

Pertemuan 6 — *Orchestration*, dbt, dan *Data quality*

Pipeline bronze → silver → gold yang berjalan sendiri, aman diulang, dan berhenti ketika datanya rusak

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. *Ingestion: full vs incremental*
2. ETL vs ELT
3. *Orchestration: Airflow*
4. Dagster dan cara berpikir berbasis *asset*
5. *Idempotency, backfill*, dan partisi waktu
6. Transformasi dengan dbt
7. *Data quality*: dimensi dan pengujiannya
8. DataOps: Git, CI, lingkungan terpisah
9. Praktikum: Airflow + dbt + *data quality test* + *backfill*
10. Rekap P1–P5 dan arahan tugas besar UTS

Sasaran pertemuan ini

Anda dapat membangun *pipeline* harian yang terjadwal, dapat dijalankan ulang tanpa menggandakan data, dapat mengisi ulang hari yang terlewat, dan **menolak** meneruskan data yang gagal *data quality test*.

Ingestion: full vs incremental

Tiga cara mengambil data dari sumber

Cara	Bagaimana	Kapan cocok
Full load	Salin seluruh tabel setiap kali jalan	Tabel kecil (daftar cabang, produk akad); paling sederhana dan paling mudah diperiksa
Incremental	Ambil baris yang berubah sejak terakhir, berdasarkan kolom <code>updated_at</code> atau ID naik	Tabel besar yang barisnya punya penanda waktu andal
Log-based CDC	Baca log transaksi basis data (WAL)	Butuh setiap perubahan, termasuk <i>delete</i> ; dibahas di P11

Jebakan *incremental*

`updated_at` yang tidak diisi oleh semua aplikasi; baris yang dihapus tidak pernah terlihat; transaksi yang *commit* terlambat lolos dari jendela waktu.

Praktik umum

Ambil dengan jendela yang sedikit tumpang tindih (misalnya mundur satu jam), lalu hapus duplikat dengan *key* unik di tahap berikutnya.

- ▶ **Skrip sendiri** (Python, psycopg, requests): kendali penuh, tetapi Anda yang menanggung *retry*, paginasi, perubahan skema.
- ▶ **Konektor terkelola** atau terbuka (Airbyte, Fivetran, dlt): ratusan konektor siap pakai untuk basis data, SaaS, dan API.
- ▶ **Debezium dan Kafka Connect**: CDC dan aliran berkelanjutan (P11).

Intinya

Tulis sendiri hanya bila sumbernya unik. Untuk sumber umum, konektor yang sudah teruji lebih murah daripada waktu engineer Anda.

Ke mana data mendarat

Selalu ke *layer* **bronze** lebih dulu (P4): apa adanya, ditambah kolom metadata.

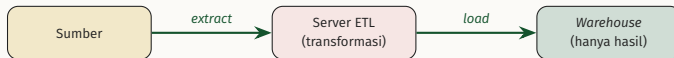
- ▶ `_ingested_at`: kapan diambil
- ▶ `_source`: dari sistem mana
- ▶ `_batch_id`: dari eksekusi yang mana

Metadata ini yang membuat *backfill* dan audit mungkin dilakukan.

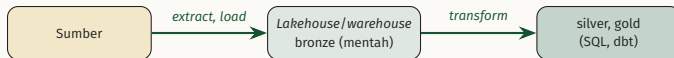
ETL vs ELT

Di mana transformasi terjadi

ETL



ELT



ETL

Lahir ketika *storage warehouse* mahal: hanya data bersih yang disimpan. Data mentah hilang; salah transformasi berarti ambil ulang dari sumber.

ELT

Storage murah dan *compute* terpisah: simpan mentah, transformasi di dalam platform dengan SQL. Mentah tetap ada, transformasi bisa diulang kapan saja.

Tetapi: data pribadi yang tidak boleh disimpan mentah (misalnya nomor kartu) tetap harus di-*mask* atau ditokenisasi **sebelum** *load* — ETL kecil di depan ELT (P14).

Orchestration: Airflow

Mengapa butuh *orchestrator*

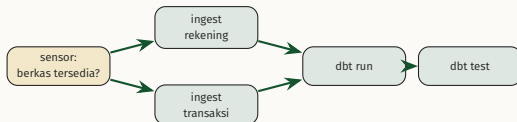
cron + skrip, bulan ketiga

- ▶ Skrip B jalan walau skrip A gagal
- ▶ Tidak ada yang tahu kemarin gagal sampai rapat pagi
- ▶ Mengisi ulang seminggu berarti menjalankan tangan tujuh kali
- ▶ Logika *retry* disalin di tiap skrip

Yang diberikan *orchestrator*

- ▶ **Dependensi:** B hanya jalan bila A sukses
- ▶ **Scheduling** dan **catch-up/backfill** per tanggal
- ▶ **Retry**, *timeout*, *alert*
- ▶ **Riwayat:** log dan status setiap eksekusi
- ▶ **Parameter** tanggal yang konsisten untuk tiap eksekusi

Konsep inti Airflow



- **DAG**: graf tugas tanpa siklus, ditulis sebagai kode Python.
- **Task** dibuat dari **operator** (Bash, Python, SQL, Spark, dbt, Kubernetes).
- **Sensor**: task yang menunggu kondisi (berkas datang, partisi ada).
- **Scheduler** membuat **DAG run** per interval data; tiap *run* membawa tanggal logisnya sendiri.
- **Executor** menjalankan task (lokal, Celery, Kubernetes).

Airflow mengatur, bukan mengolah

Jangan memproses jutaan baris di dalam *worker* Airflow. Suruh dbt, Polars, DuckDB, basis data, atau nanti Spark (P9–P10) mengerjakannya; Airflow cukup memicu dan memantau.

DAG harian dengan TaskFlow API

```
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
import pendulum

@dag(schedule="@daily", catchup=True,
      start_date=pendulum.datetime(2026, 11, 1,
                                     tz="Asia/Jakarta"),
      default_args={"retries": 2})
def trx_harian():

    @task
    def ingest(ds=None): # ds: tanggal logis
        # tulis TIMPA partisi tgl=ds
        salin_ke_bronze(tanggal=ds)

    dbt_run = BashOperator(task_id="dbt_run",
                           bash_command="dbt run --vars '{tgl: {{ ds }}}'")
    dbt_test = BashOperator(task_id="dbt_test",
                            bash_command="dbt test")

    ingest() >> dbt_run >> dbt_test

trx_harian()
```

- ▶ Semua langkah memakai ds, bukan “hari ini”: eksekusi untuk 3 November selalu mengolah data 3 November, kapan pun ia dijalankan.
- ▶ catchup=True: hari yang terlewat sejak start_date ikut dijalankan.
- ▶ Bila dbt test gagal, DAG berstatus gagal dan alert terkirim; lapis gold tidak diperbarui.
- ▶ Nama modul impor sedikit berbeda antar versi Airflow; ikuti dokumentasi versi yang dipasang.

Dagster dan cara berpikir berbasis *asset*

Airflow: berpikir dalam task

“Jalankan langkah A, lalu B, lalu C.” *Orchestrator* tahu **langkah** mana yang sukses, tetapi tidak tahu **tabel** apa yang dihasilkan.

Dagster: berpikir dalam asset

“Tabel `silver.transaksi` dibangun dari `bronze.transaksi`.” Dependensi ditulis antar data. *Orchestrator* tahu tabel mana yang basi, partisi mana yang belum ada, dan *lineage*-nya.

```
from dagster import (asset,
    DailyPartitionsDefinition)

harian = DailyPartitionsDefinition(
    start_date="2026-11-01")

@asset(partitions_def=harian)
def bronze_transaksi(context):
    salin_ke_bronze(context.partition_key)

@asset(partitions_def=harian,
    deps=[bronze_transaksi])
def silver_transaksi(context):
    bersihkan(context.partition_key)
```

Fungsi *asset* adalah fungsi Python biasa — mudah diuji dengan *unit test* tanpa menyalakan server.

When to use: orchestrator

Alat	Pakai bila	Pertimbangkan alternatif bila
Airflow	Standar industri, operator untuk hampir semua sistem, tersedia sebagai layanan terkelola di <i>cloud</i> besar, banyak engineer sudah mengenalnya	Ingin <i>lineage</i> dan status data bawaan; tim kecil yang ingin mudah menguji lokal
Dagster	Platform data baru, berpikir per tabel/partisi, butuh pengujian lokal dan <i>lineage</i> bawaan	Organisasi sudah punya ratusan DAG Airflow; butuh ekosistem operator terluas
Prefect, Argo	Alur Python dinamis (Prefect); alur berbasis kontainer di Kubernetes (Argo Workflows)	Butuh fitur khas data seperti <i>backfill</i> per partisi
cron	Satu tugas kecil tanpa dependensi	Lebih dari dua langkah yang saling bergantung

Best practice industri

Orchestrator memicu, sistem lain mengolah. Simpan DAG di Git, uji di lingkungan *staging*, dan jangan pernah menaruh kata sandi di kode DAG — pakai *connection* atau *secret backend*.

Idempotency, backfill, dan partisi waktu

Idempotency: jalankan dua kali, hasilnya sama

Tidak *idempotent*

```
INSERT INTO gold.harian SELECT ... WHERE tgl =  
'2026-11-16'
```

Dijalankan ulang karena *retry* $\Rightarrow$ total transaksi 16 November menjadi **dua kali lipat**. Laporan ke manajemen salah tanpa ada *error*.

Idempotent

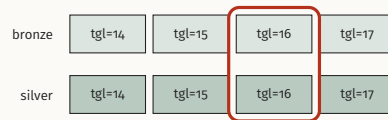
Timpa partisi: hapus dan tulis ulang partisi `tgl=2026-11-16` (INSERT OVERWRITE, atau DELETE lalu INSERT dalam satu transaksi).

MERGE dengan *key unik*: baris yang sudah ada diperbarui, bukan ditambah.

Intinya

Di sistem terdistribusi, *retry* pasti terjadi. *Pipeline* yang tidak *idempotent* hanya menunggu waktu untuk menghasilkan angka ganda.

1. **Tugas murni:** keluaran hanya bergantung pada masukan dan parameter tanggal, bukan pada jam saat dijalankan.
2. **Partisi tak berubah:** tiap eksekusi menulis satu partisi utuh; partisi lama tidak ditambah sedikit-sedikit.
3. **Timpa, jangan tambah:** menjalankan ulang berarti menulis ulang partisi.
4. **Simpan mentah:** bronze tidak pernah dihapus selama masa retensi, sehingga semua lapis di atasnya bisa dibangun ulang.



jalankan ulang 16 Nov:
hanya kolom ini ditulis ulang

Sumber: Maxime Beauchemin (2018), *Functional Data Engineering: a modern paradigm for batch data processing*.

Backfill: mengisi ulang masa lalu

Kapan dibutuhkan:

- ▶ Sumber terlambat mengirim data tiga hari
- ▶ Rumus margin murabahah di lapis gold dikoreksi; enam bulan harus dihitung ulang
- ▶ Kolom baru ditambahkan dan perlu diisi untuk data lama

Prasyaratnya: tugas *idempotent* dan berparameter tanggal. Tanpa itu, *backfill* menjadi operasi manual yang menakutkan.

Perintah (contoh)

Airflow 2.x:

```
airflow dags backfill -s 2026-11-01 -e  
2026-11-07 trx_harian
```

Airflow 3.x:

```
airflow backfill create -dag-id trx_harian  
-from-date 2026-11-01 -to-date 2026-11-07
```

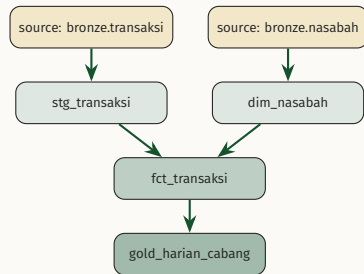
Dagster: pilih rentang partisi di UI, lalu *materialize*.

Sintaks berubah antar versi; periksa dokumentasi versi yang dipasang. Batasi paralelisme *backfill* agar tidak membanjiri basis data sumber.

Transformasi dengan dbt

dbt: SQL yang diperlakukan seperti perangkat lunak

- ▶ Satu **model** = satu berkas .sql berisi SELECT. dbt yang membuat tabel atau *view*-nya.
- ▶ `{{ ref('model_lain') }}` membangun graf dependensi otomatis, urutan eksekusi, dan *lineage*.
- ▶ `{{ source('bronze', 'transaksi') }}` menandai data mentah dari luar dbt.
- ▶ **Materialization**: view, table, incremental, ephemeral.
- ▶ **Tests**, **docs**, dan **snapshot** dalam satu proyek Git.
- ▶ Adaptor untuk banyak mesin: Postgres, DuckDB, Spark, Trino, BigQuery, Snowflake.



graf yang dibangun dbt dari `ref()` dan `source()`

Model *incremental* dan *snapshot* SCD tipe 2

```
-- models/silver/fct_transaksi.sql
{{ config(materialized='incremental',
          unique_key='trx_id') }}

select trx_id, rekening_id, cabang_id,
       nominal, waktu_trx
from {{ ref('stg_transaksi') }}
{% if is_incremental() %}
where waktu_trx >=
      (select max(waktu_trx) - interval '1 hour'
       from {{ this }})
{% endif %}
```

Jendela mundur satu jam + unique_key = baris terlambat tertangkap, duplikat tidak terjadi.

```
-- snapshots/nasabah_snapshot.sql
{% snapshot nasabah_snapshot %}
{{ config(target_schema='snapshots',
          unique_key='nasabah_id',
          strategy='timestamp',
          updated_at='updated_at') }}
select * from {{ source('bronze', 'nasabah') }}
{% endsnapshot %}
```

dbt menambahkan dbt_valid_from dan dbt_valid_to: riwayat segmen dan alamat nasabah terjaga — SCD tipe 2 dari pertemuan 2 tanpa ditulis tangan.

Bronze → silver → gold di satu mesin: dbt-duckdb atau Polars

Layer	Yang dikerjakan	Alat (satu mesin)	Uji minimal
bronze	<i>Ingest</i> apa adanya, <i>append-only</i> , partisi tgl, kolom metadata	Python/dlt → Parquet di MinIO	jumlah baris > 0, <i>freshness</i>
silver	<i>Cast</i> tipe, <i>dedup</i> , standarisasi kode akad, <i>tokenization</i> PII	dbt-duckdb (stg_*, fct_*) atau Polars <i>lazy</i>	unique, not_null, accepted_values
gold	<i>Star schema</i> dan agregat siap <i>dash-board</i>	dbt <i>table / incremental</i>	rekonsiliasi total vs silver

```
# silver dengan Polars (lazy)
import polars as pl
src = "s3://bronze/transaksi/tgl=2026-11-09/*"
dst = "silver/transaksi/2026-11-09.parquet"
(pl.scan_parquet(src)
 .unique(subset="trx_id")
 .with_columns(pl.col("nominal")
               .cast(pl.Decimal(18, 2)))
 .sink_parquet(dst))
```

- ▶ Pilih **satu** jalur per kelompok: dbt-duckdb (SQL) atau Polars (Python).
- ▶ Airflow/Dagster hanya memicu urutan bronze → silver → gold dan menghentikannya bila uji gagal.
- ▶ Data puluhan GB masih nyaman di satu mesin (P5); klaster Spark baru di P9-P10.

Data quality

Enam dimensi *data quality*

Dimensi	Pertanyaan	Contoh uji di bank
Completeness	Ada yang hilang?	rekening_id tidak boleh kosong; jumlah cabang yang mengirim data = jumlah cabang aktif
Validity	Sesuai aturan format dan domain?	jenis_akad hanya murabahah, musyarakah, mudharabah, ijarah, ...
Uniqueness	Ada yang ganda?	trx_id unik
Timeliness	Sampai tepat waktu?	Partisi hari kemarin tersedia sebelum 06.00
Consistency	Cocok antar sistem?	Total mutasi = selisih saldo awal dan akhir
Accuracy	Sesuai kenyataan?	Rekonsiliasi dengan laporan keuangan resmi

Sumber: Dimensi mengikuti DAMA-DMBOK; penamaan dan jumlah dimensi berbeda sedikit antar rujukan.

Menguji data quality: dbt tests

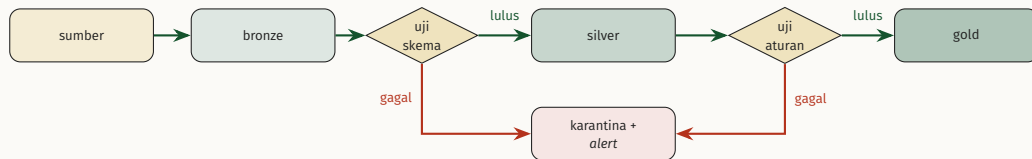
```
# models/silver/schema.yml
models:
  - name: fct_transaksi
    columns:
      - name: trx_id
        tests: [unique, not_null]
      - name: jenis_akad
        tests:
          - accepted_values:
              values: [murabahah, musyarakah,
                      mudharabah, ijarah, wadiah]
      - name: rekening_id
        tests:
          - relationships:
              to: ref('dim_rekening')
              field: rekening_id
```

- ▶ Empat uji generik bawaan: unique, not_null, accepted_values, relationships.
- ▶ Uji khusus ditulis sebagai SELECT yang mengembalikan baris yang **melanggar**; nol baris berarti lulus.
- ▶ severity: warn untuk masalah yang perlu diketahui tetapi tidak boleh menghentikan laporan.

Great Expectations dan Soda

Data quality test di luar dbt: profil distribusi, rentang nilai, perubahan volume, dan laporan HTML. Soda memakai bahasa pemeriksaan deklaratif (SodaCL).

Di mana uji ditempatkan



Prinsip *circuit breaker*

Lebih baik *dashboard* menampilkan data kemarin dengan peringatan “data hari ini tertunda” daripada menampilkan angka salah dengan percaya diri.

Uji di *upstream*

Semakin dekat ke sumber, semakin murah perbaikannya. Pada P15 uji ini menjadi *data contract* antara tim aplikasi dan tim data.

DataOps

DataOps: kebiasaan rekayasa perangkat lunak untuk data

1. **Semua di Git:** DAG, model dbt, konfigurasi, docker-compose.
2. **Review:** perubahan model lewat *pull request*.
3. **CI:** setiap *pull request* menjalankan `dbt build` pada data contoh dan uji unit fungsi Python.
4. **Lingkungan terpisah:** dev, staging, prod dengan skema atau katalog berbeda (`dbt -target`).
5. **Rilis kecil dan sering,** dengan kemampuan *rollback*.

Hanya bangun yang berubah

`dbt build --select state:modified+` membangun model yang diubah beserta turunannya saja, dengan membandingkan terhadap *manifest* produksi. CI tetap cepat meski proyek membesar.

Data produksi di laptop?

Tidak. Pakai data sintetis atau sampel yang sudah *di-mask*. Ini kewajiban, bukan anjuran (P14).

Praktikum

```
services:
  minio:
    image: minio/minio
    command: server /data --console-address :9001
    ports: ["9000:9000", "9001:9001"]
  postgres:
    # sumber + metadata Airflow
    image: postgres:16
    environment: {POSTGRES_PASSWORD: lab}
  airflow:
    image: apache/airflow
    command: standalone
    ports: ["8080:8080"]
    volumes:
      - ./dags:/opt/airflow/dags
      - ./dbt:/opt/airflow/dbt
```

- ▶ Mode standalone hanya untuk lab: satu container berisi *scheduler*, *webserver*, dan basis data metadata.
- ▶ dbt memakai adaptor dbt-duckdb yang membaca Parquet di MinIO, atau dbt-postgres.
- ▶ Versi *image* di atas contoh; pakai berkas resmi yang dibagikan di kelas.
- ▶ Lab ini sekaligus kerangka tugas besar UTS.

Praktikum: lima langkah yang diperiksa

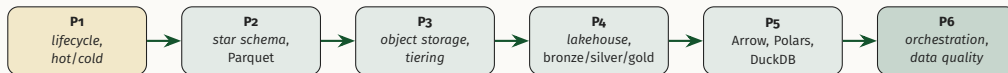
1. DAG harian `trx_harian`: *ingest* transaksi sintetis ke bronze dengan partisi `tgl`, lalu `dbt run` dan `dbt test`.
2. Model `dbt stg_transaksi`, `dim_nasabah` (dari *snapshot*), `fct_transaksi` (*incremental*), dan `gold_harian_cabang`.
3. Minimal enam *data quality test*, satu per dimensi yang bisa diuji otomatis.
4. **Sabotase**: sisipkan 50 baris dengan `trx_id` ganda dan satu `jenis_akad` tidak valid. Tunjukkan DAG berhenti dan lapis gold tidak berubah.
5. **Backfill**: jalankan 1–3 November dua kali. Buktikan dengan kueri bahwa jumlah baris dan total nominal per hari identik setelah eksekusi pertama dan kedua.

Intinya

Bukti yang dikumpulkan: tangkapan layar graf DAG, log uji yang gagal, dan hasil kueri sebelum–sesudah *backfill*.

Rekap P1–P5 dan arahan tugas besar UTS

Enam pertemuan dalam satu *pipeline* ETL



Pertanyaan yang harus bisa Anda jawab

- ▶ Mengapa tabel ini memakai *grain* per transaksi, bukan per hari?
- ▶ Mengapa Parquet, dan dengan partisi apa?
- ▶ Data ini di *tier* mana setelah 90 hari, dan berapa biayanya?
- ▶ Mengapa cukup Polars/DuckDB di satu mesin, dan kapan perlu Spark?
- ▶ Apa yang terjadi bila DAG ini dijalankan dua kali?

UTS: “*Pipeline* ETL bronze-silver-gold”

Spesifikasi dan rubrik (dek Po8) dibagikan **hari ini**; rancangan kelompok dinilai hari ini juga. Demo dan presentasi: Senin 23 November (P8). Bobot 35%, mengukur CPMK-1 sampai CPMK-3. Satu mesin; Spark **tidak** wajib. *Orchestration* dan *data quality test* dari lab hari ini wajib ada.

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum: graf DAG, daftar model dbt, hasil *data quality test* termasuk yang gagal saat sabotase, dan bukti *backfill idempotent*.
2. Baca Reis dan Housley Bab 7–8 dan dokumentasi dbt tentang *incremental models* dan *snapshots*.
3. Tugas besar UTS: finalkan rancangan (diagram bronze-silver-gold, dataset, *schema* gold), integrasikan DAG dan *data quality test* ke repositori kelompok.
4. Baca dek Po8 (spesifikasi dan rubrik UTS) dan siapkan pertanyaan untuk P7.

Pertemuan depan

Storage engine, NoSQL, dan *serving store* untuk *layer* gold, plus *immutable storage* dan *ledger* (immudb, Hyperledger Fabric).

Terima kasih

Pertanyaan?