm": str(new)}))
```

- *High-water mark* disimpan **setelah** *batch* berhasil ditulis. Bila gagal di tengah, *run* berikutnya mengulang dari titik lama.
- Bronze hanya *append*; duplikat dari jendela tumpang tindih dibuang di silver.
- Kolom `_ingest_ts` dan `_source` sesuai aturan bronze (P4).
- Akses ke *source* memakai *user* khusus, hanya baca.

Contoh disederhanakan: `run_id` dari *orchestrator* (P6); di produksi, kredensial dari *secret manager* (P14) dan parameter tidak disusun dengan *f-string*.

ETL vs ELT



ETL: data diubah dulu baru disimpan. ELT: simpan mentah dulu, ubah di dalam *storage* atau *engine* analitik.

- **ELT** menjadi pola umum sejak *storage* murah dan *engine* analitik kuat: data mentah tersimpan, *transformation* bisa diulang (inilah bronze).
- **ETL** masih tepat bila data **tidak boleh** disimpan mentah, misalnya PII yang harus di-*mask* sebelum keluar dari jaringan *source*.
- Medallion adalah ELT yang bertingkat: *load* ke bronze, *transform* ke silver dan gold.

Ringkasan

Yang perlu diputuskan adalah **di mana** *transformation* paling aman dan murah dijalankan, sebelum memilih istilah ETL atau ELT.

Alat *ingestion*: *when to use*

Alat	Pakai bila	Hati-hati
Skrip sendiri (Python, DuckDB)	Sumber sedikit, logika khas, ingin kendali penuh	Anda memelihara semuanya: *retry*, *schema change*, *state*
dlt (*library* Python)	Ingin *pipeline* sebagai kode Python biasa dengan *incremental* dan *schema evolution* bawaan	Konektor lebih sedikit dari platform besar
Airbyte	Ratusan konektor siap pakai, ada versi terbuka yang bisa di-*host* sendiri	Butuh operasi sendiri; perhatikan versi dan lisensi konektor
Fivetran	Layanan terkelola, nol operasi	Berbayar per volume; data lewat pihak ketiga (cek kepatuhan)
Debezium	CDC dari *database* ke Kafka	Dibahas di P11

Untuk data nasabah bank, pertanyaan pertama adalah **kepatuhan**: boleh tidak data ini melewati layanan pihak ketiga? Jawabannya sering menentukan alat sebelum *feature*.

Scale-up dulu

Pertanyaan yang jarang ditanyakan: perlukah *cluster*?

Mesin satu *node* modern bisa punya puluhan *core*, ratusan GB RAM, dan SSD NVMe berkecepatan beberapa GB/s. Banyak data "besar" di perusahaan muat di situ.

Hitungan kasar kasus kita

1,8 TB transaksi per tahun (P1) disimpan sebagai Parquet terkompresi bisa menyusut beberapa kali. *Query* yang hanya menyentuh 3 kolom dan 1 bulan data mungkin hanya membaca beberapa GB, muat di satu laptop.

Cluster cost yang sering dilupakan

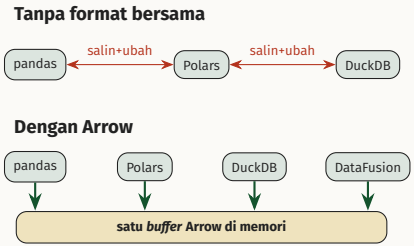
- *Shuffle* lewat jaringan
- Waktu mulai *executor* dan JVM
- *Serialization* antar-proses
- *Operational cost*: *configuration*, *monitoring*, *upgrade*

Sumber: McSherry, Isard, Murray, *Scalability! But at what COST?*, HotOS 2015.

Ringkasan

Scale-out menyelesaikan masalah ukuran dengan menambah *cost* koordinasi. *Scale-up* menghindari *cost* itu selama datanya masih muat. *Cluster* datang di P9–P10.

Apache Arrow: format *columnar* di memori



- **Arrow** mendefinisikan *layout* kolom di memori: nilai satu kolom berurutan, ditambah *validity bitmap* untuk *null*.
- ***Zero-copy***: *library* yang sama-sama paham Arrow dapat memakai *buffer* yang sama tanpa menyalin dan mengubah format.
- ***Arrow Flight***: protokol mengirim data Arrow antar-proses dan antar-mesin tanpa konversi baris.
- Parquet adalah format *columnar* **di disk**; Arrow adalah pasangannya **di memori**.

Zero-copy dalam praktik

```
import duckdb, polars as pl, pyarrow.parquet as pq

tbl = pq.read_table("silver/trx.parquet")  # Arrow
df  = pl.from_arrow(tbl)        # Polars, umumnya
                                # tanpa salin data
res = duckdb.sql("""
  SELECT kanal, count(*) n FROM df GROUP BY kanal
""").pl()                       # DuckDB membaca df
                                # Polars, hasil Polars
out = res.to_arrow()            # kembali ke Arrow
```

- DuckDB dapat meng-*query* DataFrame Polars, pandas, atau tabel Arrow yang ada di memori Python hanya dengan menyebut namanya.
- Berpindah *engine* di tengah *pipeline* menjadi murah: pakai yang paling cocok untuk tiap langkah.
- “Umumnya tanpa salin”: beberapa tipe data tetap butuh konversi. Ukur bila datanya besar.

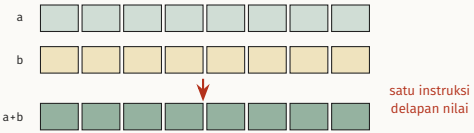
Vectorized execution: memproses kolom, bukan baris

Row-at-a-time (model *Volcano*)

Setiap operator meminta satu baris dari operator di bawahnya. Untuk satu miliar baris: satu miliar panggilan fungsi per operator, banyak percabangan.

Vectorized

Operator bertukar *batch* ribuan nilai satu kolom. Satu panggilan fungsi memproses ribuan nilai dalam satu *loop* rapat yang ramah *cache* dan dapat memakai instruksi SIMD.



Engine yang memakai pendekatan ini: DuckDB, Polars, DataFusion, ClickHouse, Photon, dan Velox.

Sumber: Boncz dkk., MonetDB/X100, CIDR 2005.

Ringkasan

Hari ini kita **mengukur** efeknya. Mengapa *cache* dan SIMD membuat ini cepat dibedah di Sistem Terdistribusi P10 (Sabtu, 5 Desember).

Polars, DuckDB, dan DataFusion

Empat *engine scale-up*

Polars (Rust, API Python)

Dataframe columnar berbasis Arrow, *multithreaded*. ***Lazy API***: seluruh *query* dioptimalkan dulu (*pushdown*, *pruning*). ***Streaming engine***: memproses data lebih besar dari RAM per potongan.

DataFusion (Rust)

Query engine SQL dan DataFrame berbasis Arrow yang dirancang untuk **disematkan** ke produk lain. Banyak sistem data baru membangun di atasnya. Tersedia `datafusion-python`.

DuckDB (C++)

Database analitik *in-process*, seperti SQLite untuk analitik. Membaca Parquet, CSV, Iceberg, dan *object storage* langsung dengan SQL.

Ballista (Rust)

Versi terdistribusi DataFusion. Menarik untuk dipelajari, tetapi ekosistem dan kematangan operasinya belum setara Spark, **masih berkembang**.

Sumber: Lamb dkk., Apache Arrow DataFusion, SIGMOD 2024; Raasveldt dan Mühleisen, DuckDB, SIGMOD 2019; dokumentasi Polars.

Polars: *eager* vs *lazy*

```python
import polars as pl

# eager: tiap baris kode langsung dikerjakan
df = pl.read_parquet("bronze/trx/*.parquet")
df = df.filter(pl.col("kanal") == "QRIS")

# lazy: susun rencana, optimalkan, baru kerjakan
q = (pl.scan_parquet("bronze/trx/*.parquet")
       .filter(pl.col("kanal") == "QRIS")
       .select("id_trx", "rek", "nominal", "waktu"))
print(q.explain())     # rencana yang sudah dioptimalkan
df = q.collect()       # baru dibaca di sini
```

- Pada versi *eager*, seluruh kolom dibaca dulu baru disaring.
- Pada versi *lazy*, *optimizer* mendorong filter ke pembacaan Parquet (*predicate pushdown*) dan hanya membaca empat kolom (*projection pushdown*).
- `explain()` menunjukkan rencana itu: biasakan membacanya, sama seperti `EXPLAIN` di SQL.

Ringkasan

Untuk *pipeline*, mulai selalu dari `scan_*` dan *lazy API*.

Polars: *expression*, bukan *loop*

```python
# lambat: fungsi Python per baris
df.with_columns(
  pl.col("nominal").map_elements(lambda x: x * 1.1))

# cepat: expression, dijalankan di Rust
df.with_columns((pl.col("nominal") * 1.1)
                  .alias("nominal_x"))

# data lebih besar dari RAM: streaming engine
(pl.scan_parquet("bronze/trx/*.parquet")
   .group_by("kanal").agg(pl.len())
   .collect(engine="streaming"))
```

API Polars bergerak cepat antar versi (misalnya cara memanggil *streaming engine*); cocokkan dengan versi di repositori kelas.

- `map_elements` memanggil Python untuk setiap baris: kehilangan *multithreading* dan *vectorization*.
- *Expression* Polars dikompilasi menjadi operasi kolom di Rust yang berjalan paralel di semua *core*.
- *Streaming engine* memproses data per potongan, sehingga *dataset* yang lebih besar dari RAM tetap bisa diolah di laptop.

DuckDB: SQL langsung atas *file* dan *object storage*

```
INSTALL httpfs; LOAD httpfs;
CREATE SECRET minio (TYPE s3, KEY_ID 'minioadmin',
  SECRET 'minioadmin', ENDPOINT 'localhost:9000',
  URL_STYLE 'path', USE_SSL false);

-- baca bronze langsung dari MinIO
CREATE VIEW bronze_trx AS
SELECT * FROM read_parquet('s3://lake/bronze/trx/*/*.parquet',
                           hive_partitioning = true);

-- tulis silver, dipartisi tanggal
COPY (SELECT ... FROM bronze_trx)
TO 's3://lake/silver/trx'
(FORMAT parquet, PARTITION_BY (tanggal),
 OVERWRITE_OR_IGNORE);
```

- Tidak ada server: DuckDB berjalan di dalam proses Python atau CLI.
- `hive_partitioning` membaca kolom *partition* dari nama folder (`tanggal=2026-11-02`).
- `COPY ... PARTITION_BY` menulis satu folder per tanggal, tata letak yang sama dengan P3.
- Membaca Iceberg dengan *extension* `iceberg` (lihat praktikum P4).

Batas DuckDB

Satu proses penulis per *file database*. Cocok untuk *job* ETL dan analisis, bukan sebagai *database* OLTP bersama.

Query yang sama, tiga *engine* satu *node*

```python
# Polars: lazy API
import polars as pl
harian = (pl.scan_parquet("data/trx/*.parquet")
  .filter(pl.col("tanggal") >= pl.date(2026,10,1))
  .join(pl.scan_parquet("data/nasabah.parquet"),
        on="id_nasabah")
  .group_by("tanggal", "cabang")
  .agg(pl.col("nominal").sum().alias("total"))
  .collect(engine="streaming"))

# DuckDB: SQL langsung atas Parquet
import duckdb
duckdb.sql("""
  SELECT tanggal, cabang, SUM(nominal) total
  FROM 'data/trx/*.parquet'
  JOIN 'data/nasabah.parquet' USING (id_nasabah)
  WHERE tanggal >= DATE '2026-10-01'
  GROUP BY ALL""").df()
```

```python
# DataFusion
from datafusion import SessionContext
ctx = SessionContext()
ctx.register_parquet("trx", "data/trx/")
ctx.register_parquet("nasabah",
                     "data/nasabah.parquet")
df = ctx.sql("""
  SELECT tanggal, cabang, SUM(nominal) total
  FROM trx JOIN nasabah USING (id_nasabah)
  WHERE tanggal >= DATE '2026-10-01'
  GROUP BY tanggal, cabang""")
df.to_pandas()
```

Ketiganya membaca Parquet langsung dan hanya kolom yang dipakai. Tidak ada *cluster*, tidak ada JVM.

Pipeline bronze → silver → gold

Bronze → silver dengan Polars *lazy*

```
import polars as pl
tgl = "2026-11-02"
bronze = pl.scan_parquet(f"bronze/trx/tanggal={tgl}/*.parquet")
silver = (bronze
  .with_columns(
     pl.col("nominal").cast(pl.Int64, strict=False),
     pl.col("waktu").str.to_datetime(),
     pl.col("kanal").str.to_uppercase())
  .filter(pl.col("nominal").is_not_null()
          & (pl.col("nominal") > 0))
  .sort("_ingest_ts", descending=True)
  .unique(subset="id_trx", keep="first")  # dedup
  .drop("nik"))             # PII tidak masuk silver
silver.sink_parquet(f"silver/trx/tanggal={tgl}/part.parquet")
```

- ***Cast***: `nominal` teks dari bronze menjadi angka; `strict=False` membuat nilai rusak menjadi *null*, lalu disaring.
- ***Deduplication***: versi terbaru per `id_trx` menang (jendela tumpang tindih dari *ingestion*).
- **PII**: NIK dibuang; rekening diganti *token* di langkah berikut.
- `sink_parquet` menulis tanpa memuat semua data ke memori.

Baris yang disaring sebaiknya ditulis ke tabel *quarantine*, bukan dibuang diam-diam (P6).

Pseudonymization rekening di silver

```
-- DuckDB: hash berkunci (salt rahasia dari env)
SET VARIABLE salt = getenv('TOKEN_SALT');
CREATE OR REPLACE VIEW silver_trx AS
SELECT id_trx,
       sha256(getvariable('salt') || rek) AS rek_token,
       nominal, kanal, waktu, tanggal
FROM read_parquet('silver_tmp/trx/*/*.parquet',
                  hive_partitioning = true);
```

- Rekening yang sama selalu menjadi *token* yang sama, sehingga *join* dan *aggregation* per rekening tetap bisa dilakukan.
- Tanpa *salt* rahasia, nomor rekening yang pendek mudah ditebak lewat *brute force*.
- Ini *pseudonymization*, bukan *anonymization*: menurut UU PDP datanya tetap data pribadi. *Tokenization* yang lebih kuat (*vault*, *key rotation*) di P14.

Sumber: Fungsi `getenv` dan variabel DuckDB bisa berbeda antar versi; cek dokumentasi versi lab.

Silver → gold dengan DuckDB SQL

```
COPY (
  SELECT t.tanggal,
         n.cabang,
         t.kanal,
         count(*)        AS jml_trx,
         sum(t.nominal)  AS total_nominal,
         count(DISTINCT t.rek_token) AS rek_aktif
  FROM silver_trx t
  JOIN read_parquet('silver/nasabah.parquet') n
    USING (rek_token)
  WHERE t.tanggal = DATE '2026-11-02'
  GROUP BY ALL
) TO 'gold/fakta_trx_harian'
(FORMAT parquet, PARTITION_BY (tanggal),
 OVERWRITE_OR_IGNORE);
```

- Tabel gold ini adalah *fact table periodic snapshot* dari P2: satu baris per tanggal, cabang, kanal.
- Gold tidak memuat PII sama sekali; aman dibaca analis dan *dashboard*.
- SQL di sini nanti mudah dipindah ke model dbt (P6) atau Spark SQL (P9–P10).

Ringkasan

Polars untuk pembersihan berbasis *expression*, DuckDB untuk *aggregation* berbasis SQL, keduanya berbagi Arrow dan Parquet.

Menulis secara *idempotent*

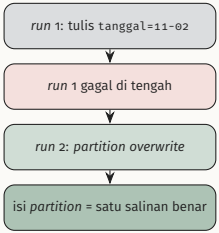
Idempotent: jalankan ulang, hasil sama

Yang salah

Job tanggal 2 November gagal di tengah, lalu dijalankan ulang dengan mode *append*. Gold kini memuat transaksi tanggal itu **dua kali**; laporan manajemen ikut salah.

Polanya

- Satu *run* = satu *partition* (misalnya satu tanggal).
- Tulis ke lokasi sementara, lalu ganti *partition* itu secara utuh (*overwrite partition*).
- Di Iceberg/Delta: *overwrite* dengan filter *partition* atau MERGE, keduanya satu *commit* atomik (P4).
- *Transformation* deterministik: tidak bergantung pada `now()` atau urutan acak.



Dengan pola ini, *backfill* (menjalankan ulang banyak tanggal) menjadi aman. P6 menjadwalkannya dengan Airflow atau Dagster.

Pemeriksaan ringan sebelum menulis

```
df = silver.collect()
cek = {
  "baris > 0":      df.height > 0,
  "id unik":        df["id_trx"].n_unique() == df.height,
  "nominal positif": (df["nominal"] > 0).all(),
  "tanpa NIK":      "nik" not in df.columns,
}
gagal = [k for k, ok in cek.items() if not ok]
if gagal:
    raise ValueError(f"silver {tgl} gagal: {gagal}")
df.write_parquet(f"silver/trx/tanggal={tgl}/part.parquet")
```

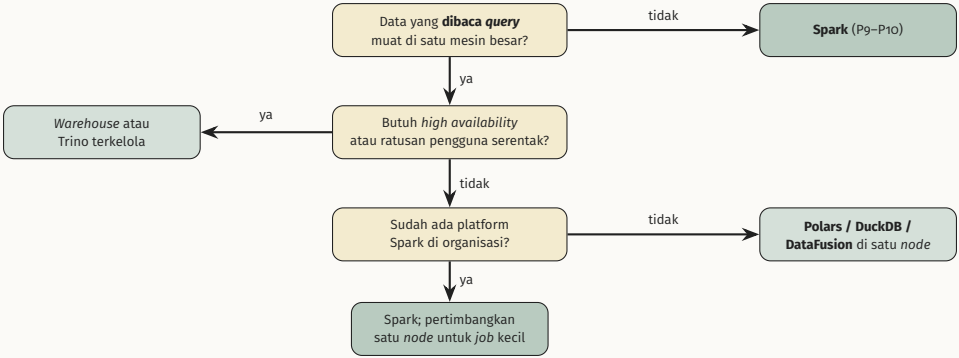
- Gagal lebih baik daripada menulis data rusak: *downstream* lebih sulit dibersihkan daripada *upstream*.
- Ini versi sederhana dari gerbang P4. Di P6 pemeriksaan seperti ini ditulis sebagai *data quality test* di dbt, Great Expectations, atau Soda.

When to use dan *best practice*

When to use: pandas, Polars, DuckDB, DataFusion

Engine	Pakai bila	Pertimbangkan alternatif bila
pandas	Data kecil, eksplorasi, ekosistem *library* yang hanya menerima pandas; pandas 2 bisa memakai tipe berbasis Arrow	Data puluhan juta baris ke atas, butuh semua *core*
Polars	*Transformation dataframe* besar di satu mesin, *pipeline* Python yang cepat dan hemat memori	Tim lebih nyaman SQL; API berubah cepat antar versi
DuckDB	SQL analitik atas Parquet, CSV, Iceberg, atau *object storage* tanpa server; *aggregation* dan *join*	Banyak penulis serentak, beban OLTP
DataFusion	Membangun *engine* atau produk data sendiri di atas Arrow (Rust atau Python)	Hanya butuh alat analisis siap pakai (DuckDB atau Polars lebih matang)
Spark	Data melampaui satu mesin besar atau butuh *cluster* bersama	Dibahas P9–P10

Satu mesin atau *cluster*: alur keputusan



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

Best practice industri untuk ETL satu mesin

1. ***Start simple.*** Satu mesin dulu; pindah ke *cluster* hanya bila angka menuntutnya.
2. ***Lazy* dan *scan***: biarkan *optimizer* memangkas kolom dan baris sejak awal.
3. ***Expression* atau SQL, bukan UDF Python** per baris.
4. **Parquet di antara tahap**, bukan CSV.
5. **Satu *run* = satu *partition***, tulis dengan *overwrite partition*.
6. **PII keluar sebelum silver**: buang atau *tokenize*.
7. **Simpan *state*** (*high-water mark*) setelah tulis berhasil.
8. **Gagal dengan keras** bila pemeriksaan tidak lolos.
9. ***Pin* versi** *library* di `requirements` atau `pyproject`.
10. **Catat ukuran input, waktu, memori** tiap *run*.

Kapan perlu *cluster*: menuju P9–P10

Tanda-tanda satu mesin tidak cukup

- Data yang **harus** dibaca satu *job* melebihi disk dan RAM satu mesin besar, bahkan dengan *streaming engine*
- Waktu proses melewati jendela *batch* (misalnya harus selesai sebelum pukul 06.00)
- Banyak tim menjalankan *job* bersamaan di platform bersama
- Butuh toleransi gagal: satu mesin mati tidak boleh menghentikan semuanya

Yang datang setelah UTS

P9: ekosistem Hadoop dan Spark dari dalam (RDD, DAG, *lineage*) dalam Scala dan PySpark. P10: Spark skala besar, *tuning*, dan mengukur *scaling*, dengan ***baseline* Polars/DuckDB dari hari ini** sebagai pembanding.

Ringkasan

Simpan hasil *benchmark* praktikum hari ini. Angka itu menjadi *baseline* tugas besar UAS.

Praktikum

Praktikum: lingkungan dan data

```
# pyproject / requirements (contoh)
# polars, duckdb, datafusion, pyarrow,
# pandas, numpy, faker

# bangkitkan transaksi sintetis
python gen_trx.py --baris 10_000_000 \
  --hari 30 --out bronze/trx/
python gen_nasabah.py --baris 100_000 \
  --out bronze/nasabah/
```

Skrip generator dan versi *library* ada di repositori kelas.

1. Bangkitkan bronze: mulai 1 juta baris, naikkan ke 10 juta dan 50 juta bila laptop sanggup.
2. Sisipkan data kotor dengan sengaja: duplikat `id_trx`, `nominal` berupa teks rusak, waktu yang salah format.
3. Simpan per tanggal di MinIO atau disk lokal (`tanggal=YYYY-MM-DD`).

Data sintetis

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

Praktikum: *pipeline* dan eksperimen yang diukur

1. Bangun bronze → silver dengan Polars *lazy* (*cast*, *deduplication*, buang NIK, *pseudonymization* rekening).
2. Bangun silver → gold dengan DuckDB SQL (`gold.fakta_trx_harian`).
3. Jalankan ulang satu tanggal dua kali; buktikan jumlah baris gold tidak berubah.
4. Jalankan *aggregation* gold yang sama dengan pandas, Polars, DuckDB, dan DataFusion.

Engine	**1 juta baris**	**10 juta**	**50 juta**	**Memori puncak**
pandas	…	…	…	…
Polars (*lazy*)	…	…	…	…
DuckDB	…	…	…	…
DataFusion	…	…	…	…

Isi dengan median dari minimal tiga kali jalan. Memori puncak bisa diukur dengan `/usr/bin/time -v` (Linux). Jelaskan **mengapa** urutannya begitu.

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum: kode *pipeline*, bukti *idempotent*, tabel *benchmark*, dan *output* `explain()` Polars yang dianotasi (di mana *pushdown* terjadi).
2. Jawab dengan angka: sampai ukuran berapa laptop Anda masih nyaman? Apa yang habis duluan: RAM, CPU, atau disk?
3. Lanjutkan rancangan tugas besar UTS: sumber data, cara *ingestion*, dan kode bronze $\rightarrow$ silver pertama.
4. Baca dokumentasi Airflow (konsep DAG) dan dbt (*models*, *tests*).

Pertemuan depan

Orchestration, dbt, dan *data quality*: menjadwalkan *pipeline* hari ini dengan Airflow atau Dagster, menulis ulang *transformation*-nya sebagai model dbt, dan memasang *data quality test* yang bisa menghentikan *pipeline*.

Terima kasih

Pertanyaan?