Pertemuan 4 — *Lakehouse* dan *Medallion Architecture*

Dari *warehouse* dan *lake* ke *lakehouse*: ACID di atas *object storage*, lalu bronze, silver, gold

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Masalah: tumpukan *file* Parquet bukan tabel
2. *Data warehouse*: Inmon, Kimball, MPP, *cloud*
3. *Data lake* dan bahaya *data swamp*
4. *Lakehouse*: satu salinan data untuk semua
5. Cara kerja *table format*: metadata, *snapshot*, ACID
6. *Time travel*, *schema* dan *partition evolution*
7. MERGE, *compaction*, dan perawatan tabel
8. Katalog tabel dan *query engine*
9. *Medallion architecture*: aturan bronze, silver, gold
10. Contoh tabel bank syariah per *layer*, *when to use*, praktikum

Sasaran pertemuan ini

Anda dapat menjelaskan bagaimana Iceberg atau Delta mengubah *file* Parquet di *object storage* menjadi tabel yang aman ditulis bersamaan, dapat diputar ke masa lalu, dan dapat diubah skemanya, lalu merancang *layer* bronze, silver, gold beserta aturan dan gerbang *data quality* di antaranya. Inilah kerangka tugas besar UTS.

Masalah: *file* bukan tabel

Yang terjadi bila “tabel” hanyalah folder berisi Parquet

Sejak P2–P3 kita menyimpan transaksi sebagai Parquet di MinIO, dipartisi per tanggal. Sekarang bayangkan dua *job* berjalan bersamaan:

- *Job* A menimpa *partition* `tanggal=2026-11-01` dengan data koreksi.
- *Job* B (*dashboard*) sedang membaca *partition* yang sama.

B bisa melihat **separuh *file* lama dan separuh *file* baru**. Bila A gagal di tengah jalan, *partition* itu tinggal setengah.

Masalah folder Parquet polos

- Tidak atomik: penulisan bisa terlihat sebagian
- Tidak ada isolasi antara pembaca dan penulis
- Tidak ada UPDATE atau DELETE per baris (koreksi transaksi, hak hapus UU PDP)
- *Schema* tidak dijaga: satu *job* menulis `nominal` sebagai teks, semua pembaca rusak
- *Listing* jutaan objek lambat
- Tidak bisa kembali ke versi kemarin

Ringkasan

Data warehouse sudah lama menyelesaikan semua masalah ini, dengan harga mahal dan data terkunci di dalamnya. *Lakehouse* mencoba mendapatkan keduanya.

Data warehouse

Data warehouse: definisi klasik

Inmon: empat sifat

- ***Subject-oriented***: disusun menurut subjek bisnis (nasabah, pembiayaan), bukan menurut aplikasi
- ***Integrated***: data dari banyak sistem diseragamkan (kode cabang, mata uang)
- ***Time-variant***: menyimpan sejarah
- ***Non-volatile***: dimuat dan dibaca, tidak diubah seperti OLTP

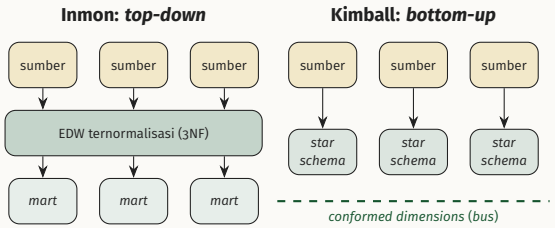
Tujuannya satu: **satu sumber kebenaran untuk analitik**, terpisah dari sistem transaksi agar *query* berat tidak mengganggu nasabah.

Di bank syariah digital

Database inti perbankan, sistem pembiayaan, aplikasi *mobile*, dan CRM disatukan di *warehouse*. Laporan kolektibilitas dan *dashboard* manajemen membaca dari sana, bukan dari *database* produksi.

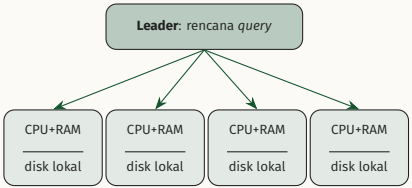
Sumber: Inmon, *Building the Data Warehouse*; Kimball dan Ross, *The Data Warehouse Toolkit*.

Inmon vs Kimball: dua jalan membangun *warehouse*



- **Inmon**: bangun dulu gudang perusahaan yang ternormalisasi dan terintegrasi; *data mart* per divisi diturunkan darinya. Konsisten, tetapi lama sebelum ada hasil.
- **Kimball**: bangun *star schema* per proses bisnis (P2), satukan lewat ***conformed dimensions*** (dimensi nasabah, waktu, cabang yang sama di semua fakta). Cepat memberi nilai.
- Di praktik modern, keduanya bercampur: *layer* bersih yang terintegrasi (gaya Inmon) lalu *layer* siap pakai berbentuk *star schema* (gaya Kimball), persis *silver* dan *gold* nanti.

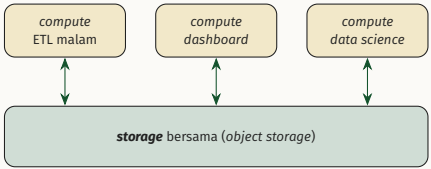
MPP: *warehouse* yang terdistribusi



shared-nothing: tiap *node* memegang sepotong tabel

- **MPP** (*Massively Parallel Processing*): tabel dipotong ke banyak *node*; tiap *node* punya CPU, memori, dan disk sendiri.
- *Query* dipecah, dijalankan paralel, hasilnya digabung, gagasan yang sama dengan Spark, tetapi di dalam satu produk *database*.
- Kelemahan klasik: *compute* dan *storage* terikat. Butuh ruang simpan lebih = beli *node* lengkap; ingin *query* lebih cepat sesaat = tetap membayar disk.

Cloud data warehouse: memisahkan *compute* dan *storage*



Tiap beban kerja punya *compute* sendiri yang bisa dinyalakan, dimatikan, dan diperbesar terpisah, membaca data yang sama.

Produk	Ciri
BigQuery	*Serverless*; ditagih per data yang dipindai atau per kapasitas
Snowflake	*Virtual warehouse* terpisah di atas *storage* bersama
Redshift	Berawal dari MPP; kini punya *storage* terkelola terpisah

Cost-nya

Data tersimpan dalam format milik vendor. Membaca dengan mesin lain berarti mengekspor dulu. Model harga berbeda-beda, selalu cek harga terkini.

Data lake dan *data swamp*

Data lake: simpan semuanya, putuskan nanti

Gagasan

- Semua data mentah (tabel, JSON, log, citra, audio) disimpan murah di HDFS atau *object storage*
- ***Schema-on-read***: *schema* diterapkan saat membaca, bukan saat menulis
- Format terbuka: siapa pun dapat membaca dengan mesin apa pun

Kelebihan

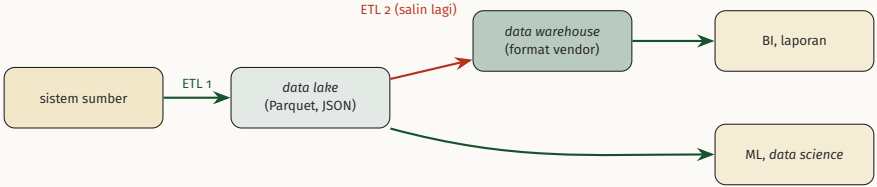
Murah, fleksibel, cocok untuk *data science* dan *machine learning* yang butuh data mentah dan tak terstruktur.

Data swamp: gejala danau yang membusuk

- Tidak ada yang tahu isi folder `/raw/tmp_final_v3/`
- Lima salinan "tabel nasabah" dengan isi berbeda
- Tidak jelas pemilik, asal-usul, dan *data quality*
- Data pribadi tercecer tanpa kendali akses
- Pembaca rusak karena *schema* berubah diam-diam

Obatnya bukan teknologi saja: katalog, kepemilikan, dan kontrak data (P15).

Arsitektur dua tingkat: *lake + warehouse*



Masalahnya

- Data disalin dua kali; dua *pipeline* untuk dirawat
- *Warehouse* selalu tertinggal dari *lake*
- BI dan ML melihat angka yang berbeda
- Membayar *storage* dua kali

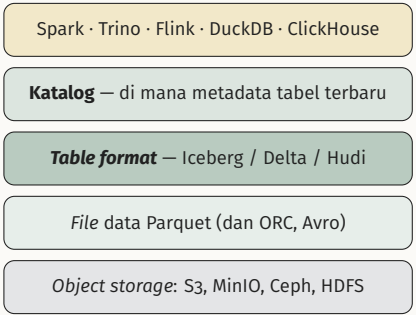
Pertanyaan *lakehouse*

Bisakah *lake* sendiri punya ACID, *schema* yang dijaga, kinerja *query* yang baik, dan kendali akses, sehingga *warehouse* terpisah tidak diperlukan untuk sebagian besar kebutuhan?

Sumber: Armbrust dkk., *Lakehouse*, CIDR 2021.

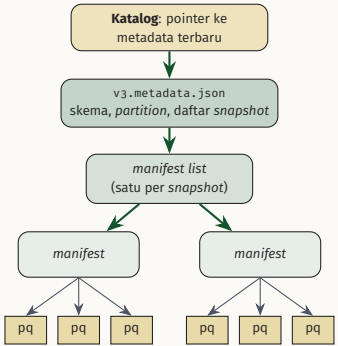
Lakehouse dan *table format*

Lakehouse: *layer* metadata di atas *file* terbuka



- ***Table format***: spesifikasi terbuka yang mencatat ***file*** **mana yang menyusun tabel pada saat tertentu**, beserta *schema* dan *partition*-nya.
- Data tetap Parquet biasa; yang ditambahkan hanyalah ***file*** **metadata** di sebelahnya.
- Banyak mesin bisa membaca dan menulis tabel yang sama selama mematuhi spesifikasinya, *compute* dan *storage* benar-benar terpisah.
- Tiga format utama lahir di industri: **Iceberg** (Netflix), **Delta Lake** (Databricks), **Hudi** (Uber).

Struktur metadata Iceberg



Sumber: Spesifikasi Apache Iceberg (*table spec*).

- ***Manifest***: daftar *file* data beserta statistik per kolom (min, maks, jumlah *null*). *Query engine* melewati *file* yang pasti tidak cocok **tanpa membukanya** dan tanpa *listing* folder.
- ***Manifest list***: daftar *manifest* yang menyusun satu *snapshot*.
- ***Snapshot***: keadaan tabel pada satu waktu. Setiap penulisan membuat *snapshot* baru; *file* lama tidak diubah.
- **Katalog**: satu-satunya tempat yang bisa berubah, pointer ke `metadata.json` terbaru.

ACID di atas *object storage*: *commit* dengan menukar pointer

1. Penulis menulis *file* Parquet baru (belum terlihat siapa pun).
2. Penulis menulis *manifest* dan `metadata.json` baru.
3. Penulis meminta katalog menukar pointer dari `v3` ke `v4` **hanya bila pointer masih `v3`** (*compare-and-swap*).
4. Berhasil: semua pembaca baru melihat `v4` sekaligus. Gagal karena ada penulis lain: baca `v4` milik orang lain, cek konflik, coba lagi.

Ini ***optimistic concurrency control***: tidak mengunci di depan, memeriksa konflik saat *commit*.

Hasilnya: ACID

- ***Atomicity***: *commit* adalah satu pertukaran pointer
- ***Consistency***: *schema* diperiksa sebelum *commit*
- ***Isolation***: pembaca memegang satu *snapshot* sampai selesai (*snapshot isolation*)
- ***Durability***: *file* di *object storage* yang direplikasi (P3)

Delta Lake melakukan hal serupa

Tiap *commit* adalah *file* JSON bernomor urut di folder `_delta_log/`, sesekali diringkas menjadi *checkpoint* Parquet.

Time travel dan *evolution*

Time travel dan *rollback*

```sql
-- Spark SQL + Iceberg (Spark dipakai di P9--P10)
SELECT * FROM lake.silver.trx
  VERSION AS OF 4820199935123456789;

SELECT cabang, SUM(nominal)
FROM lake.silver.trx
  TIMESTAMP AS OF '2026-11-01 06:00:00'
GROUP BY cabang;

-- kembalikan tabel ke snapshot sebelum job rusak
CALL lake.system.rollback_to_snapshot(
  'silver.trx', 4820199935123456789);
```

Kegunaannya di bank

- **Audit**: angka laporan bulan lalu dapat direproduksi persis seperti saat dikirim
- **Pemulihan**: *job* yang merusak data dibatalkan dalam detik
- **ML**: *training data* model dapat dicatat sebagai nomor *snapshot*

Jebakan kepatuhan

Data yang di-DELETE masih ada di *snapshot* lama. Untuk hak hapus UU PDP, *snapshot* harus **dikedaluwarsakan** dan *file*-nya benar-benar dihapus (P14–P15).

Schema evolution: mengubah kolom tanpa menulis ulang data

Operasi yang aman di Iceberg

- Tambah kolom (`kanal`: *mobile*, ATM, agen)
- Ganti nama kolom
- Hapus kolom
- Perlebar tipe (`int` → `long`)
- Ubah urutan kolom

Semua hanya mengubah metadata; *file* Parquet lama tidak disentuh.

Mengapa bisa aman

Iceberg memberi setiap kolom **ID** tetap. *File* lama dibaca menurut ID, bukan menurut nama atau posisi. Kolom baru terbaca sebagai *null* pada *file* lama; kolom yang diganti nama tetap menunjuk data yang sama.

Tanpa *table format*

Mengganti nama kolom di Parquet polos berarti *file* lama tiba-tiba "kehilangan" kolom itu, atau semua data harus ditulis ulang.

Hidden partitioning dan *partition evolution*

```
CREATE TABLE lake.silver.trx (
  id_trx      BIGINT,
  id_rekening BIGINT,
  waktu       TIMESTAMP,
  nominal     DECIMAL(18,2),
  kanal       STRING)
USING iceberg
PARTITIONED BY (days(waktu));

-- pengguna cukup memfilter kolom aslinya:
SELECT * FROM lake.silver.trx
WHERE waktu >= '2026-11-01';

-- volume naik: tambah partisi tanpa menulis ulang
ALTER TABLE lake.silver.trx
  ADD PARTITION FIELD bucket(16, id_rekening);
```

- ***Hidden partitioning***: *partition* diturunkan dari kolom (`days(waktu)`); pengguna tidak perlu tahu atau menulis kolom `tanggal` terpisah.
- Pada folder Parquet polos, lupa memfilter kolom *partition* berarti memindai seluruh tabel. Di sini filter pada `waktu` otomatis memangkas *partition*.
- ***Partition evolution***: *schema partition* boleh berubah. Data lama tetap dengan *schema* lama, data baru dengan *schema* baru; *query* menangani keduanya.

MERGE, *compaction*, dan perawatan tabel

MERGE: koreksi dan *upsert*

```
MERGE INTO lake.silver.trx t
USING koreksi_harian k
ON t.id_trx = k.id_trx
WHEN MATCHED AND k.status = 'BATAL'
  THEN DELETE
WHEN MATCHED
  THEN UPDATE SET *
WHEN NOT MATCHED
  THEN INSERT *;
```

Satu *commit* atomik: pembaca melihat sebelum atau sesudah, tidak pernah di tengah. Hari ini MERGE dicoba lewat Trino atau `merge` di delta-rs; Spark SQL menyusul di P9–P10.

Dua cara menulis perubahan baris

- ***Copy-on-write***: *file* yang memuat baris berubah ditulis ulang utuh. Menulis mahal, membaca murah. Cocok untuk koreksi jarang.
- ***Merge-on-read***: tulis *delete file* kecil yang menandai baris terhapus; pembaca menggabungkan saat membaca. Menulis murah, membaca makin mahal sampai *compaction*. Cocok untuk perubahan sering (CDC, P11).

Perawatan tabel: tiga tugas rutin

```sql
-- 1. satukan berkas kecil
CALL lake.system.rewrite_data_files(
  table => 'silver.trx');

-- 2. buang snapshot lama (dan berkasnya)
CALL lake.system.expire_snapshots(
  table       => 'silver.trx',
  older_than  => TIMESTAMP '2026-10-01 00:00:00',
  retain_last => 10);

-- 3. hapus berkas yatim dari job yang gagal
CALL lake.system.remove_orphan_files(
  table => 'silver.trx');
```

- ***Compaction***: *streaming* dan *merge-on-read* menghasilkan banyak *file* kecil dan *delete file*; *compaction* menyatukannya (masalah *small files* di P3).
- ***Expire snapshots***: menentukan berapa lama *time travel* tersedia. Ini **keputusan kebijakan**: audit ingin lama, privasi dan *cost* ingin singkat.
- ***Orphan files***: *file* yang ditulis *job* gagal tetapi tidak pernah di-*commit*.

Ringkasan

Jadwalkan ketiganya di Airflow (P6). Tabel *lakehouse* tanpa perawatan lambat laun kembali menjadi rawa.

Katalog tabel dan *query engine*

Katalog: “tabel `silver.trx` ada di mana?”

Katalog	Ciri	Cocok bila
Hive Metastore	Warisan Hadoop, didukung hampir semua mesin, butuh *database* relasional di belakangnya	Sudah punya ekosistem Hadoop/Hive
REST catalog (Iceberg)	Spesifikasi HTTP standar; implementasinya beragam (Apache Polaris, Lakekeeper, Gravitino, layanan *cloud*)	Ingin netral vendor, akses lintas mesin dan bahasa
Nessie	Versi ala Git: *branch*, *commit*, *merge* lintas banyak tabel	Uji perubahan *pipeline* di *branch* sebelum ke produksi
Unity Catalog, AWS Glue	Terintegrasi dengan platform vendor, termasuk kendali akses	Platform sudah Databricks atau AWS

Ringkasan

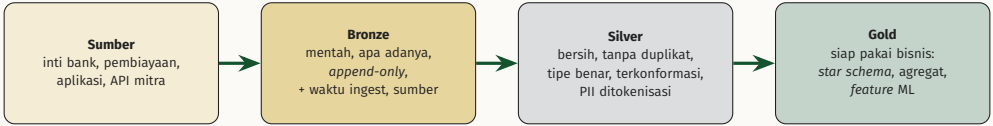
Katalog adalah titik koordinasi *commit* **dan** titik kendali akses. Pilihan katalog lebih sulit diganti daripada pilihan *query engine*.

Query engine di atas *lakehouse*

Mesin	Kekuatan	Peran khas
Spark SQL	Menulis dan membaca, MERGE, prosedur perawatan, skala besar (P9–P10)	*Pipeline* bronze → silver → gold
Trino	SQL interaktif terdistribusi, *federation* ke banyak sumber	Analis dan *dashboard* atas *gold layer*
Flink	Menulis *streaming* ke Iceberg dengan *exactly-once*	Jalur *hot* ke tabel *lakehouse* (P11)
DuckDB	Membaca Iceberg/Delta langsung dari laptop	Eksplorasi, uji, analisis kecil-menengah
PyIceberg, delta-rs	*Library* Python/Rust tanpa JVM	*Job* kecil, pengujian, integrasi aplikasi

Medallion architecture

Bronze, silver, gold



	Bronze	**Silver**	**Gold**
Pengguna	*Data engineer*	*Engineer*, *data scientist*	Analis, *dashboard*, regulator
Contoh	`bronze.trx_core` (JSON CDC)	`silver.trx` (satu baris per transaksi)	`gold.fakta_trx_harian`
Aturan	Jangan diubah; ulang ingest bila perlu	*Data quality test* wajib lulus (P6)	Definisi *metric* disepakati (P13)

Aturan tiap *layer*

	Bronze	**Silver**	**Gold**
Tujuan	Salinan setia dari *source*, bisa diputar ulang	Satu versi bersih dan konsisten per entitas	Jawaban siap pakai untuk pertanyaan bisnis
Schema	Longgar: kolom asli + `_ingest_ts`, `_source`, `_batch_id`	Ketat: tipe benar, nama baku, *key* jelas	*Star schema*: *fact* dan *dimension*, atau tabel agregat
Cara menulis	*Append-only*; tidak pernah UPDATE	*Deduplication*, MERGE koreksi, SCD tipe 2	*Overwrite* per *partition* atau MERGE, *idempotent*
Data quality	Cek minimal: *file* terbaca, jumlah baris	Wajib lulus: *not null*, unik, tipe, rentang, *referential*	Rekonsiliasi total dengan *source*, definisi *metric*
PII	Ada dalam bentuk mentah	Di-*tokenize* atau di-*mask*	Tidak ada, kecuali benar-benar perlu
Akses	Sangat terbatas (*data engineer*)	*Engineer*, *data scientist*	Analis, *dashboard*, regulator
Retention	Pendek–sedang, lalu *cold*/*archive*	Sesuai kebutuhan analitik	Sesuai kebutuhan laporan

Sumber: Pola umum *medallion architecture*; nama dan jumlah *layer* bisa berbeda antar organisasi.

Contoh tabel bank syariah per *layer*

Bronze	Silver	Gold
`bronze.core_trx_cdc` *event* CDC JSON dari *database* inti	`silver.trx` satu baris per transaksi, status final	`gold.fakta_trx_harian` per cabang, produk, kanal, hari
`bronze.nasabah_raw` ekspor harian, memuat NIK dan alamat	`silver.nasabah` NIK diganti `nik_token`, alamat dibakukan	`gold.dim_nasabah` SCD tipe 2: segmen, kota, tanggal berlaku
`bronze.pembiayaan_raw` akad murabahah, musyarakah, ijarah	`silver.akad` dan `silver.angsuran` tipe dan tanggal benar	`gold.npf_bulanan` kolektibilitas per produk (definisi ikut OJK)
`bronze.app_event` klik dan login aplikasi *mobile*	`silver.sesi_app` sesi per perangkat, bot dibuang	`gold.fitur_fraud` *feature* per rekening untuk model ML

Penamaan `<layer>.<entitas>` membuat siapa pun tahu tingkat kepercayaan dan aturan akses sebuah tabel hanya dari namanya.

Satu transaksi melewati tiga *layer*

Bronze — apa adanya dari CDC

```
{"op":"c","ts_ms":1793520000123,
 "after":{"id":"T-88123","rek":"0071234",
  "nik":"3201xxxxxxxxxxx","nominal":"250000",
  "kanal":"QRIS","waktu":"2026-11-02 09:14:05"},
 "_ingest_ts":"2026-11-02T09:14:07Z",
 "_source":"core.transaksi"}
```

Silver — bersih, bertipe, tanpa PII mentah

id_trx	rek_token	nominal	kanal	jam
T-88123	rk_9f2c..	250000	QRIS	09:14:05

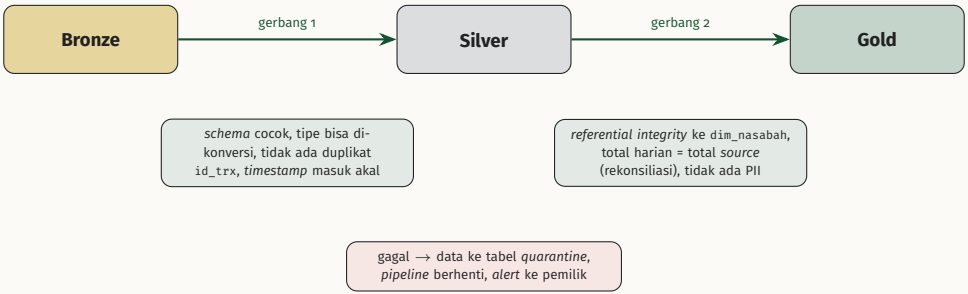
Gold — jawaban bisnis

tanggal	cabang	kanal	jml_trx	total
2026-11-02	BGR01	QRIS	18204	3.1e9

- Bronze menyimpan `nominal` sebagai teks persis seperti sumber; silver yang mengubahnya ke angka.
- NIK hilang di silver; rekening diganti *token* (P14).
- Gold hanya angka agregat, aman dibaca analis dan *dashboard*.
- Bila logika silver salah, bangun ulang dari bronze tanpa menarik lagi dari *source*.

Data ilustrasi, bukan data nasabah sungguhan.

Gerbang *data quality* di antara *layer*



Ringkasan

Layer yang tidak punya gerbang hanyalah nama folder. Gerbangnya ditulis sebagai *data quality test* di dbt atau Great Expectations dan dijalankan oleh *orchestrator* (P6).

Mengapa bertingkat, dan kapan tidak perlu tiga

Alasan bertingkat

- **Bisa diulang**: bila logika silver salah, bangun ulang dari bronze tanpa menarik lagi dari sumber
- **Pemisahan tanggung jawab**: tiap *layer* punya pemilik dan *data quality test* sendiri
- **Keamanan bertingkat**: hanya sedikit orang boleh membaca bronze yang memuat PII mentah; analis cukup gold
- ***Cost***: bronze jarang dibaca setelah diproses, kandidat *tier* dingin

Jangan dogmatis

Tiga *layer* adalah pola yang lazim dan boleh disesuaikan. Sumber yang sudah bersih boleh langsung ke silver; beberapa organisasi memakai nama *raw–staging–mart*. Yang penting: setiap *layer* punya **kontrak** yang jelas.

Kaitan dengan P2

Star schema Kimball yang Anda rancang di P2 hidup di *gold layer*. *Slowly changing dimension* tipe 2 dibuat dari silver dengan *snapshot* dbt (P6).

Lakehouse dan *storage tiering* (P3)

Bagian	***Tier* yang wajar**
Metadata, *manifest*	***hot***: selalu dibaca setiap *query*
Gold, *partition* 13 bulan	***warm***
Silver, *partition* lama	***cold*** (*infrequent access*)
Bronze lama, *snapshot* yang wajib disimpan	**archive**, dengan kebijakan retensi

- Karena data dipartisi menurut waktu, *lifecycle policy object storage* dapat memindahkan *file partition* lama ke kelas lebih murah.
- **Hati-hati**: *file* di kelas arsip tidak bisa dibaca langsung. *Query* yang menyentuhnya gagal atau menunggu *restore*. Pastikan metadata dan *snapshot* yang masih aktif tidak menunjuk *file* yang sudah diarsipkan.
- *Retrieval cost* dari *tier* dingin bisa melebihi penghematannya bila datanya ternyata sering dibaca, ukur pola akses dulu.

When to use

Iceberg vs Delta Lake vs Hudi

	Apache Iceberg	**Delta Lake**	**Apache Hudi**
Asal	Netflix	Databricks	Uber
Kekuatan	Netral mesin, *hidden partitioning*, *partition evolution*, spesifikasi rinci	Integrasi terdalam dengan Spark dan Databricks; delta-rs untuk Python/Rust	*Upsert* dan *incremental pull* untuk beban CDC berat
Metadata	Pohon *manifest*	Log JSON + *checkpoint*	*Timeline commit*
Cocok bila	Banyak mesin (Spark, Flink, Trino, DuckDB) membaca tabel yang sama	Platform utama Databricks/Spark	Banyak *update* per *key*, kebutuhan *near real-time*

Ketiganya terus saling mengejar *feature*, dan ada *layer* penerjemah metadata antar-format (misalnya Apache XTable, Delta UniForm). Periksa versi terkini sebelum memutuskan.

Ringkasan

Di mata kuliah ini kita memakai **Iceberg** karena netral mesin. Kelompok boleh memakai Delta untuk tugas besar.

Warehouse, *lake*, atau *lakehouse*?

	Cloud warehouse	***Data lake* polos**	***Lakehouse***
Format	Milik vendor	Terbuka	Terbuka + metadata
ACID, UPDATE	Ya	Tidak	Ya
Kinerja SQL	Sangat baik, tanpa diatur	Bergantung *layout*	Baik, perlu perawatan
Operasi	Terkelola vendor	Murah tetapi rawan rawa	Perlu katalog dan perawatan
Cocok bila	Tim kecil, kebutuhan BI dominan, siap terikat vendor	Arsip mentah, data tak terstruktur	Banyak mesin, BI dan ML dari satu salinan, *on-premise* atau multi-*cloud*

Banyak organisasi memakai kombinasi: *lakehouse* sebagai pusat, *warehouse* atau *real-time* OLAP untuk beban tertentu.

Best practice industri untuk *lakehouse*

1. **Satu katalog sebagai sumber kebenaran**; jangan menulis tabel yang sama lewat dua katalog.
2. ***Partition* menurut waktu**, biarkan *hidden partitioning* bekerja.
3. **Jadwalkan perawatan**: *compaction*, *expire snapshots*, *orphan files*.
4. **Tetapkan retensi *snapshot*** bersama tim kepatuhan, bukan default.
5. **Bronze tidak diubah**; semua koreksi terjadi di silver.
6. ***Data quality test* sebelum *commit*** ke silver dan gold (P6).
7. ***Tokenization* PII** saat bronze → silver (P14).
8. **Kendali akses di katalog**; izin *bucket* saja terlalu kasar.
9. **Catat nomor *snapshot*** pada setiap laporan resmi dan *training data*.
10. **Ukur sebelum memindah ke *tier* dingin**.

Praktikum

Praktikum: MinIO + katalog REST Iceberg

```
services:
  minio:
    image: minio/minio
    command: server /data --console-address :9001
    ports: ["9000:9000", "9001:9001"]
  iceberg-rest:
    image: apache/iceberg-rest-fixture
    ports: ["8181:8181"]
    environment:
      CATALOG_WAREHOUSE: s3://lake/
      CATALOG_S3_ENDPOINT: http://minio:9000
      # + IO impl S3FileIO dan kredensial
```

Kerangka lengkap (kredensial, *bucket* awal) ada di repositori kelas; Spark dengan paket Iceberg ditambahkan di P9. Nama *image* dan variabelnya dapat berubah antar versi.

1. Jalankan MinIO dan katalog REST, buat *bucket* `lake`.
2. Buat *namespace* `bronze`, `silver`, `gold`.
3. Muat transaksi sintetis dari P2–P3 ke `bronze.trx` apa adanya.
4. Bersihkan ke `silver.trx`: buang duplikat, perbaiki tipe, *partition* `days(waktu)`.
5. *Aggregation* ke `gold.trx_harian_cabang`.

Praktikum: PyIceberg tanpa JVM

```python
import pyarrow.parquet as pq
from pyiceberg.catalog import load_catalog
from pyiceberg.types import StringType

cat = load_catalog("lake", type="rest",
    uri="http://localhost:8181",
    **{"s3.endpoint": "http://localhost:9000"})
cat.create_namespace("bronze")
data = pq.read_table("data/trx_2026-11-01.parquet")
tbl = cat.create_table("bronze.trx", schema=data.schema)
tbl.append(data)                        # snapshot 1
tbl.append(pq.read_table("data/trx_2026-11-02.parquet"))
for s in tbl.snapshots():               # riwayat
    print(s.snapshot_id, s.timestamp_ms)
sid0 = tbl.snapshots()[0].snapshot_id
lama = tbl.scan(snapshot_id=sid0)       # time travel
print(lama.to_arrow().num_rows)
with tbl.update_schema() as u:          # schema evolution
    u.add_column("kanal", StringType())
```

- Buka MinIO di peramban: lihat folder `data/` dan `metadata/` yang tercipta. Buka satu `metadata.json` dan temukan daftar *snapshot*.
- Perhatikan: menambah kolom tidak menulis ulang satu pun *file* Parquet.

Lalu: koreksi tanpa Spark

Ganti satu *partition* `silver.trx` dengan data koreksi lewat `tbl.overwrite(..., overwrite_filter=...)`, lalu baca lagi versi lama dengan `snapshot_id`. MERGE penuh dan `rewrite_data_files` kita ulang dengan Spark SQL di P9–P10.

Praktikum: membaca *gold layer* dari DuckDB

```
INSTALL iceberg; LOAD iceberg;
INSTALL httpfs;  LOAD httpfs;

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

SELECT cabang, SUM(total) AS total_nov
FROM iceberg_scan('s3://lake/gold/trx_harian_cabang')
WHERE tanggal BETWEEN '2026-11-01' AND '2026-11-30'
GROUP BY cabang
ORDER BY total_nov DESC;
```

Kredensial di atas hanya untuk lab lokal. Di produksi, kredensial tidak pernah ditulis di kode (P14).

Yang dikumpulkan

1. Diagram tiga *layer* Anda: tabel, *schema*, *partition*, siapa pembacanya
2. Bukti *time travel* dan *rollback* (nomor *snapshot* sebelum dan sesudah)
3. Bukti *schema evolution* tanpa menulis ulang data
4. Daftar *file* per *snapshot* (sebelum dan sesudah *overwrite*)
5. Satu *query* gold dari DuckDB atau Trino

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum *lakehouse* tiga *layer*.
2. Tulis satu halaman: kebijakan retensi *snapshot* untuk `silver.trx` (berapa lama, mengapa), dengan mempertimbangkan audit, hak hapus UU PDP, dan *cost*.
3. Mulai rancang tugas besar UTS: *dataset* yang dipilih, aturan tiap *layer*, dan tabel gold yang akan dihasilkan.
4. Baca Reis dan Housley Bab 8 (*queries, modeling, and transformation*) dan dokumentasi Polars bagian *lazy API*.

Pertemuan depan

ETL di satu mesin: Apache Arrow, Polars, DuckDB, dan DataFusion, mengisi bronze, silver, dan gold dengan cepat tanpa *cluster*.

Terima kasih

Pertanyaan?