# Data Engineer Handboek

Praktische handleiding voor data engineers

Datapartner365

by Datapartner365

# Inhoudsopgave

HOOFDSTUK 1

# Introductie in Data Engineering

## Introductie

Data engineering is het vakgebied dat zich bezighoudt met het verplaatsen van data van de plek waar het ontstaat naar de plek waar het bruikbaar is — betrouwbaar en op schaal. Dat klinkt eenvoudig. In de praktijk betekent het systemen ontwerpen die data binnenhalen uit tientallen bronnen met verschillende formaten en update-frequenties, die data opslaan zodat je er goedkoop en snel op kunt bevragen, die data omzetten naar iets wat een business user of een machine learning-model daadwerkelijk kan gebruiken, en dat allemaal op een manier die blijft werken als een bronsysteem dinsdagnacht om 3 uur ineens zijn schema wijzigt.

Het vakgebied bestaat omdat niets hiervan vanzelf gaat. Een productiedatabase is geoptimaliseerd voor transacties, niet voor analyse — een zware aggregatiequery daartegen uitvoeren vertraagt de applicatie die hij bedient, en soms blokkeert het rijen die een klant op dat moment probeert te wijzigen. Een dashboard dat rechtstreeks op ruwe applicatietabellen is gebouwd, breekt zodra een product-engineer een kolom hernoemt, omdat niemand buiten het applicatieteam wist dat die kolom stroomafwaarts werd gebruikt. Data engineers bouwen de laag ertussenin: pipelines, warehouses en transformatielogica die operationele chaos omzetten in iets stabiels genoeg om een bedrijf op te bouwen.

Het is de moeite waard om de scope precies af te bakenen, want "data engineering" wordt vaak als verzamelnaam gebruikt voor alles wat met data te maken heeft. Dit hoofdstuk — en dit handbook — richt zich op **analytische data engineering**: het bouwen van de systemen die rapportage, analytics en machine learning aandrijven. Dat is iets anders dan, hoewel overlappend met, backend engineering (het bouwen van de applicatiedatabases zelf) en MLOps (het deployen en monitoren van modellen in productie). De grenzen zijn vaag bij kleinere bedrijven waar één persoon alle drie doet, maar de vaardigheden en denkmodellen zijn verschillend genoeg om apart te behandelen.

## Een Korte Geschiedenis: van ETL Developer naar Data Engineer

Begrijpen waar de rol vandaan komt, verklaart waarom die er nu zo uitziet. Door de jaren 2000 en vroege 2010 heette de equivalente rol meestal "ETL Developer" of "BI Developer", met tools als Informatica, SSIS of Talend. Het patroon was star: **E**xtract uit bronsystemen, **T**ransform de data via een dedicated ETL-server (omdat het doelwarehouse — meestal een on-premises SQL Server- of Oracle-instantie — beperkte en dure compute had), en pas dan **L**oad alleen het uiteindelijke, gemodelleerde resultaat naar het warehouse. Transformatielogica leefde in proprietary tool-configuraties die lastig te versiebeheren waren en nog lastiger te testen.

Twee dingen braken dit model open. Ten eerste ontkoppelden cloud data warehouses (Redshift in 2012, daarna Snowflake, daarna BigQuery) storage van compute en maakten warehouse-compute goedkoop en elastisch — ineens was het logischer om ruwe data eerst te laden en die *binnen* het warehouse te transformeren met SQL, omdat je niet langer een premium betaalde voor die compute. Dit is de verschuiving van ETL naar ELT, uitgebreid behandeld in ETL vs ELT. Ten tweede explodeerde de diversiteit en het volume aan databronnen — event streams, third-party API's, SaaS-tools, IoT — sneller dan point-and-click ETL-tools redelijkerwijs konden configureren. De rol ging echte software engineering-vaardigheden vereisen: versiebeheer, testen, code review, CI/CD. "Data Engineer" als aparte functietitel brak door tussen ongeveer 2015 en 2018, en begin jaren 2020 was het een van de meest gevraagde rollen in tech geworden — een trend die verder wordt uitgewerkt in Becoming a Data Engineer.

## Wat Data Engineers Daadwerkelijk Bouwen

Vacatureteksten zeggen vaak "beheert data-infrastructuur", wat je bijna niets vertelt. Concreet valt het dagelijkse werk van een data engineer meestal in vier categorieën.

**Ingestion pipelines.** Code (of geconfigureerde tools) die data ophaalt uit bronsystemen: applicatiedatabases via change data capture (CDC), third-party API's (Stripe, Salesforce, HubSpot), event streams via Kafka of Kinesis, bestanden die door externe partners in cloud storage worden gezet. Dit is het deel dat de meeste mensen voor zich zien bij "data pipeline", maar het is vaak het minst intellectueel

uitdagende deel van het werk zodra een patroon staat — het moeilijke zit in de lange staart aan edge cases: rate limits, paginering, gedeeltelijke fouten, API-schemawijzigingen.

**Storage design.** Bepalen hoe data georganiseerd wordt zodra die binnenkomt: partitioneringsstrategie (op datum is het meest gebruikelijk, maar niet altijd juist), bestandsformaten (Parquet, Delta, Iceberg — behandeld in Delta Lake), en tabelstructuren in een warehouse. Slechte storage design toont zich pas maanden later als trage queries en oplopende compute-kosten, wat precies de reden is dat het makkelijk fout gaat en duur is om later te repareren. Een tabel gepartitioneerd op de verkeerde kolom betekent dat elke query de hele dataset scant, ongeacht het toegepaste filter.

**Transformatielogica.** SQL of code die ruwe, rommelige data omzet naar schone, gemodelleerde tabellen: deduplicatie, type-casting, businesslogica, joins over bronnen heen, slowly changing dimension-verwerking. Hier zit het meeste echte engineering-oordeel, en het wordt uitgebreid behandeld in de hoofdstukken over dbt en Medallion Architecture. Een verrassend groot deel van "data engineering" in een volwassen organisatie is eigenlijk dit — niet bytes verplaatsen, maar businesslogica correct vastleggen en onderhoudbaar houden terwijl het bedrijf verandert.

**Orchestratie en betrouwbaarheid.** Pipelines plannen, fouten en retries afhandelen, alerteren wanneer data niet op tijd binnenkomt, en zorgen dat een fout in de ene pipeline niet stilletjes alles stroomafwaarts corrumpeert. Dit is de operationele helft van het werk die makkelijk wordt onderschat totdat je om 2 uur 's nachts gepiept wordt omdat een valuta-conversie-API stilletjes nulls begon terug te geven.

## De Moderne Data Engineering Stack

De meeste dataplatformen worden tegenwoordig samengesteld uit gespecialiseerde tools in plaats van één monolithisch systeem — een patroon dat de industrie de "modern data stack" noemt. De lagen begrijpen is belangrijker dan specifieke productnamen uit je hoofd leren, want de producten veranderen elke paar jaar en de lagen niet.

**Ingestion.** Tools als Fivetran, Airbyte, of de copy-activity van Azure Data Factory halen data uit bronnen en laden die, grotendeels ongewijzigd, naar storage. Dit is de "EL" in ELT. Voor use cases met hoog volume en lage latency is deze laag mogelijk in plaats daarvan een Kafka- of Event Hubs-stream. Zie Azure Data Factory.

**Storage.** Een cloud data warehouse (Snowflake, BigQuery, Redshift) of een lakehouse (Databricks, Microsoft Fabric) bevat de data, waarbij storage-kosten losstaan van compute-kosten — je betaalt voor de schijfruimte die je data inneemt, onafhankelijk van de compute die je opstart om die te bevragen. Deze ene architectuurkeuze maakt de rest van de moderne stack economisch haalbaar. Zie Snowflake, Databricks en Microsoft Fabric.

**Transformatie.** dbt, Spark, of platte SQL-scripts draaien binnen het warehouse om ruwe data om te vormen naar gemodelleerde, betrouwbare tabellen. Dit is de "T" in ELT — transformatie gebeurt *na* het laden, met de eigen compute van het warehouse, wat de bepalende verschuiving is ten opzichte van de oudere ETL-aanpak.

**Orchestratie.** Airflow, Dagster, of de native scheduler van een platform (Fabric pipelines, Databricks Workflows) bepaalt wanneer elke stap draait, in welke volgorde, en wat er gebeurt als iets faalt. Orchestratie maakt van een verzameling scripts een systeem dat je kunt vertrouwen.

**Serving.** BI-tools (Power BI, Tableau, Looker), reverse ETL-tools die gemodelleerde data terugduwen naar operationele tools (Salesforce, HubSpot), of API's ontsluiten de uiteindelijke, gemodelleerde data naar de mensen en systemen die het daadwerkelijk gebruiken. Deze laag wordt vaak over het hoofd gezien in discussies over data engineering, maar is het hele punt — een perfect gebouwde pipeline die niemand kan bevragen is waardeloos.

Officiële referentiepunten per laag staan onderaan dit hoofdstuk.

## Batch versus Streaming

Een fundamentele keuze bij elk pipeline-ontwerp is of data in **batches** beweegt (verwerk alles wat is opgebouwd sinds de laatste run, volgens een schema) of als een **stream** (verwerk elk event continu, zodra het binnenkomt). Deze keuze bepaalt bijna alles wat erna

komt.

Batch processing is eenvoudiger te bouwen, makkelijker te doorgronden, makkelijker te backfillen en goedkoper om te draaien — je start compute op, verwerkt een blok data, en sluit hem weer af. De meeste analytics use cases (dagelijkse omzetrapportages, wekelijkse cohortanalyse) hebben geen data nodig die verser is dan een paar uur oud, dus batch blijft om goede redenen de standaard. De afweging is latency: als je pipeline elke 6 uur draait, is je data per definitie tot 6 uur oud.

Streaming verwerkt elk record zodra het binnenkomt — met tools als Kafka, Kinesis, Spark Structured Streaming of Flink — en is nodig wanneer het bedrijf echt sub-minuut-versheid nodig heeft: fraudedetectie, real-time personalisatie, operationele dashboards die live systeemgezondheid tonen. Streaming-systemen zijn aanzienlijk moeilijker correct te bouwen: je moet omgaan met events die niet in volgorde binnenkomen, exactly-once-verwerkingsgaranties, en state die blijft bestaan in een systeem dat nooit echt "klaar" is met draaien.

De praktische vuistregel waar de meeste ervaren engineers op uitkomen: **kies standaard voor batch, en grijp alleen naar streaming als er een specifieke, gekwantificeerde businessvereiste is voor lage latency die batch écht niet kan halen.** Een groot deel van "we hebben real-time nodig" blijkt bij nader inzien te betekenen "we hebben dit binnen het uur nodig", wat een goed afgestelde batch-pipeline die elke 15 minuten draait prima aankan, tegen een fractie van de engineering- en operationele kosten.

## Een Dag uit het Leven van een Data Pipeline

Neem een concreet voorbeeld: een e-commercebedrijf wil dagelijks omzet per productcategorie zien, en de bron is een `orders`-tabel in een productie-Postgres-database.

**Stap 1 — Extract & Load.** Een nachtelijke job leest nieuwe en gewijzigde rijen uit `orders` met een `updated_at`-watermark, niet met een volledige tabelscan, en zet die, grotendeels ongewijzigd, in een `raw.orders`-tabel in het warehouse. Dit is de **bronze**-laag: ongewijzigd, maar nu bevraagbaar zonder productie aan te raken. De extractiejob moet het geval afhandelen waarin de vorige run halverwege faalde — opnieuw draaien mag geen dubbele rijen creëren of rijen overslaan, wat precies de reden is dat de watermark-aanpak gecombineerd moet worden met idempotent laden (hieronder meer).

**Stap 2 — Clean & Standardize.** Een transformatiestap dedupliceert rijen (dezelfde order kan tweemaal voorkomen als er tijdens extractie een netwerk-retry plaatsvond), zet `order_date` om naar een echt datumtype, verwijdert testorders die door het QA-team zijn geplaatst, en valideert dat verplichte velden zoals `customer_id` niet null zijn. De output komt in `silver.orders` terecht — één schone, betrouwbare rij per echte order. Hier vang je ook misvormde records op en zet je die apart, in plaats van ze stilletjes downstream-aggregaties te laten breken.

**Stap 3 — Model & Aggregate.** Een laatste stap koppelt `silver.orders` aan `silver.products`, groepeert op categorie en dag, en schrijft het resultaat naar `gold.revenue_by_category`. Dit is wat het BI-dashboard daadwerkelijk bevraagt — een kleine, snelle, vooraf geaggregeerde tabel in plaats van een query die miljoenen ruwe orderrijen scant bij elke dashboard-refresh.

Elke stap is een aparte, testbare eenheid. Als stap 2 breekt, is de output van stap 1 er nog steeds en krijgt stap 3 gewoon geen verse data — het produceert niet stilletjes rommel op basis van halfgeschoonde input. Die scheiding, geformaliseerd als bronze/silver/gold, is het onderwerp van het hoofdstuk Medallion Architecture, en het is een van de meest bepalende architectuurbeslissingen in dit hele handbook, omdat het bepaalt hoe gracieus je platform degradeert wanneer er onvermijdelijk iets misgaat.

## Codevoorbeelden

Hieronder zie je hoe de extract-, load- en transformstappen hierboven er in de praktijk uitzien — bewust vereenvoudigd voor de leesbaarheid, maar structureel hetzelfde patroon dat op veel grotere schaal in productie draait.

**Extractie met een watermark**, zodat we alleen ophalen wat er is veranderd sinds de laatste run:

```
# extract_load.py — haalt nieuwe orders op uit Postgres naar het warehouse
import snowflake.connector
import psycopg2
```

```
def get_watermark(sf_conn):
    """Geeft de laatste updated_at terug die we al hebben geladen, of het epoch."""
    cur = sf_conn.cursor()
    cur.execute("SELECT MAX(updated_at) FROM raw.orders")
    result = cur.fetchone()[0]
    return result or "1970-01-01"

def extract_new_orders(pg_conn, since):
    cur = pg_conn.cursor()
    cur.execute(
        "SELECT id, customer_id, order_date, amount, updated_at "
        "FROM orders WHERE updated_at > %s ORDER BY updated_at",
        (since,)
    )
    return cur.fetchall()

def load_to_staging(sf_conn, rows):
    """Laad eerst naar een staging-tabel — schrijf nooit direct naar raw.orders,
    zodat een mislukte load die tabel nooit half-geschreven achterlaat."""
    cur = sf_conn.cursor()
    cur.executemany(
        "INSERT INTO staging.orders (id, customer_id, order_date, amount, updated_at) "
        "VALUES (%s, %s, %s, %s, %s)",
        rows
    )
```

**Idempotent laden met MERGE**, zodat het opnieuw draaien van deze job na een gedeeltelijke fout nooit duplicaten creëert:

```
-- merge_raw.sql — staging naar raw, idempotent by design
MERGE INTO raw.orders AS tgt
USING staging.orders AS src
ON tgt.id = src.id
WHEN MATCHED THEN
  UPDATE SET amount = src.amount, updated_at = src.updated_at
WHEN NOT MATCHED THEN
  INSERT (id, customer_id, order_date, amount, updated_at)
  VALUES (src.id, src.customer_id, src.order_date, src.amount, src.updated_at);
```

**Deduplicatie en opschoning** naar de silver-laag, met `ROW_NUMBER()` om alleen de laatste versie van elke order te bewaren:

```
-- transform_silver.sql — dedup + opschonen, één rij per echte order
CREATE OR REPLACE TABLE silver.orders AS
SELECT
    order_id,
    customer_id,
    order_date,
    amount,
    updated_at
FROM (
    SELECT
        id AS order_id,
        customer_id,
        CAST(order_date AS DATE) AS order_date,
        amount,
        updated_at,
        ROW_NUMBER() OVER (PARTITION BY id ORDER BY updated_at DESC) AS rn
    FROM raw.orders
    WHERE amount > 0        -- test-/geannuleerde orders eruit
      AND customer_id IS NOT NULL
)
WHERE rn = 1;
```

Dat patroon van `ROW_NUMBER() OVER (PARTITION BY ...)` voor deduplicatie is een van de meest voorkomende dingen die je als data engineer zult schrijven — het krijgt een veel diepere behandeling, inclusief performance-overwegingen en Snowflake's `QUALIFY`-shortcut, in SQL for Data Engineers.

**Een minimale Airflow DAG** die de stappen samenbrengt met goede foutafhandeling:

```
# orders_pipeline_dag.py
from airflow import DAG
```

```
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

default_args = {
    "retries": 2,
    "retry_delay": timedelta(minutes=5),
    "email_on_failure": True,
}

with DAG(
    "orders_pipeline",
    schedule="0 3 * * *",       # dagelijks om 03:00
    start_date=datetime(2026, 1, 1),
    default_args=default_args,
    catchup=False,
) as dag:

    extract = PythonOperator(task_id="extract_load", python_callable=run_extract_load)
    merge = PythonOperator(task_id="merge_raw", python_callable=run_merge)
    clean = PythonOperator(task_id="build_silver", python_callable=run_silver_transform)
    aggregate = PythonOperator(task_id="build_gold", python_callable=run_gold_aggregate)

    extract >> merge >> clean >> aggregate
```

Let op de `retries` en `retry_delay` — tijdelijke fouten (een haperende databaseverbinding, een API-timeout) zijn de norm in productiepipelines, niet de uitzondering. Een pipeline die hard faalt bij de eerste tijdelijke fout piept onnodig iemand op, meerdere keren per week.

## Data Engineer versus Data Analist versus Data Scientist versus ML Engineer

Deze rollen worden voortdurend door elkaar gehaald, dus hier is het praktische verschil in termen van waar elke rol daadwerkelijk verantwoordelijk voor is:

| Rol | Belangrijkste output | Belangrijkste tools | Gaat ervan uit dat... |
|---|---|---|---|
| Data Engineer | Pipelines, warehouses, gemodelleerde tabellen | SQL, Python, orchestratietools | Ruwe data ergens bestaat, hoe rommelig ook |
| Data Analist | Dashboards, rapportages, business inzicht | SQL, BI-tools (Power BI, Tableau) | Gemodelleerde tabellen al bestaan en betrouwbaar zijn |
| Data Scientist | Modellen, experimenten, statistische bevindingen | Python, statistiek, ML-frameworks | Feature-ready data al bestaat |
| ML Engineer | Gedeployde, gemonitorde modellen in productie | Python, MLOps-tooling, soms Spark | Er een getraind model bestaat dat betrouwbaar moet draaien |

Een data analist gaat ervan uit dat de data al schoon en beschikbaar is. Het werk van een data engineer is om die aanname waar te maken — en waar te houden terwijl bronsystemen, businesslogica én datavolume tegelijk veranderen. Een data scientist bouwt voort op beide, en heeft in toenemende mate ook pipeline-vaardigheden nodig, aangezien feature engineering op schaal zelf een data engineering-probleem is — een trend die verder wordt behandeld in AI for Data Engineers. De grens tussen data engineer en ML engineer is behoorlijk vervaagd nu feature stores en real-time inference-pipelines gemeengoed zijn geworden; bij veel organisaties is het inmiddels hetzelfde team.

## Performance Tips

- **Filter vóór je joint, niet erna.** Een `WHERE`-clausule toepassen na een grote join dwingt het warehouse om de volledige dataset te joinen voordat rijen worden weggegooid. Duw filters zo vroeg mogelijk in de query — de meeste moderne query-optimizers doen dit automatisch, maar niet altijd, zeker niet over CTE's heen.

- **Partitioneer op de kolom waar je het meest op filtert.** Als 90% van de queries op datum filtert, partitioneer dan op datum. Partitioneren op de verkeerde kolom betekent dat elke query data scant die hij niet nodig heeft, ongeacht de `WHERE`-clausule.
- **Let op de grootte van je warehouse-compute, niet alleen op querycomplexiteit.** Een klein warehouse dat een complexe query draait, wacht en staat in de rij; een enorm warehouse dat een triviale query draait, verbrandt credits zonder enig voordeel. De juiste compute-grootte kiezen voor de workload is een grotere kostenhefboom dan bijna elke query-optimalisatie — zie het hoofdstuk Snowflake voor concrete sizing-richtlijnen.
- **Incrementeel verslaat volledige refresh op schaal.** Een volledige fact table elke nacht herberekenen is prima bij 10 miljoen rijen en rampzalig bij 10 miljard. Incrementele modellen (alleen nieuwe/gewijzigde data verwerken) zijn de meest impactvolle performanceverbetering die beschikbaar is in een volwassen dbt-project.
- **Vermijd `SELECT *` in productiemodellen.** Naast leesbaarheid breekt het stilletjes downstream-modellen wanneer een bovenliggende tabel een kolom krijgt of verliest, en het voorkomt dat het warehouse ongebruikte kolommen kan overslaan in kolomgeoriënteerde opslagformaten.

## Best Practices

- **Maak pipelines idempotent.** Dezelfde job tweemaal draaien mag nooit duplicaten of onjuiste data opleveren. De `MERGE`- en `ROW_NUMBER()`-patronen hierboven zijn de twee meest gebruikte manieren om dit af te dwingen.
- **Gebruik een watermark, geen volledige herlaad**, voor incrementele bronnen — een hele tabel dagelijks herladen wordt duur en traag naarmate data groeit, en wordt uiteindelijk fysiek onmogelijk binnen je batch-window.
- **Scheid ruwe van gemodelleerde data.** Transformeer nooit data op de plek zelf; bewaar altijd een ongewijzigde bronze-kopie zodat je downstream-lagen kunt herbouwen als transformatielogica verandert of er maanden later een bug wordt ontdekt.
- **Versiebeheer je transformatielogica.** SQL en pipeline-code horen in Git, gereviewd als elke andere code — zie Git & Branching Strategies.
- **Alerteer op afwezigheid van data, niet alleen op pipeline-fouten.** Een pipeline die "slaagt" maar nul rijen laadt omdat een bovenstroomse API stilletjes zijn responsformaat wijzigde, is een faalmodus die monitoring op alleen jobstatus nooit zal opvangen — zie Monitoring & Observability.
- **Documenteer aannames, niet alleen code.** "Deze tabel gaat uit van één rij per order" is waardevoller voor de volgende engineer dan een comment die uitlegt wat een `JOIN` doet.

## Veelgemaakte Fouten

- **Alles zelf bouwen.** Een eigen ingestion-framework schrijven terwijl een managed tool (Fivetran, Airbyte) 90% van de behoefte dekt in een fractie van de tijd. Custom ingestion-code is een onderhoudslast die zichzelf zelden terugverdient, en het is de meest voorkomende vorm van verspilde engineering-inspanning die ik zie bij nieuwe dataeams.
- **Geen schema-contracten.** Bronschema's als vast beschouwen, waardoor pipelines breken — of erger, stilletjes verkeerd laden — wanneer een bronteam zonder waarschuwing een kolom toevoegt, hernoemt of van type verandert.
- **Tests overslaan.** Transformatie-SQL uitrollen zonder validatie dat rijaantallen, uniekheid of referentiële integriteit kloppen — zie Data Quality & Testing.
- **"Het draaide" verwarren met "het werkte".** Een pipeline kan succesvol afronden en toch foute cijfers opleveren. Datavalidatie is een aparte zorg naast orchestratiestatus, en die twee door elkaar halen is precies hoe foute cijfers in een directiedashboard belanden.
- **Voortijdig streamen.** Een Kafka-gebaseerde streamingpipeline bouwen voor een dashboard dat toch maar één keer per dag wordt ververst. Match de architectuur met de daadwerkelijke latency-eis, niet met wat technisch indrukwekkend is.
- **Geen kostenzicht.** Pipelines onbeperkt op te grote compute laten draaien omdat niemand naar de warehouse-rekening kijkt totdat finance vraagt waarom de kosten zijn verdrievoudigd.

## Interviewvragen

**"Loop met me door hoe je een pipeline zou ontwerpen om data van een productiedatabase naar een warehouse te laden."** Let op: incrementele extractie met een watermark, scheiding van ruwe/bronze en gemodelleerde/silver data, idempotentie via MERGE of dedup-logica, en hoe ze schemawijzigingen zouden afhandelen zonder downstream-tabellen te breken.

**"Wat is het verschil tussen ETL en ELT, en waarom maakt het uit?"** Let op: het begrip dat ELT eerst ruwe data laadt en transformeert met de compute van het warehouse, wat verklaart waarom het domineert op moderne cloudplatformen — volledige uitleg in ETL vs ELT.

**"Hoe zou je detecteren dat een pipeline onvolledige data heeft geladen, ook al is de job zelf geslaagd?"** Let op: controles op afwijkende rijaantallen, freshness-checks, het vergelijken van het daadwerkelijke volume met een verwachte bandbreedte — niet alleen "de logs checken".

**"Wat maakt een pipeline idempotent, en waarom maakt dat uit?"** Let op: een duidelijk voorbeeld (zoals het `MERGE`-patroon hierboven) en het begrip dat retries en backfills onvermijdelijk zijn in productie, waardoor niet-idempotente pipelines uiteindelijk data corrumperen.

**"Wanneer kies je voor batch processing boven streaming, en andersom?"** Let op: afwegingen in kosten en complexiteit, de herkenning dat de meeste als "real-time" omschreven businessvereisten in werkelijkheid minuten latency tolereren, en specifieke streaming use cases (fraudedetectie, live operationele dashboards) waar het echt nodig is.

**"Hoe ga je om met een bronsysteem dat zonder waarschuwing zijn schema wijzigt?"** Let op: detectie van schema drift, defensief parsen (niet aannemen dat een kolom bestaat), alerteren in plaats van stilletjes falen, en idealiter een data contract of afspraak met het bronteam.

**"Wat is in jouw eigen woorden het verschil tussen een data engineer en een data analist?"** Let op: een heldere articulatie dat engineers de systemen bouwen en onderhouden die data betrouwbaar en beschikbaar maken, terwijl analisten die data gebruiken om businessvragen te beantwoorden — niet een vaag "engineers schrijven meer code".

## Relevante Documentatie

- Snowflake Documentation — officiële referentie voor warehouse-architectuur en SQL.
- Databricks Documentation — lakehouse-architectuur, Delta Lake en Spark.
- dbt Labs Documentation — patronen en best practices voor de transformatielaag.
- Microsoft Fabric Documentation — Microsofts unified analytics-platform.
- Apache Airflow Documentation — orchestratieconcepten en DAG-ontwerp.

## Samenvatting

Data engineering is de praktijk van het bouwen van betrouwbare systemen die data verplaatsen en hervormen — geen vage verzamelnaam voor "data-infrastructuur", en anders dan de rol van ETL Developer waaruit het is voortgekomen. De moderne stack scheidt ingestion, storage, transformatie, orchestratie en serving in aparte lagen, doorgaans volgens een ELT-patroon waarbij transformatie plaatsvindt binnen het warehouse ná het laden. Batch blijft de verstandige standaard; streaming is een bewuste keuze voor een specifieke, gekwantificeerde latency-eis, geen standaardarchitectuur. Het bronze/silver/gold-patroon uit de pipeline-walkthrough van dit hoofdstuk — extract met een watermark, idempotent laden, dedupliceren en opschonen, dan modelleren en aggregeren — is het fundament voor bijna alles wat verderop in dit handbook aan bod komt, te beginnen met de SQL-patronen die het mogelijk maken, in het volgende hoofdstuk.

HOOFDSTUK 2

# SQL voor Data Engineers

## Introductie

Elke vacaturetekst voor data engineering noemt Python, Spark en steeds vaker ervaring met een cloudplatform — maar de vaardigheid die in letterlijk elke vacature terugkomt, en die het verschil maakt tussen engineers die snel kunnen bouwen en engineers die dat niet kunnen, is SQL. Niet "ik kan een `SELECT` met een `JOIN` schrijven"-SQL, maar SQL die deduplicatie correct afhandelt, data idempotent laadt, en niet stilletjes je omzetcijfers verdubbelt door een one-to-many join die je over het hoofd zag.

Dit hoofdstuk is geen SQL-syntaxcursus — als je al een basisquery kunt schrijven, behandelt dit de patronen die voortdurend terugkomen in echte pipelines en die de meeste SQL-cursussen volledig overslaan: window functions voor deduplicatie, `MERGE` voor idempotent laden, en het join- en query-executiegedrag dat subtiele, lastig te debuggen bugs veroorzaakt in productie.

## Query Execution Order

Vóór de patronen eerst één fundamenteel concept dat de helft van de "waarom werkt dit niet"-verwarring bij beginners verklaart: SQL wordt in één volgorde geschreven, maar in een andere volgorde *uitgevoerd*.

`FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT`

Dit is waarom je geen kolomalias uit `SELECT` kunt gebruiken in een `WHERE`-clausule — op het moment dat `WHERE` wordt geëvalueerd, is `SELECT` nog niet uitgevoerd:

```
-- Dit faalt: 'total' bestaat nog niet wanneer WHERE wordt geëvalueerd
SELECT amount * quantity AS total
FROM order_items
WHERE total > 100;   -- FOUT bij de meeste databases

-- Dit werkt wel: HAVING draait na SELECT/GROUP BY
SELECT amount * quantity AS total
FROM order_items
GROUP BY amount, quantity
HAVING amount * quantity > 100;
```

Het verklaart ook waarom filteren in `WHERE` doorgaans goedkoper is dan filteren in `HAVING`: `WHERE` reduceert het aantal rijen *voordat* er wordt gegroepeerd en geaggregeerd, terwijl `HAVING` filtert *nadat* de (mogelijk kostbare) aggregatie al over de volledige set is berekend. Deze execution order begrijpen is het nuttigste stukje SQL-theorie dat er is voor het schrijven van zowel correcte als snelle queries.

## Window Functions in Detail

Window functions zijn de belangrijkste SQL-feature specifiek voor data engineering, omdat je er iets mee kunt berekenen *over een set gerelateerde rijen* zonder die rijen samen te voegen tot één, zoals `GROUP BY` doet. De syntax is altijd een functie gevolgd door `OVER (PARTITION BY ... ORDER BY ...)`.

**`ROW_NUMBER()`** kent een uniek, opeenvolgend nummer toe binnen elke partitie — de ruggengraat van deduplicatie, hieronder uitgebreid behandeld.

**`RANK()` en `DENSE_RANK()`** lijken erop, maar gaan anders om met gelijkspel: `RANK()` laat gaten vallen na een gelijkspel (1, 2, 2, 4), `DENSE_RANK()` niet (1, 2, 2, 3). Gebruik `DENSE_RANK()` voor dingen als "top 3 categorieën op omzet" waarbij je exact 3 unieke rangwaarden wilt, ook bij gelijkspel.

**`LAG()` en `LEAD()`** kijken naar de vorige of volgende rij binnen een partitie — essentieel voor tijdreeksvergelijkingen zoals "omzet vs. vorige dag":

```
SELECT
    order_date,
    daily_revenue,
    LAG(daily_revenue) OVER (ORDER BY order_date) AS prev_day_revenue,
    daily_revenue - LAG(daily_revenue) OVER (ORDER BY order_date) AS day_over_day_change
FROM gold.daily_revenue
ORDER BY order_date;
```

**Lopende totalen en voortschrijdende gemiddelden** gebruiken aggregatiefuncties als window function door een frame-clausule toe te voegen:

```
SELECT
    order_date,
    daily_revenue,
    SUM(daily_revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total,
    AVG(daily_revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS trailing_7day_avg
FROM gold.daily_revenue;
```

## Deduplicatiepatronen in Detail

Deduplicatie is een van de meest voorkomende bewerkingen in een data pipeline, en er zijn drie manieren om het te doen met betekenisvol verschillend gedrag en performance.

**`ROW_NUMBER()` + filter** — de meest expliciete en controleerbare aanpak, en degene waar je standaard voor moet kiezen:

```
SELECT * FROM (
    SELECT *,
        ROW_NUMBER() OVER (
            PARTITION BY order_id
            ORDER BY updated_at DESC, ingested_at DESC  -- tie-breaker is belangrijk!
        ) AS rn
    FROM raw.orders
)
WHERE rn = 1;
```

Let op de tweekolommige `ORDER BY` in het window — `updated_at DESC` alleen is niet genoeg als twee versies van dezelfde rij een *identieke* timestamp hebben (wat vaker voorkomt dan je zou denken bij batch-geladen data). Zonder een deterministische tie-breaker kiest `ROW_NUMBER()` willekeurig, en welke rij "wint" kan verschillen tussen query-runs — een subtiele bron van niet-reproduceerbare pipeline-output die achteraf lastig te debuggen is.

**`SELECT DISTINCT`** verwijdert volledig identieke rijen, maar doet niets als rijen in ook maar één kolom verschillen (zoals een `loaded_at`-timestamp die tijdens ingestion is toegevoegd, wat extreem vaak voorkomt). Het is verleidelijk omdat het kort is, maar het is het verkeerde gereedschap voor "bewaar de laatste versie van elk record" — het kan alleen exacte duplicaten verwijderen, geen conflicterende versies.

**`QUALIFY`** (ondersteund in Snowflake, Databricks SQL en BigQuery) laat je direct filteren op een window function, zonder de subquery-wrapper:

```
SELECT *
FROM raw.orders
QUALIFY ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY updated_at DESC) = 1;
```

Functioneel identiek aan het `ROW_NUMBER()` + subquery-patroon, maar aanzienlijk leesbaarder zodra je eraan gewend bent. Het bestaat niet in standaard ANSI SQL, PostgreSQL of SQL Server, dus het subquery-patroon blijft de portabele keuze wanneer je SQL over meerdere dialecten heen moet werken — relevant als je dbt-modellen schrijft die meerdere warehouse-adapters moeten ondersteunen.

## Idempotente, Incrementele SQL Schrijven

Een pipeline die veilig opnieuw gedraaid kan worden — na een fout, voor een backfill, of gewoon omdat iemand hem per ongeluk twee keer triggerde — moet idempotent zijn: N keer draaien levert hetzelfde resultaat op als één keer draaien. `MERGE` (in sommige dialecten ook `UPSERT` genoemd) is hiervoor het belangrijkste gereedschap:

```
MERGE INTO silver.orders AS tgt
USING staging.orders AS src
ON tgt.order_id = src.order_id
WHEN MATCHED AND src.updated_at > tgt.updated_at THEN
    UPDATE SET amount = src.amount, updated_at = src.updated_at, status = src.status
WHEN NOT MATCHED THEN
    INSERT (order_id, amount, updated_at, status)
    VALUES (src.order_id, src.amount, src.updated_at, src.status);
```

Twee details die ertoe doen en die beginners vaak missen. Ten eerste de `AND src.updated_at > tgt.updated_at`-guard op de `UPDATE`-tak — zonder deze zou een herhaalde merge met *verouderde* staging-data (bijvoorbeeld een opnieuw getriggerde job die uit een bron leest die niet is veranderd) een nieuwere doelrij overschrijven met oudere data. Ten tweede **te laat binnenkomende data**: als je watermark-logica extraheert op basis van `updated_at`, kan een record dat *na* het verstrijken van je extractievenster wordt gebackfilled of gecorrigeerd volledig gemist worden, tenzij je extractiequery een lookback-buffer bevat (bijvoorbeeld altijd de laatste 3 dagen opnieuw ophalen, niet alleen "sinds de watermark") om late correcties op te vangen.

MySQL gebruikt een andere syntax voor hetzelfde idee — `INSERT ... ON DUPLICATE KEY UPDATE` — en PostgreSQL gebruikt `INSERT ... ON CONFLICT ... DO UPDATE`. Het concept is universeel, ook waar de syntax dat niet is; de SQL Generator-tool regelt de dialectvertaling voor je.

## CTE's versus Subqueries versus Temp Tables versus Views

Alle vier kunnen dezelfde logische transformatie uitdrukken, maar ze gedragen zich verschillend genoeg dat de keuze ertoe doet:

- **CTE's** (`WITH x AS (...)`) zijn de meest leesbare manier om een complexe query op te breken in benoemde, opeenvolgende stappen. In de meeste moderne warehouses (Snowflake, Databricks SQL, PostgreSQL 12+) kan de optimizer een CTE inlinen of materialiseren afhankelijk van hoe hij wordt gebruikt — je hoeft je meestal geen zorgen te maken over performance, tenzij dezelfde CTE veel keren wordt aangeroepen, in welk geval sommige engines hem elke keer opnieuw uitvoeren in plaats van het resultaat te cachen.
- **Subqueries** zijn logisch equivalent aan CTE's, maar schaden de leesbaarheid zodra ze meer dan één of twee niveaus diep genest worden. Geef bij alles wat niet-triviaal is de voorkeur aan CTE's.
- **Temp tables** materialiseren resultaten fysiek, wat nuttig is wanneer hetzelfde tussenresultaat duur is om te berekenen en vaak wordt hergebruikt binnen een sessie — de afweging is dat je de schrijfkosten vooraf betaalt.
- **Views** slaan helemaal geen data op; het is een opgeslagen querydefinitie, die elke keer opnieuw wordt uitgevoerd bij bevraging. Handig als stabiele abstractie over veranderende brontabellen, maar geen performance-optimalisatie — als de onderliggende query traag is, is de view precies zo traag, elke keer weer. Dit onderscheid, en wanneer je in plaats daarvan naar een Snowflake Dynamic Table grijpt, wordt verder behandeld in het hoofdstuk Snowflake.

## Joins en hun Performance-implicaties

Joins zijn waar correctheidsbugs zich het vaakst verstoppen, en de fan-out-valkuil is verreweg de meest voorkomende. Stel je voor dat je een `orders`-tabel (één rij per order) joint met een `order_items`-tabel (meerdere rijen per order — één per orderregel) en dan probeert de omzet te sommeren:

```
-- FOUT: dit vermenigvuldigt order-niveau totalen met het aantal items per order
SELECT
    o.order_id,
    o.order_total,          -- een kolom op order-niveau
    SUM(o.order_total) AS revenue   -- uitgewaaierd door de join, enorm overschat
FROM orders o
```

```
JOIN order_items oi ON oi.order_id = o.order_id
GROUP BY o.order_id, o.order_total;
```

Omdat de join één rij per orderregel oplevert, wordt `o.order_total` herhaald voor elk item, en het sommeren daarvan overschat de omzet enorm. De oplossing is om `order_items` te aggregeren tot één rij per order *vóórdat* je joint, of om het order-niveau-totaal te selecteren uit een bron die al op de juiste granulariteit staat:

```
-- CORRECT: eerst aggregeren naar order-granulariteit, dan pas joinen indien nodig
SELECT order_id, SUM(order_total) AS revenue
FROM orders
GROUP BY order_id;
```

Deze "fan-out"-bug is verraderlijk makkelijk te introduceren in een grotere query met meerdere joins, en geeft zelden een foutmelding — hij produceert gewoon stilletjes een getal dat te hoog is. Wees altijd expliciet, in elk geval voor jezelf, over op welke granulariteit elke tabel in een join staat vóórdat je aggregeert.

Over het type join: kies standaard voor `INNER JOIN`, tenzij je specifiek niet-gematchte rijen van één kant nodig hebt, in welk geval je bewust een `LEFT JOIN` gebruikt. Een per ongeluk gebruikte `LEFT JOIN` waar een `INNER JOIN` bedoeld was, introduceert stilletjes `NULL`-rijen die downstream-aggregaties op subtiele wijze kunnen breken.

## Veelvoorkomende Performance-valkuilen

- **Een gefilterde kolom in een functie wikkelen.** `WHERE UPPER(email) = 'X@Y.COM'` voorkomt dat de query-engine efficiënt kan pruning op `email`, omdat hij de functie op elke rij moet evalueren voordat hij kan filteren. Normaliseer data bij het schrijven in plaats van te leunen op runtime-functies bij het filteren.
- **Impliciete type casts.** Een `VARCHAR`-kolom vergelijken met een numerieke literal dwingt een impliciete cast af op elke rij, wat — afhankelijk van de engine — stilletjes partition pruning of indexering kan uitschakelen. Cast expliciet en consistent in plaats van te vertrouwen op de impliciete coercion-regels van de engine.
- **`SELECT DISTINCT` als vervanging voor correcte deduplicatie.** Zoals hierboven behandeld, verwijdert het alleen exact identieke rijen — het gebruiken om data "op te schonen" die eigenlijk conflicterende versies bevat, levert onjuiste resultaten op, niet alleen trage.
- **Ontbrekende partition pruning.** Filteren op een afgeleide expressie (`WHERE YEAR(order_date) = 2026`) in plaats van op de ruwe gepartitioneerde kolom (`WHERE order_date >= '2026-01-01' AND order_date < '2027-01-01'`) kan voorkomen dat de engine irrelevante partities overslaat, wat een volledige scan afdwingt.
- **Per ongeluk cross joins.** Een kommagescheiden `FROM a, b` zonder een echte `JOIN ... ON`-voorwaarde produceert stilletjes een cartesisch product — elke rij van `a` gekoppeld aan elke rij van `b`. Bij kleine tabellen valt dit misschien niet op; bij grote kan het een warehouse tot stilstand brengen.

## Best Practices

- **Gebruik altijd een expliciete, deterministische tie-breaker** in de `ROW_NUMBER()`-ordening voor deduplicatie — neem nooit aan dat timestamps uniek zijn.
- **Bescherm `MERGE`-updates met een freshness-check** (`src.updated_at > tgt.updated_at`) zodat een herhaalde run met verouderde data nooit goede data kan overschrijven met oude data.
- **Aggregeer naar de juiste granulariteit vóór het joinen**, niet erna, om fan-out-bugs te voorkomen.
- **Geef standaard de voorkeur aan `INNER JOIN`**; gebruik `LEFT JOIN` bewust en controleer achteraf op onverwachte `NULL`s.
- **Filter op ruwe kolommen, niet op afgeleide expressies**, om partition pruning en indexgebruik te behouden.
- **Gebruik CTE's om tussenstappen te benoemen** — een query die leest als een reeks benoemde stappen is aanzienlijk makkelijker voor de volgende persoon (vaak jijzelf, over zes maanden) om te debuggen.

## Veelgemaakte Fouten

- **`SELECT DISTINCT` gebruiken om duplicaten "op te lossen"** zonder te begrijpen of de duplicaten exacte kopieën zijn of conflicterende versies van hetzelfde record.
- **De tie-breaker vergeten** bij een `ROW_NUMBER()`-dedup, wat leidt tot niet-deterministische output die verschilt tussen runs.
- **Joinen vóór het aggregeren** en stilletjes een metric uitwaaieren, wat getallen oplevert die fout zijn maar niet duidelijk fout — de query draait prima en geeft plausibel ogende resultaten terug.
- **`MERGE` als automatisch veilig beschouwen** zonder freshness-guard, waardoor verouderde herhaalde runs data stilletjes kunnen terugdraaien.
- **Subqueries nesten in plaats van CTE's gebruiken**, wat SQL oplevert die technisch correct is maar bijna onmogelijk voor een collega om te reviewen.

## Performance Tips

- **Duw filters zo vroeg mogelijk** — in CTE's, niet alleen in de buitenste query — zodat downstream joins en aggregaties op minder data hoeven te werken.
- **Materialiseer dure, hergebruikte tussenresultaten** als temp tables in plaats van dezelfde CTE-logica meerdere keren in één query te herhalen.
- **Match je `ORDER BY`/`PARTITION BY`-kolommen met de clustering- of partitioneringssleutel van je tabel** waar mogelijk, zodat window functions kunnen profiteren van al gesorteerde data in plaats van een volledige sortering te triggeren.
- **Vermijd onnodige `ORDER BY` in subqueries** — sorteervolgorde blijft toch zelden behouden na een buitenste aggregatie, dus vroeg sorteren verspilt compute zonder voordeel.
- **Batch je `MERGE`-operaties** — één rij tegelijk mergen in een loop is aanzienlijk trager dan een gestagede batch in één enkele statement mergen.

## Interviewvragen

**"Schrijf een query om de meest recente order per klant op te halen."** Let op: `ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC)` met een filter op `rn = 1`, en idealiter een vermelding van tie-breaking.

**"Wat is het verschil tussen `RANK()`, `DENSE_RANK()` en `ROW_NUMBER()`?"** Let op: `ROW_NUMBER()` levert altijd unieke, opeenvolgende nummers op, ook bij gelijkspel; `RANK()` laat gaten vallen na gelijkspel; `DENSE_RANK()` niet.

**"Hoe zou je een tabel dedupliceren waar hetzelfde record meerdere keren kan voorkomen met licht verschillende waarden?"** Let op: het besef dat `SELECT DISTINCT` hier niet werkt, en een `ROW_NUMBER()`-gebaseerde aanpak met een duidelijke "welke versie wint"-regel.

**"Wat is er mis met deze query?"** (toon een fan-out-voorbeeld zoals hierboven) Let op: het herkennen dat de join een waarde op order-niveau vermenigvuldigt over orderregels vóór het sommeren, en een oplossing die eerst naar de juiste granulariteit aggregeert.

**"Leg uit wat er gebeurt als je een `MERGE`-statement twee keer draait met dezelfde input."** Let op: het begrip dat een goed geschreven `MERGE` idempotent is — de tweede run zou een no-op moeten zijn als er niets is veranderd — en het besef dat zonder freshness-guard verouderde herhaalde runs regressies kunnen veroorzaken.

**"Wat is de SQL execution order, en waarom maakt het uit?"** Let op: `FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT`, en een concreet voorbeeld van waarom een `SELECT`-alias niet in `WHERE` gebruikt kan worden.

**"Wanneer zou je een CTE gebruiken versus een temp table?"** Let op: CTE's voor leesbaarheid en eenvoudige sequencing; temp tables wanneer een duur tussenresultaat gematerialiseerd en meerdere keren hergebruikt moet worden.

## Relevante Documentatie

- Snowflake Window Functions — officiële syntaxreferentie, inclusief `QUALIFY`.
- PostgreSQL Window Functions Tutorial — het ANSI-standaardgedrag dat de meeste engines volgen.
- Databricks SQL Reference — `MERGE INTO` en window function-syntax op het lakehouse.
- dbt Labs: SQL Style Guide — CTE-naamgeving en structureringsconventies die in de industrie worden gebruikt.

## Samenvatting

SQL voor data engineering is een andere vaardigheid dan SQL voor analytics, ook al overlapt de syntax bijna volledig. De patronen die ertoe doen zijn window functions voor deduplicatie en tijdreekslogica, `MERGE` voor idempotent laden met de juiste freshness-guards, zorgvuldige aandacht voor join-granulariteit om fan-out-bugs te voorkomen, en begrip van query execution order dat verklaart waarom bepaalde dingen wel of niet werken. Niets hiervan vereist exotische syntax — het vereist weten welke van de "voor de hand liggende" manieren om een query te schrijven stilletjes foute getallen oplevert onder realistische omstandigheden zoals retries, te laat binnenkomende data en one-to-many-relaties. Het volgende hoofdstuk bouwt direct voort op deze patronen om te laten zien hoe ze passen in een volledig dataplatform-ontwerp.

HOOFDSTUK 3

# Data Warehousing Fundamenten

## Introductie

In Introduction to Data Engineering zagen we waarom een productiedatabase niet geschikt is om rechtstreeks op te rapporteren — een zware aggregatiequery vertraagt de applicatie die hij bedient. Dit hoofdstuk gaat een niveau dieper: wat maakt een data warehouse dan wél geschikt voor die queries? Het antwoord zit niet in "meer geheugen" of "een snellere server" — het zit in fundamenteel andere ontwerpkeuzes op het niveau van hoe data fysiek wordt opgeslagen en hoe queries worden uitgevoerd.

Dit is geen historisch overzicht om te doorstaan voordat je bij de "echte" tools mag; het is de basis waarop Snowflake, Databricks SQL en elk ander modern platform is gebouwd. Zonder dit fundament blijft "warehouse-performance" een black box die je alleen via trial-and-error kunt beïnvloeden.

## OLTP versus OLAP: het Fundamentele Onderscheid

Elke discussie over data warehousing begint bij dit onderscheid, omdat het de reden ís dat warehouses bestaan.

**OLTP (Online Transaction Processing)** is wat een productiedatabase doet: veel korte, gelijktijdige transacties, elk gericht op een klein aantal rijen. Denk aan "voeg deze order toe", "werk de voorraad van dit product bij", "haal de gegevens van deze ene klant op". De database is geoptimaliseerd voor lees-/schrijfsnelheid op individuele rijen, met sterke consistentiegaranties (ACID) omdat een halve transactie — bijvoorbeeld geld afschrijven zonder het bij de ontvanger bij te schrijven — onacceptabel is.

**OLAP (Online Analytical Processing)** is wat een warehouse doet: weinig, maar zware queries die grote hoeveelheden rijen doorzoeken en aggregeren. Denk aan "wat was de totale omzet per regio per maand, afgelopen twee jaar". Deze query raakt mogelijk miljoenen rijen, maar wijzigt niets — het is puur lezen en aggregeren. De prioriteit is niet transactiesnelheid maar doorvoersnelheid over grote datavolumes.

Het probleem ontstaat wanneer je OLAP-werk op een OLTP-systeem probeert te doen: de fysieke opslagstructuur die transacties snel maakt (rij-georiënteerde opslag, zie hieronder) is exact de verkeerde structuur voor het scannen van miljoenen rijen om er twee kolommen uit te aggregeren.

Concreet: op een OLTP-database met tien miljoen orderregels en twintig kolommen per rij, moet een aggregatiequery die alleen `category` en `amount` nodig heeft, alsnog alle twintig kolommen van elke rij van schijf lezen omdat die fysiek aan elkaar vastzitten — al snel een veelvoud van de data die je daadwerkelijk nodig hebt. Op honderdduizenden gelijktijdige applicatietransacties komt die extra I/O-belasting bovenop, met merkbare vertraging voor elke gebruiker die op dat moment de applicatie gebruikt. Dat is het probleem dat een warehouse structureel oplost, niet door "meer hardware", maar door de opslagstructuur zelf anders in te richten — zie hieronder.

## Columnar Storage: de Technische Kern van een Warehouse

Dit is het belangrijkste technische concept in dit hoofdstuk, en het verklaart bijna alles over waarom warehouses zo anders presteren dan productiedatabases.

Een traditionele (OLTP) database slaat data **rij-georiënteerd** op: alle kolommen van rij 1 staan fysiek bij elkaar op schijf, dan alle kolommen van rij 2, enzovoort. Dit is ideaal wanneer je "geef mij alle gegevens van klant X" opvraagt — één schijftoegang levert de complete rij. Maar wanneer je "som de kolom `amount` op over 10 miljoen rijen" vraagt, moet het systeem elke rij volledig inlezen, ook al gebruik je maar één van de twintig kolommen uit die rij.

Een data warehouse slaat data **kolom-georiënteerd** op: alle waarden van kolom `amount` staan fysiek bij elkaar, gescheiden van alle waarden van kolom `customer_id`, enzovoort. Voor de aggregatiequery hierboven leest het systeem nu alleen de fysieke blokken die bij

`amount` horen — de andere negentien kolommen worden niet eens van schijf gehaald. Bij een brede tabel met tientallen kolommen is dit het verschil tussen een query die seconden duurt en een query die minuten duurt.

Columnar storage heeft nog een tweede, minder besproken voordeel: **compressie**. Omdat alle waarden in een kolom hetzelfde datatype hebben en vaak weinig variatie vertonen (denk aan een `status`-kolom met maar vier mogelijke waarden, of een `country_code`-kolom), comprimeren kolomsgewijze blokken veel effectiever dan rijsgewijze blokken — vaak met factor 5 tot 10. Minder data op schijf betekent minder data die van schijf naar geheugen moet, wat queries verder versnelt bovenop het voordeel van alleen-de-benodigde-kolommen-lezen.

```
-- Deze query illustreert het verschil:
-- Op een rij-georiënteerde OLTP-database moet elke rij volledig gelezen
-- worden, ook al gebruiken we maar 2 van de 20 kolommen van 'orders'.
-- Op een kolom-georiënteerd warehouse worden alleen 'category' en 'amount'
-- fysiek van schijf gehaald — de overige 18 kolommen raakt de engine niet aan.
SELECT category, SUM(amount) AS revenue
FROM orders
GROUP BY category;
```

## MPP: Massively Parallel Processing

Columnar storage verklaart waarom één query minder data hoeft te lezen; **MPP** verklaart waarom moderne warehouses die query ook nog eens over meerdere machines tegelijk kunnen uitvoeren. In een MPP-architectuur wordt een tabel logisch verdeeld over meerdere compute-nodes, en wanneer je een query uitvoert, verwerkt elke node zijn eigen stuk van de data parallel, waarna de resultaten worden samengevoegd.

Snowflake, BigQuery, Redshift en Databricks SQL zijn allemaal, in de kern, MPP-systemen — het verschil zit vooral in hoe expliciet jij als engineer die parallellisatie moet aansturen. Snowflake regelt dit vrijwel volledig automatisch via micro-partities; bij Databricks/Spark heb je meer directe controle (en verantwoordelijkheid) over partitionering. Dit is de reden dat het opschalen van een query op deze platformen vaak zo simpel is als "een groter warehouse kiezen" — je voegt letterlijk meer parallelle rekenkracht toe, in plaats van de query zelf te herschrijven.

## Data Warehouse versus Data Lake versus Lakehouse

Deze drie termen worden vaak door elkaar gebruikt, maar verwijzen naar verschillende ontwerpfilosofieën:

**Data warehouse** — gestructureerde data, schema wordt afgedwongen bij het schrijven (*schema-on-write*), geoptimaliseerd voor SQL-analytics. Klassiek voorbeeld: Snowflake, Redshift, on-premises Teradata.

**Data lake** — ruwe, vaak ongestructureerde of semi-gestructureerde data (JSON, logs, afbeeldingen, Parquet-bestanden), opgeslagen in goedkope objectopslag (S3, ADLS), zonder dat een schema wordt afgedwongen bij het schrijven — het schema wordt pas bepaald op het moment dat je de data leest (*schema-on-read*). Flexibeler, maar zonder discipline eindigt een data lake al snel als een "data swamp": een berg bestanden zonder betrouwbare structuur.

**Lakehouse** — de poging om het beste van beide te combineren: data lake-achtige opslag (goedkoop, flexibel, in objectopslag) met warehouse-achtige garanties bovenop (ACID-transacties, schema-afdwinging, indexering) via een tabelformaat als Delta Lake of Iceberg. Databricks is het meest bekende voorbeeld van dit patroon; zie Delta Lake voor de technische details van hoe dat precies werkt.

Voor de praktijk van dit handbook is het onderscheid vooral relevant omdat het bepaalt *waar* je bronze-laag (zie Medallion Architecture) leeft: in een puur warehouse-platform (Snowflake) landt zelfs ruwe bronze-data al in warehousetabellen; in een lakehouse-platform (Databricks) landt bronze vaak eerst als bestanden in objectopslag voordat het als Delta-tabel wordt geregistreerd.

## De Klassieke Architectuur: Staging, Warehouse, Marts

Voordat "medallion architecture" de gangbare term werd, gebruikte de warehousing-wereld al decennialang een vergelijkbaar drielagen-model, en het is de moeite waard de link te zien:

**Staging** — een tijdelijk landingsgebied voor ruwe, ongewijzigde brondata, functioneel bijna identiek aan de bronze-laag. Data hier wordt regelmatig volledig ververst of zelfs weggegooid na verwerking.

**Warehouse (of "integration layer")** — schone, gededupliceerde, geïntegreerde data over bronnen heen, in een genormaliseerde of licht gedenormaliseerde vorm. Functioneel vergelijkbaar met silver.

**Data marts** — kleine, sterk gedenormaliseerde, onderwerpgerichte subsets van het warehouse, toegesneden op één afdeling of use case (een "sales mart", een "finance mart"). Functioneel vergelijkbaar met gold, met als verschil dat marts historisch vaak per-afdeling gescheiden werden gebouwd, terwijl moderne gold-tabellen vaker organisatiebreed herbruikbaar worden gehouden.

De terminologie is verschoven, maar het onderliggende inzicht niet: ruwe data, geïntegreerde data, en presentatieklare data zijn drie verschillende verantwoordelijkheden die drie verschillende lagen verdienen — of je ze nu staging/warehouse/marts noemt of bronze/silver/gold.

## Praktisch Voorbeeld: een Warehouse Opzetten voor Verkoopdata

Stel je bouwt een warehouse voor een retailbedrijf met verkoopdata uit meerdere winkels. De staging-laag ontvangt dagelijks ruwe CSV-exports per winkel:

```
-- staging.sales_raw: exact zoals aangeleverd, geen aannames
CREATE TABLE staging.sales_raw (
    store_id      VARCHAR,
    sale_date     VARCHAR,   -- bewust nog VARCHAR: bronformaat is inconsistent
    product_sku   VARCHAR,
    quantity      VARCHAR,
    unit_price    VARCHAR,
    loaded_at     TIMESTAMP DEFAULT CURRENT_TIMESTAMP()
);
```

De integratielaag (silver) dwingt pas hier types en consistentie af — bewust niet eerder, omdat je in staging nooit data mag verliezen door een te vroege, te strikte cast:

```
CREATE OR REPLACE TABLE warehouse.sales AS
SELECT
    store_id,
    TRY_CAST(sale_date AS DATE) AS sale_date,
    product_sku,
    TRY_CAST(quantity AS INTEGER) AS quantity,
    TRY_CAST(unit_price AS NUMBER(10,2)) AS unit_price
FROM staging.sales_raw
WHERE TRY_CAST(sale_date AS DATE) IS NOT NULL;   -- rijen met onbruikbare datums expliciet eruit, niet stilletjes NULL
```

De `TRY_CAST` in plaats van `CAST` is bewust: een gewone `CAST` op een enkele foutieve waarde laat de hele query falen; `TRY_CAST` geeft `NULL` terug voor die ene rij en laat de rest van de batch gewoon doorgaan — waarna de `WHERE`-clausule expliciet beslist wat er met die onbruikbare rijen gebeurt, in plaats van dat de pipeline om 3 uur 's nachts crasht op één rommelige rij.

Een data mart (gold) voor het salesteam aggregeert dit vervolgens tot iets direct bruikbaars:

```
CREATE OR REPLACE TABLE marts.sales_by_store_day AS
SELECT
    store_id,
    sale_date,
    SUM(quantity * unit_price) AS revenue,
    SUM(quantity) AS units_sold
FROM warehouse.sales
GROUP BY store_id, sale_date;
```

## Slowly Changing Dimensions: een Preview

Eén vraag duikt onvermijdelijk op zodra je warehouse langer dan een paar weken meegaat: wat doe je als een attribuut van een entiteit verandert — een klant verhuist, een product wisselt van categorie? Overschrijf je de oude waarde, of bewaar je de historie? Dit is het domein van **slowly changing dimensions (SCD)**, een van de kernconcepten van dimensioneel modelleren. We behandelen dit niet volledig in dit hoofdstuk — het krijgt de aandacht die het verdient in Star Schema & Dimensional Modeling — maar het is belangrijk om nu al te weten dat "gewoon de rij updaten" een bewuste modelleerkeuze is (SCD Type 1), niet de enige optie, en dat die keuze historische rapportage onherstelbaar kan beïnvloeden als je hem niet vooraf overweegt.

## Best Practices

- **Cast nooit te vroeg in staging.** Bewaar ruwe waarden als tekst totdat de integratielaag ze expliciet en foutentolerant (`TRY_CAST`) omzet — anders verlies je stilletjes rijen bij het laden.
- **Behandel columnar storage als een reden om breed te modelleren, niet smal.** Extra kolommen in een warehousetabel zijn vrijwel gratis om op te slaan (dankzij compressie) en kosten niets bij queries die ze niet gebruiken — een andere afweging dan bij een rij-georiënteerde database.
- **Kies clustering-/partitiesleutels op basis van je meest voorkomende filterkolom**, meestal een datum — dit bepaalt of MPP-parallellisatie je query daadwerkelijk versnelt of niet.
- **Scheid staging/bronze expliciet van warehouse/silver**, ook al voelt het als dubbel werk voor een klein project — zie Introduction to Data Engineering voor waarom deze scheiding zich terugbetaalt zodra er een bug wordt gevonden.
- **Behandel data marts (gold) als wegwerpbaar en herbouwbaar.** Als een mart niet zonder dataverlies herbouwd kan worden vanuit warehouse/silver, zit er ongemerkt businesslogica in een laag die dat niet zou moeten bevatten.

## Veelgemaakte Fouten

- **Rapporteren rechtstreeks op de productiedatabase** "omdat het maar een klein bedrijf is" — de OLTP/OLAP-mismatch treedt al op bij bescheiden datavolumes, lang voordat "big data" in beeld komt.
- **Alles in één brede, ongestructureerde tabel dumpen** zonder onderscheid tussen staging, geïntegreerde data en presentatieklare data — werkt de eerste maand, wordt daarna onmogelijk te onderhouden.
- **Een `CAST` gebruiken waar een `TRY_CAST` hoort**, waardoor één rommelige rij een hele nachtelijke load laat falen.
- **Denken dat een data lake een vervanging is voor een warehouse** in plaats van een aanvulling — ruwe bestanden in objectopslag zonder schema-discipline lossen geen enkel analytisch probleem op, ze verplaatsen het probleem alleen.
- **Slowly changing dimensions negeren tot het te laat is** — pas nadenken over historisering nadat een klant al drie keer is verhuisd en niemand meer weet welke omzet bij welke regio hoorde op welk moment.

## Performance Tips

- **Vermijd `SELECT *` op brede warehousetabellen** — het teniet doen van het columnar-voordeel door alle kolommen op te vragen terwijl je er twee nodig hebt, is een van de meest voorkomende manieren om onnodig warehouse-compute te verbranden.
- **Filter op je clustering-/partitiesleutel** waar mogelijk, zodat de MPP-engine hele blokken data kan overslaan in plaats van ze te moeten inlezen en daarna pas te filteren.
- **Wees voorzichtig met veel kleine, frequente writes naar een warehouse.** Warehouses zijn geoptimaliseerd voor grote, batch-gewijze writes; veel losse INSERT-statements (het OLTP-patroon) presteren hier merkbaar slechter dan op een echte transactiedatabase.
- **Gebruik geaggregeerde marts voor dashboards, niet ruwe silver-tabellen.** Een dashboard dat live aggregeert over miljoenen silver-rijen bij elke paginaload is trager en duurder dan een vooraf berekende gold/mart-tabel die elke nacht wordt ververst.

## Interviewvragen

**"Wat is het verschil tussen OLTP en OLAP, en waarom kun je niet zomaar op je productiedatabase rapporteren?"** Let op: begrip van gelijktijdigheid/locking bij OLTP versus doorvoer bij OLAP, en de fysieke reden (rij- versus kolomgeoriënteerde opslag) waarom die twee workloads elkaar in de weg zitten.

**"Leg uit wat columnar storage is en waarom het analytische queries versnelt."** Let op: het concrete voorbeeld dat alleen de benodigde kolommen gelezen worden, plus het compressie-voordeel — niet alleen "het is sneller".

**"Wat is het verschil tussen een data warehouse, een data lake en een lakehouse?"** Let op: schema-on-write versus schema-on-read als kernonderscheid, en het lakehouse als poging beide te combineren via een tabelformaat zoals Delta Lake.

**"Waarom zou je `TRY_CAST` gebruiken in plaats van `CAST` bij het laden van staging-data?"** Let op: het besef dat één foutieve waarde anders een hele batch laat falen, en dat expliciet beslissen wat er met onbruikbare rijen gebeurt beter is dan een pipeline die crasht.

**"Wat is MPP, en hoe helpt het bij het schalen van een warehouse-query?"** Let op: parallelle verwerking over meerdere compute-nodes, en de link naar waarom "een groter warehouse kiezen" op deze platformen vaak volstaat om een trage query te versnellen.

**"Wat is een slowly changing dimension, en waarom is 'gewoon updaten' niet altijd het juiste antwoord?"** Let op: het besef dat overschrijven historische rapportage kan vervormen, en dat dit een bewuste modelleerkeuze is — een volledig antwoord hoort thuis in het hoofdstuk over dimensioneel modelleren, maar het besef dat de keuze bestaat is hier al relevant.

## Relevante Documentatie

- Snowflake: Micro-partitions & Data Clustering — hoe Snowflake columnar storage en MPP in de praktijk implementeert.
- Databricks: Delta Lake — What is a Lakehouse? — het lakehouse-concept vanuit de bron.
- Google BigQuery: Storage Internals — een andere implementatie van dezelfde columnar/MPP-principes.

## Samenvatting

Een data warehouse is geen "grotere database" — het is een fundamenteel andere ontwerpkeuze, gebouwd voor OLAP (weinig, zware, lezende queries) in plaats van OLTP (veel, lichte, schrijvende transacties). Columnar storage en MPP zijn de twee technische pijlers die dat mogelijk maken: kolomsgewijze opslag zodat queries alleen de benodigde kolommen lezen (met flinke compressiewinst als bonus), en parallelle verwerking over meerdere nodes zodat schaal een kwestie van meer rekenkracht wordt in plaats van query-herontwerp. De klassieke staging/warehouse/marts-architectuur is functioneel de voorloper van bronze/silver/gold uit Medallion Architecture — dezelfde verantwoordelijkheidsscheiding, andere naam. Met deze fundamenten begrijp je niet alleen wat je warehouse doet, maar ook waarom — een noodzakelijke basis voordat we in de volgende hoofdstukken dieper ingaan op hoe je die data modelleert.

HOOFDSTUK 4

# Datamodellering

## Introductie

Data modeling wordt vaak verward met "een ERD tekenen" of "tabellen aanmaken", maar het is fundamenteler dan dat: het is de beslissing over hoe de werkelijkheid van een organisatie — klanten, orders, producten, hun onderlinge relaties — wordt vertaald naar een structuur die een database kan opslaan en een query-engine efficiënt kan bevragen. Een goed model maakt correcte antwoorden vanzelfsprekend; een slecht model maakt foute antwoorden onvermijdelijk, ongeacht hoe zorgvuldig de SQL erbovenop is geschreven.

Dit hoofdstuk behandelt de fundamenten die onder élk specifiek modelleerpatroon liggen — normalisatie, entity-relationship modeling, en de reden waarom analytische systemen daar bewust van afwijken. Het is de basis waarop Star Schema & Dimensional Modeling en Data Vault 2.0 voortbouwen; zonder dit fundament is het lastig te begrijpen waarom die twee aanpakken bestaan en wanneer je welke kiest.

## De Drie Abstractieniveaus: Conceptueel, Logisch, Fysiek

Een veelgemaakte vergissing is meteen beginnen met tabellen en kolommen. Ervaren modelleerders werken doorgaans via drie niveaus, elk met een ander doel en publiek.

**Conceptueel model** — de businessentiteiten en hun relaties, zonder technische details. "Een klant plaatst orders. Een order bevat producten." Geen datatypes, geen sleutels, geen tabellen — dit niveau is bedoeld om met domeinexperts (sales, finance, product) te valideren of je de business correct begrijpt, vóórdat er ook maar iets gebouwd wordt.

**Logisch model** — hetzelfde, maar nu met attributen, sleutels en cardinaliteit, nog steeds onafhankelijk van een specifiek databasesysteem. Hier leg je vast dat een order een `order_id`, een `order_date` en een `customer_id` heeft, en dat de relatie tussen `customer` en `order` één-op-veel is (één klant kan meerdere orders hebben, een order hoort bij precies één klant).

**Fysiek model** — de daadwerkelijke implementatie: specifieke datatypes (`VARCHAR(50)` versus `TEXT`), indexen, partitioneringsstrategie, en platform-specifieke keuzes zoals clustering-sleutels in Snowflake of partition-kolommen in Databricks. Dit is het niveau waarop de meeste data engineers dagelijks werken, maar het is een vertaling van het logische model — niet het startpunt.

Het overslaan van de eerste twee niveaus is de meest voorkomende reden dat een fysiek model achteraf pijnlijk blijkt te zijn: technische beslissingen (welke kolom wordt de primary key, hoe worden many-to-many-relaties opgelost) zonder dat businessregels expliciet zijn vastgelegd, leiden tot een schema dat toevallig werkt voor de eerste use case en breekt bij de tweede.

## Entity-Relationship Modeling: de Basis

Een entity-relationship model (ER-model) legt entiteiten (dingen die bestaan: klanten, producten, orders) en hun relaties vast, met drie kernbegrippen die in vrijwel elk schema terugkomen.

**Entiteiten en attributen.** Een entiteit is een op zichzelf staand ding met eigen attributen — `Customer` met `customer_id`, `name`, `email`. Een goede vuistregel: als iets een eigen levenscyclus heeft (aangemaakt, gewijzigd, mogelijk verwijderd, onafhankelijk van andere dingen), is het waarschijnlijk een aparte entiteit, geen attribuut van iets anders.

**Relaties en cardinaliteit.** Relaties tussen entiteiten hebben een cardinaliteit: één-op-één (zeldzaam — meestal een teken dat twee entiteiten eigenlijk één zouden moeten zijn), één-op-veel (de meest voorkomende — één klant, veel orders), en veel-op-veel (bijvoorbeeld producten en categorieën, als een product in meerdere categorieën kan vallen). Veel-op-veel-relaties kun je niet direct in twee tabellen met een foreign key vastleggen — daarvoor is een **bridging table** (ook wel junction table) nodig, een derde tabel die per combinatie één rij bevat.

**Primary keys en foreign keys.** De primary key identificeert een rij uniek binnen zijn eigen tabel; de foreign key verwijst naar de primary key van een andere tabel om de relatie fysiek vast te leggen. Dit mechanisme — niet de tabelnamen, niet de kolomvolgorde — is wat een verzameling losse tabellen tot een samenhangend model maakt.

```
-- Één-op-veel: customer -> orders
CREATE TABLE customer (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT REFERENCES customer(customer_id),
    order_date DATE
);

-- Veel-op-veel: product <-> category, via een bridging table
CREATE TABLE product_category (
    product_id INT REFERENCES product(product_id),
    category_id INT REFERENCES category(category_id),
    PRIMARY KEY (product_id, category_id)
);
```

## Normalisatie: 1NF, 2NF, 3NF

Normalisatie is het proces waarmee je een model stapsgewijs herstructureert om **update-anomalieën** te voorkomen — situaties waarin het wijzigen van één feit meerdere rijen zou moeten raken, met het risico dat je er een mist en de data inconsistent wordt.

Bekijk deze bewust slecht genormaliseerde tabel:

```
-- Ongenormaliseerd: klantgegevens herhaald op elke orderregel
CREATE TABLE orders_denormalized (
    order_id INT,
    customer_name VARCHAR(100),
    customer_email VARCHAR(100),
    product_name VARCHAR(100),
    product_price NUMERIC(10,2),
    quantity INT
);
```

Als een klant haar e-mailadres wijzigt, moet je hier *elke* rij met een order van die klant bijwerken. Vergeet je er één, dan heb je twee verschillende e-mailadressen voor dezelfde klant in je systeem — een **update-anomalie**. Normalisatie lost dit stapsgewijs op:

**1NF (eerste normaalvorm)** — elke kolom bevat atomaire waarden (geen kommagescheiden lijsten in één cel), en er is een duidelijke primary key. Een kolom `product_names` met `"Laptop, Muis, Toetsenbord"` erin schendt 1NF.

**2NF (tweede normaalvorm)** — bouwt voort op 1NF, en elimineert attributen die afhankelijk zijn van slechts een deel van een samengestelde primary key. Als de primary key `(order_id, product_id)` is, maar `customer_email` alleen van `order_id` afhangt (niet van `product_id`), hoort die kolom niet in deze tabel.

**3NF (derde normaalvorm)** — bouwt voort op 2NF, en elimineert attributen die afhankelijk zijn van een ander niet-sleutel-attribuut in plaats van rechtstreeks van de primary key. `customer_email` hangt af van `customer_id`, niet rechtstreeks van `order_id` — dus hoort het in een aparte `customer`-tabel, niet in `orders`.

Het resultaat van het toepassen van deze regels op het voorbeeld hierboven is precies het `customer`/`orders`/`product`-schema dat we eerder al zagen: drie aparte tabellen, gekoppeld via foreign keys, waarbij elk feit precies één keer wordt opgeslagen.

## Waarom Analytische Modellen Denormaliseren

Hier wordt het interessant, en het is de brug naar de rest van dit deel van het handboek: **3NF is uitstekend voor OLTP, maar problematisch voor OLAP.** Een volledig genormaliseerd schema met tien of twintig tabellen betekent dat een simpele analytische vraag — "omzet per productcategorie per maand" — al snel vier of vijf joins vereist. Elke join kost rekenwerk, en op grote datavolumes stapelt dat op tot merkbaar tragere queries, precies het probleem dat we in Data Warehousing Fundamentals bespraken bij columnar storage en MPP.

De oplossing in analytische systemen is **bewust denormaliseren**: sommige redundantie accepteren (dezelfde productcategorie-naam op duizenden rijen herhalen) in ruil voor minder joins en snellere, eenvoudigere queries. Dit is geen "slecht modelleren" — het is een andere afweging, omdat het probleem dat normalisatie oplost (voorkomen van update-anomalieën bij schrijfoperaties) simpelweg niet relevant is in een warehouse waar gold-tabellen elke nacht volledig herbouwd worden vanuit silver, in plaats van rij voor rij bijgewerkt. Dit exacte principe — bewuste, gecontroleerde denormalisatie voor leesprestaties — is de kern van dimensioneel modelleren, uitgewerkt in Star Schema & Dimensional Modeling. Data Vault, behandeld in Data Vault 2.0, kiest weer een andere afweging: een vorm die dichter bij normalisatie blijft, geoptimaliseerd voor traceerbaarheid en flexibiliteit bij schemawijzigingen in plaats van voor leessnelheid.

## Praktisch Voorbeeld: van Genormaliseerd naar Analytisch

Stel je hebt het genormaliseerde OLTP-schema van hierboven (`customer`, `orders`, `product`, `product_category`, `category`). Een analytische vraag als "omzet per categorie per maand, met klantsegment" vereist op dit schema:

```
-- Vijf joins voor één analytische vraag — werkt, maar kostbaar op schaal
SELECT
    c.segment,
    cat.category_name,
    DATE_TRUNC('month', o.order_date) AS month,
    SUM(oi.quantity * oi.unit_price) AS revenue
FROM orders o
JOIN customer c ON c.customer_id = o.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
JOIN product p ON p.product_id = oi.product_id
JOIN product_category pc ON pc.product_id = p.product_id
JOIN category cat ON cat.category_id = pc.category_id
GROUP BY c.segment, cat.category_name, DATE_TRUNC('month', o.order_date);
```

Een gedenormaliseerd analytisch model (een voorloper van de star schema-aanpak uit het volgende hoofdstuk) legt de categorie- en klantgegevens al vooraf plat op de feiten-rij vast, zodat dezelfde vraag zonder joins te beantwoorden is:

```
-- Dezelfde vraag, tegen een al gedenormaliseerd analytisch model
SELECT
    customer_segment,
    category_name,
    DATE_TRUNC('month', order_date) AS month,
    SUM(revenue) AS revenue
FROM gold.sales_flat
GROUP BY customer_segment, category_name, DATE_TRUNC('month', order_date);
```

De tweede versie is niet alleen korter — hij is fundamenteel goedkoper om uit te voeren, omdat het denormaliseren (het samenvoegen van klant-, product- en categoriegegevens) al één keer gebeurde tijdens de nachtelijke transformatie, in plaats van bij elke query opnieuw.

## Modelleerkeuzes die Blijven Terugkomen

Een paar ontwerpbeslissingen komen in vrijwel elk model terug, ongeacht of je normaliseert of denormaliseert.

**Surrogate keys versus natural keys.** Een natural key is een bestaand, betekenisvol attribuut dat toevallig uniek is (een e-mailadres, een BTW-nummer). Een surrogate key is een kunstmatig, betekenisloos, meestal automatisch oplopend identifier (`customer_id: 1, 2, 3, ...`) zonder relatie tot de data zelf. In analytische modellen is de vuistregel: **gebruik bijna altijd surrogate keys**, omdat natural keys kunnen

veranderen (een e-mailadres wordt gewijzigd) of dubbel kunnen voorkomen tussen bronsystemen — een probleem dat je niet wilt hebben zodra die sleutel de basis is van duizenden foreign key-relaties.

**Nullable-ontwerp is een bewuste keuze, geen toeval.** Een kolom `NOT NULL` maken is een garantie die de rest van je pipeline op mag vertrouwen; een kolom nullable laten zonder erover na te denken schuift de vraag "wat betekent afwezigheid hier?" door naar iedereen die de tabel later gebruikt.

**Veel-op-veel-relaties vereisen altijd een bridging table**, nooit een kommagescheiden lijst in een enkele kolom (dat schendt 1NF, zoals hierboven) en meestal ook geen twee foreign keys direct in dezelfde rij (dat werkt alleen als de cardinaliteit gegarandeerd één-op-veel blijft, wat bij veel-op-veel per definitie niet zo is).

**Zwakke entiteiten** (weak entities) zijn entiteiten die niet zelfstandig kunnen bestaan — een `order_item` heeft geen betekenis zonder een bijbehorende `order`, en zijn identiteit is mede afhankelijk van die relatie (vaak via een samengestelde sleutel als `(order_id, line_number)` in plaats van een eigen surrogate key). Het herkennen van een zwakke entiteit is belangrijk omdat het bepaalt of verwijderen cascadeert: als een order wordt verwijderd, horen de bijbehorende orderregels normaliter mee te gaan, terwijl het verwijderen van een klant niet automatisch alle historische orders zou moeten wissen.

**Associatieve entiteiten** zijn een bridging table die zelf ook eigen attributen heeft, niet alleen twee foreign keys. De `product_category`-tabel hierboven zou bijvoorbeeld een attribuut `is_primary_category` kunnen krijgen om aan te geven welke van meerdere categorieën de hoofdcategorie is — op dat moment is het niet langer een pure koppeltabel, maar een entiteit met een eigen betekenis die toevallig een veel-op-veel-relatie modelleert.

## Best Practices

- **Werk via de drie abstractieniveaus**, ook al voelt het conceptuele niveau als "te simpel om op te schrijven" — het is waar je met domeinexperts valideert vóórdat er code geschreven is.
- **Gebruik surrogate keys als primary key** in vrijwel elk analytisch model, en bewaar natural keys als een gewoon (geïndexeerd) attribuut ernaast.
- **Normaliseer voor OLTP, denormaliseer bewust voor OLAP** — beide zijn correct, toegepast op het juiste probleem.
- **Leg cardinaliteit expliciet vast** (één-op-veel, veel-op-veel) vóórdat je tabellen aanmaakt; dit bepaalt of je een foreign key of een bridging table nodig hebt.
- **Documenteer wat NULL betekent** in elke nullable kolom — "onbekend", "niet van toepassing" en "nog niet ingevuld" zijn drie verschillende dingen die dezelfde NULL-waarde verbergt.

## Veelgemaakte Fouten

- **Direct beginnen met tabellen bouwen** zonder eerst een conceptueel of logisch model te valideren met de mensen die de business kennen.
- **Kommagescheiden waarden in één kolom** ("tags: sql,python,snowflake") in plaats van een aparte, genormaliseerde tabel — onmogelijk om efficiënt op te filteren of te aggregeren.
- **Natural keys als primary key gebruiken** in een analytisch model, waardoor een gewijzigd e-mailadres of een dubbele waarde uit een tweede bronsysteem het hele model breekt.
- **Denormalisatie toepassen zonder te begrijpen waarom** — data redundant opslaan "omdat het sneller is" zonder te weten welk specifiek queryprobleem dat oplost, wat later tot inconsistente kopieën leidt.
- **Veel-op-veel-relaties negeren** tot het probleem zich toont, met een haastige work-around (zoals een tweede foreign key-kolom) in plaats van een nette bridging table.

## Performance Tips

- **Indexeer foreign keys** in OLTP-systemen — zonder index op de foreign key-kolom wordt elke join een volledige scan van de gerelateerde tabel.
- **Vermijd over-normaliseren in analytische modellen.** Elke extra join-laag kost rekenwerk bij elke query; in een warehouse waar je toch elke nacht herbouwt, is die kost vaak niet de moeite waard.
- **Meet het aantal joins in je meest voorkomende dashboardqueries.** Vijf of meer joins in een query die honderden keren per dag wordt uitgevoerd, is een sterk signaal dat een gedenormaliseerde gold-tabel de investering waard is.
- **Gebruik samengestelde primary keys spaarzaam** in kolomgeoriënteerde warehouses — een enkele surrogate key is vaak eenvoudiger te clusteren en te filteren op dan een combinatie van kolommen.

## Interviewvragen

**"Wat is het verschil tussen een conceptueel, logisch en fysiek datamodel?"** Let op: begrip dat elk niveau een ander doel en publiek heeft, en dat het overslaan van de eerste twee vaak tot pijnlijke fysieke ontwerpfouten leidt.

**"Leg 3NF uit met een voorbeeld, en welk probleem lost het op?"** Let op: een concreet voorbeeld van een update-anomalie (zoals de herhaalde klantgegevens hierboven) en de drie stappen (1NF, 2NF, 3NF) die dat oplossen.

**"Waarom denormaliseren analytische modellen bewust, terwijl OLTP-systemen normaliseren?"** Let op: het inzicht dat normalisatie schrijfintegriteit optimaliseert en denormalisatie leessnelheid, en dat een warehouse dat elke nacht herbouwt de downsides van denormalisatie (update-anomalieën) niet heeft.

**"Wat is het verschil tussen een surrogate key en een natural key, en wanneer gebruik je welke?"** Let op: surrogate keys als standaardkeuze in analytische modellen vanwege stabiliteit, natural keys bewaard als attribuut voor herkenbaarheid en debugging.

**"Hoe modelleer je een veel-op-veel-relatie?"** Let op: een bridging/junction table met een samengestelde primary key die verwijst naar beide kanten van de relatie — niet een kommagescheiden kolom of een work-around met twee foreign keys.

**"Wat betekent het als een kolom NULL toestaat, en waarom is dat een ontwerpbeslissing?"** Let op: het besef dat NULL meerdere betekenissen kan hebben (onbekend, niet van toepassing, nog niet ingevuld) en dat dit expliciet gedocumenteerd moet worden, niet aan toeval overgelaten.

## Relevante Documentatie

- PostgreSQL: Normalization — praktische uitleg van foreign keys en referentiële integriteit.
- dbt Labs: Data Modeling Techniques — hoe klassieke modelleertheorie zich vertaalt naar dbt-projecten.
- Snowflake: Data Modeling Best Practices — platformspecifieke aanbevelingen voor analytische modellen.

## Samenvatting

Data modeling is het vertalen van de werkelijkheid naar een structuur die correct én efficiënt bevraagbaar is, via drie abstractieniveaus die van conceptueel naar fysiek gaan. Entity-relationship modeling met primary/foreign keys en expliciete cardinaliteit is de basisgrammatica; normalisatie (1NF-3NF) voorkomt update-anomalieën door elk feit precies één keer op te slaan. Analytische systemen wijken hier bewust van af: ze denormaliseren om leesprestaties te winnen, omdat het probleem dat normalisatie oplost (schrijfconsistentie) in een warehouse dat elke nacht herbouwt simpelweg niet speelt. Dat ene inzicht — normaliseren voor schrijven, denormaliseren voor lezen — is de sleutel tot de twee grote analytische modelleeraanpakken die in de volgende hoofdstukken aan bod komen: het snelheid-geoptimaliseerde star schema en het traceerbaarheid-geoptimaliseerde Data Vault.

HOOFDSTUK 5

# Star Schema & Dimensioneel Modelleren

## Introductie

In Data Modeling zagen we het kernprincipe: analytische systemen denormaliseren bewust, omdat leessnelheid belangrijker is dan schrijfconsistentie zodra een tabel elke nacht wordt herbouwd in plaats van rij voor rij bijgewerkt. **Dimensional modeling** — ontwikkeld door Ralph Kimball in de jaren negentig en nog steeds de meest gebruikte modelleeraanpak voor BI-rapportage — is de concrete, gestandaardiseerde implementatie van dat principe. Het **star schema** is het resulterende patroon: één centrale feitentabel omringd door dimensietabellen, in een vorm die zowel voor mensen intuïtief te bevragen is als voor query-engines efficiënt te verwerken.

Dit hoofdstuk behandelt het patroon in de diepte die je nodig hebt om het zelf toe te passen: hoe je een fact table en zijn dimensies ontwerpt, waarom "grain" de belangrijkste beslissing is die je maakt, en hoe je omgaat met veranderende data over tijd via slowly changing dimensions.

## Fact Tables en Dimension Tables

Een star schema bestaat uit precies twee soorten tabellen, met een strikt verschillende rol.

**Fact tables** bevatten de meetbare gebeurtenissen van je business — een verkoop, een klik, een betaling — samen met numerieke, optelbare meetwaarden (**measures**) zoals `amount` of `quantity`, en foreign keys naar de dimensies die die gebeurtenis beschrijven. Een fact table is meestal de grootste tabel in het schema, met miljoenen tot miljarden rijen, en groeit continu doordat nieuwe gebeurtenissen worden toegevoegd.

**Dimension tables** bevatten de beschrijvende context: wie, wat, waar, wanneer. `dim_customer` beschrijft klanten (naam, segment, regio), `dim_product` beschrijft producten (naam, categorie, merk), `dim_date` beschrijft kalenderdagen (weekdag, kwartaal, is_feestdag). Dimensietabellen zijn relatief klein (honderden tot miljoenen rijen, zelden meer) en veranderen traag in vergelijking met de fact table — vandaar de naam "slowly changing dimensions", verderop in dit hoofdstuk.

De vuistregel om onderscheid te maken: **als je het kunt optellen, is het een measure in een fact table; als je het gebruikt om te filteren of te groeperen, is het een attribuut in een dimension table.** `amount` tel je op — measure. `product_category` gebruik je om te groeperen — dimensieattribuut.

## Het Star Schema Patroon

Een fact table voor verkooptransacties, omringd door zijn dimensies, ziet er in DDL als volgt uit:

```
-- Dimensies: relatief klein, beschrijvend, traag veranderend
CREATE TABLE dim_customer (
    customer_key   INT PRIMARY KEY,   -- surrogate key
    customer_id    VARCHAR(20),        -- natural key uit het bronsysteem
    customer_name  VARCHAR(100),
    segment        VARCHAR(50),
    region         VARCHAR(50)
);

CREATE TABLE dim_product (
    product_key    INT PRIMARY KEY,
    product_id     VARCHAR(20),
    product_name   VARCHAR(100),
    category       VARCHAR(50),
    brand          VARCHAR(50)
);
```

```
CREATE TABLE dim_date (
    date_key       INT PRIMARY KEY,   -- vaak als YYYYMMDD, bv. 20260813
    full_date      DATE,
    day_of_week    VARCHAR(10),
    month_name     VARCHAR(10),
    quarter        INT,
    year           INT,
    is_weekend     BOOLEAN
);

-- Fact table: groot, numeriek, foreign keys naar elke dimensie
CREATE TABLE fact_sales (
    sale_id        BIGINT PRIMARY KEY,
    customer_key   INT REFERENCES dim_customer(customer_key),
    product_key    INT REFERENCES dim_product(product_key),
    date_key       INT REFERENCES dim_date(date_key),
    quantity       INT,
    unit_price     NUMERIC(10,2),
    revenue        NUMERIC(12,2)
);
```

De naam "star schema" komt letterlijk van hoe dit eruitziet getekend: `fact_sales` in het midden, met stralen naar `dim_customer`, `dim_product` en `dim_date` eromheen — vergelijk dit met het genormaliseerde vijf-tabellen-schema uit Data Modeling, waar dezelfde informatie over meerdere gekoppelde niveaus verspreid zat.

Bevragen wordt hierdoor merkbaar eenvoudiger — elke dimensie is precies één join verwijderd van de fact table, nooit meer:

```
SELECT
    c.segment,
    p.category,
    d.month_name,
    d.year,
    SUM(f.revenue) AS total_revenue
FROM fact_sales f
JOIN dim_customer c ON c.customer_key = f.customer_key
JOIN dim_product p ON p.product_key = f.product_key
JOIN dim_date d ON d.date_key = f.date_key
GROUP BY c.segment, p.category, d.month_name, d.year;
```

## Grain: de Belangrijkste Beslissing die je Maakt

Voordat je één kolom definieert, moet je antwoord hebben op: **wat is precies één rij in de fact table?** Dit heet de **grain** (korrelgrootte), en het is de eerste en belangrijkste ontwerpbeslissing van elk star schema — alle andere keuzes volgen eruit.

Voor `fact_sales` zou de grain kunnen zijn: "één rij per orderregel" (elke productregel binnen een order is een eigen rij), of "één rij per order" (de hele order samengevat in één rij), of zelfs "één rij per klant per dag" (alle orders van een klant op een dag opgeteld). Deze drie keuzes klinken vergelijkbaar maar hebben compleet verschillende consequenties: bij "per orderregel" kun je exact zien welk product in welke hoeveelheid werd verkocht; bij "per klant per dag" kun je dat niet meer terugvinden, ook niet met de meest geraffineerde query, omdat die informatie simpelweg niet is vastgelegd.

**De vuistregel van Kimball: kies de laagste (fijnste) grain die praktisch haalbaar is.** Je kunt altijd omhoog aggregeren van fijn naar grof in een query (`SUM` over orderregels naar orderniveau), maar je kunt nooit omlaag van grof naar fijn — die detailinformatie is dan onherstelbaar verloren. Een veelgemaakte, kostbare fout is een fact table op een te grove grain bouwen om "het simpeler te houden", en zes maanden later een analist tegenkomen die precies de detailinformatie nodig heeft die is weggegooid.

## Soorten Fact Tables

Niet elke fact table volgt hetzelfde patroon van "één rij per transactie" — drie varianten komen steeds terug.

**Transaction fact tables** — één rij per discrete gebeurtenis, zoals `fact_sales` hierboven. Dit is de meest voorkomende en meest intuïtieve vorm.

**Periodic snapshot fact tables** — één rij per entiteit per vaste tijdsperiode, ongeacht of er activiteit was. Denk aan `fact_account_balance_daily`: elke dag, voor elk account, één rij met het saldo op dat moment — ook als er die dag geen enkele transactie was. Dit is de juiste vorm voor "standen" (voorraad, saldo, aantal actieve abonnementen) in plaats van "gebeurtenissen".

**Accumulating snapshot fact tables** — één rij per proces dat meerdere stadia doorloopt, waarbij de rij wordt bijgewerkt naarmate het proces vordert. Denk aan `fact_order_fulfillment`: één rij per order, met kolommen als `ordered_date`, `paid_date`, `shipped_date`, `delivered_date` — telkens bijgewerkt zodra dat stadium bereikt wordt. Dit is de enige fact table-vorm waarbij je bestaande rijen daadwerkelijk update in plaats van alleen toevoegt, wat het ontwerp van de laadlogica fundamenteel anders maakt dan bij de andere twee varianten.

## Soorten Dimensies

Naast de standaard dimension table komen een paar specifieke patronen vaak genoeg voor om apart te benoemen.

**Conformed dimensions** — dezelfde dimensietabel (bijvoorbeeld `dim_date` of `dim_customer`) hergebruikt over meerdere fact tables heen, met exact dezelfde surrogate keys en attributen. Dit is wat het mogelijk maakt om `fact_sales` en `fact_support_tickets` samen te analyseren via dezelfde `dim_customer` — zonder conformed dimensions eindig je met inconsistente, onvergelijkbare rapportages tussen afdelingen.

**Degenerate dimensions** — een dimensie-achtig attribuut (zoals een ordernummer) dat in de fact table zelf blijft staan omdat het geen eigen beschrijvende attributen heeft en dus geen aparte tabel rechtvaardigt.

**Junk dimensions** — een verzameltabel voor een aantal losse, laag-cardinale vlaggen (`is_gift_wrapped`, `payment_method`, `is_promotional`) die anders elk een eigen kleine dimensietabel zouden vereisen; ze worden gebundeld tot één tabel met alle combinaties, om het aantal foreign keys in de fact table beheersbaar te houden.

**Role-playing dimensions** — dezelfde dimensietabel meerdere keren gebruikt in dezelfde fact table met een andere betekenis, bijvoorbeeld `dim_date` die zowel als `order_date_key` als `ship_date_key` in `fact_order_fulfillment` voorkomt. Dit vereist in SQL meestal een aparte view of alias per rol, zodat je niet twee keer dezelfde tabel met dezelfde kolomnamen hoeft te joinen.

## Slowly Changing Dimensions in Detail

In Data Warehousing Fundamentals beloofden we dit onderwerp volledig te behandelen — hier is het. De vraag is simpel: wat doe je als een klant verhuist en `dim_customer.region` verandert van "Noord" naar "Zuid"? Er zijn vier standaardantwoorden, genummerd naar Kimball's classificatie.

**SCD Type 0 — nooit wijzigen.** Het attribuut wordt na de eerste keer nooit meer bijgewerkt, ook niet als de brondata verandert. Zeldzaam, maar correct voor echt onveranderlijke feiten zoals een geboortedatum.

**SCD Type 1 — overschrijven.** De oude waarde wordt simpelweg vervangen door de nieuwe; er is geen historie. Elk historisch rapport dat ooit is gegenereerd, zou bij het opnieuw draaien nu de nieuwe waarde tonen alsof die altijd al zo was. Geschikt voor het corrigeren van fouten (een verkeerd gespelde naam), niet geschikt voor legitieme wijzigingen die je historisch wilt kunnen reconstrueren.

```
-- SCD Type 1: eenvoudige UPDATE, geen historie
UPDATE dim_customer
SET region = 'Zuid'
WHERE customer_id = 'CUST-4471';
```

**SCD Type 2 — nieuwe rij toevoegen, historie bewaren.** Dit is de meest gebruikte aanpak voor echte, betekenisvolle wijzigingen. In plaats van de bestaande rij te overschrijven, voeg je een nieuwe rij toe met een nieuwe surrogate key, en markeer je met `valid_from`/`valid_to`/`is_current`-kolommen welke rij op welk moment geldig was:

```
CREATE TABLE dim_customer_scd2 (
    customer_key   INT PRIMARY KEY,   -- nieuwe surrogate key per versie
    customer_id    VARCHAR(20),        -- natural key, herhaalt over versies
    customer_name  VARCHAR(100),
    region         VARCHAR(50),
    valid_from     DATE,
    valid_to       DATE,               -- NULL of '9999-12-31' voor de huidige versie
    is_current     BOOLEAN
);

-- Nieuwe versie toevoegen bij een wijziging: sluit de oude rij af, open een nieuwe
MERGE INTO dim_customer_scd2 AS tgt
USING staging.customer_updates AS src
ON tgt.customer_id = src.customer_id AND tgt.is_current = TRUE
WHEN MATCHED AND tgt.region <> src.region THEN
    UPDATE SET valid_to = CURRENT_DATE - 1, is_current = FALSE;

INSERT INTO dim_customer_scd2 (customer_key, customer_id, customer_name, region, valid_from, valid_to, is_current)
SELECT
    next_surrogate_key(),   -- sequence of vergelijkbaar mechanisme
    src.customer_id, src.customer_name, src.region, CURRENT_DATE, NULL, TRUE
FROM staging.customer_updates src
LEFT JOIN dim_customer_scd2 tgt
    ON tgt.customer_id = src.customer_id AND tgt.is_current = TRUE
WHERE tgt.customer_key IS NULL;   -- nieuwe klant, of net afgesloten voor een update
```

Met deze structuur beantwoordt `SELECT * FROM fact_sales f JOIN dim_customer_scd2 c ON c.customer_key = f.customer_key` altijd de regio zoals die gold **op het moment van de verkoop**, omdat de fact table verwijst naar de surrogate key die op dát moment actueel was — een historisch correcte join, ook nadat de klant allang is verhuisd.

**SCD Type 3 — extra kolom voor de vorige waarde.** Een tussenvorm: je voegt een `previous_region`-kolom toe naast `region`, zodat je precies één stap terug kunt kijken, zonder de volledige historie van Type 2. Zelden gebruikt omdat het beperkt is tot één wijziging terug, maar soms voldoende voor use cases als "vergelijk met vorig kwartaal".

## Star Schema versus Snowflake Schema

Een naamgevingsverwarring die het vermelden waard is: een **snowflake schema** (geen relatie met het platform Snowflake) is een variant waarbij dimensietabellen zelf weer genormaliseerd worden — `dim_product` verwijst dan naar een aparte `dim_category`-tabel in plaats van `category_name` direct als attribuut te bevatten. Dit bespaart opslag (categorie-namen worden niet duizenden keren herhaald) maar voegt joins terug toe die het star schema nu juist probeerde te vermijden. De praktische vuistregel: houd dimensies plat (star schema) tenzij een dimensie zo groot en met zo veel herhaling is dat de opslagbesparing van normaliseren zwaarder weegt dan de extra join-kosten — in moderne kolomgeoriënteerde warehouses met sterke compressie is dat inmiddels zelden het geval.

## Best Practices

- **Bepaal de grain expliciet, in woorden, vóórdat je één kolom schrijft** — en leg die beslissing vast in de tabel-documentatie, zodat niemand later een rij toevoegt op een andere grain dan bedoeld.
- **Kies de laagste praktisch haalbare grain.** Je kunt altijd aggregeren naar boven; nooit detail terughalen dat niet is vastgelegd.
- **Gebruik conformed dimensions** over fact tables heen zodra meerdere teams dezelfde entiteiten (klanten, producten) analyseren, om vergelijkbare rapportages te garanderen.
- **Reserveer SCD Type 2 voor attributen waar historische correctheid er daadwerkelijk toe doet** (regio, segment, prijscategorie); gebruik Type 1 voor attributen waar alleen de actuele waarde relevant is (een telefoonnummer, een spelfout-correctie).
- **Automatiseer surrogate key-generatie consistent** (sequences, hash-gebaseerde keys, of een dedicated dbt-macro) zodat SCD Type 2-logica niet per ongeluk dezelfde surrogate key hergebruikt voor twee verschillende versies van een entiteit.

## Veelgemaakte Fouten

- **De grain niet expliciet vastleggen**, waardoor verschillende ETL-runs of teams onbewust op verschillende granulariteit gaan laden — het klassieke recept voor de fan-out-bug uit SQL for Data Engineers.
- **SCD Type 1 gebruiken waar Type 2 nodig was**, waardoor historische rapporten met terugwerkende kracht "veranderen" zodra iemand een dimensie-attribuut bijwerkt — vaak pas ontdekt als een directielid vraagt waarom het kwartaalcijfer van vorig kwartaal ineens anders is dan het destijds gepubliceerde rapport.
- **Measures in een dimensie stoppen, of beschrijvende attributen in een fact table.** Als je een kolom aan het optellen bent in bijna elke query, hoort hij in de fact table; als je hem gebruikt om te filteren/groeperen, hoort hij in een dimensie.
- **Snowflaken zonder concrete reden** — dimensies onnodig normaliseren "omdat het netter is", terwijl het enige resultaat extra joins is zonder meetbaar opslagvoordeel.
- **Natural keys direct in de fact table gebruiken** in plaats van surrogate keys — zie Data Modeling voor waarom dat een analytisch model fragiel maakt zodra een bronsysteem verandert.

## Performance Tips

- **Clusterer/partitioneer fact tables op** `date_key` in de meeste gevallen — het is vrijwel altijd de meest gebruikte filterkolom in analytische queries, en zorgt voor effectieve partition pruning.
- **Houd dimensietabellen plat (star, niet snowflake)** tenzij een specifieke dimensie zo groot is dat normaliseren een meetbaar verschil maakt — extra joins kosten bij elke query, opslagbesparing is eenmalig.
- **Beperk het aantal actieve SCD Type 2-rijen per entiteit.** Een dimensie waar elke kleine wijziging (ook onbeduidende) een nieuwe versie triggert, groeit onnodig snel — bepaal per attribuut bewust of het een SCD Type 2-trigger moet zijn.
- **Materialiseer veelgebruikte fact-dimensie-combinaties** als aparte, kleinere gold-tabellen (bijvoorbeeld `gold.sales_by_month_category`) in plaats van bij elke dashboard-load de volledige fact table te aggregeren.

## Interviewvragen

**"Wat is het verschil tussen een fact table en een dimension table?"** Let op: fact tables bevatten meetbare, optelbare gebeurtenissen met foreign keys; dimension tables bevatten beschrijvende context — de "kun je het optellen"-vuistregel is een goed teken van begrip.

**"Wat is 'grain', en waarom is het de belangrijkste beslissing bij het ontwerpen van een fact table?"** Let op: het besef dat grain bepaalt welk detailniveau vastligt, dat je altijd kunt aggregeren naar boven maar nooit detail kunt terughalen dat niet is vastgelegd, en de vuistregel om de laagste praktisch haalbare grain te kiezen.

**"Leg de vier soorten slowly changing dimensions uit, met een voorbeeld van wanneer je elk zou gebruiken."** Let op: Type 0 (nooit wijzigen), Type 1 (overschrijven, geen historie), Type 2 (nieuwe rij, volledige historie via valid_from/valid_to), Type 3 (één extra kolom voor de vorige waarde) — met een concreet voorbeeld per type.

**"Waarom zou een historisch rapport 'veranderen' als je SCD Type 1 gebruikt in plaats van Type 2?"** Let op: begrip dat Type 1 de oude waarde overschrijft, waardoor een join tussen een fact table en die dimensie altijd de huidige waarde teruggeeft — ook voor transacties die plaatsvonden toen de waarde nog anders was.

**"Wat is het verschil tussen een star schema en een snowflake schema?"** Let op: snowflake normaliseert dimensies verder (extra joins, minder opslag), star houdt dimensies plat (minder joins, wat meer opslag) — en het besef dat "Snowflake" hier geen relatie heeft met het gelijknamige platform.

**"Wat zijn conformed dimensions, en waarom zijn ze belangrijk?"** Let op: dezelfde dimensietabel hergebruikt over meerdere fact tables, noodzakelijk om rapportages tussen afdelingen of processen onderling vergelijkbaar te houden.

**"Hoe zou je een accumulating snapshot fact table ontwerpen voor een orderproces met meerdere stadia?"** Let op: één rij per order die wordt bijgewerkt naarmate stadia (besteld, betaald, verzonden, geleverd) worden bereikt, met een kolom per stadium-datum

— en het besef dat dit de enige fact table-variant is die bestaande rijen update in plaats van alleen toevoegt.

## Relevante Documentatie

- dbt Labs: Dimensional Modeling — hoe het Kimball-patroon zich vertaalt naar moderne dbt-projecten.
- dbt Labs: Snapshots (SCD Type 2) — dbt's ingebouwde mechanisme om SCD Type 2 automatisch te beheren.
- Snowflake: Modeling Star Schemas — platformspecifieke overwegingen voor star schemas in Snowflake.
- Microsoft: Star Schema Design in Power BI — hoe star schema-ontwerp direct de performance van Power BI-rapportages beïnvloedt.

## Samenvatting

Dimensional modeling met het star schema is de concrete uitwerking van het denormalisatie-principe uit het vorige hoofdstuk: één centrale fact table met meetbare, optelbare gebeurtenissen, omringd door beschrijvende dimension tables, elk precies één join verwijderd. De belangrijkste ontwerpbeslissing is de grain — wat precies één rij in de fact table betekent — omdat elke andere keuze daaruit volgt en detail dat niet op de juiste grain is vastgelegd nooit meer terug te halen is. Slowly changing dimensions, vooral Type 2 met zijn valid_from/valid_to-patroon, lossen het probleem op van historisch correcte rapportage wanneer beschrijvende attributen wijzigen. Waar het star schema optimaliseert voor leessnelheid en eenvoud, kiest de aanpak in het volgende hoofdstuk — Data Vault — een compleet andere afweging: geoptimaliseerd voor traceerbaarheid en flexibiliteit bij veranderende bronsystemen, ten koste van query-eenvoud.

HOOFDSTUK 6

# Data Vault 2.0

## Introductie

Het vorige hoofdstuk sloot af met een belofte: waar het star schema optimaliseert voor query-eenvoud en leessnelheid, kiest **Data Vault 2.0** een compleet andere afweging — geoptimaliseerd voor traceerbaarheid, parallel laadbaarheid, en flexibiliteit wanneer bronsystemen wijzigen, ten koste van directe bevraagbaarheid. Data Vault is geen vervanging voor het star schema; het is een aanvullende laag die er meestal *voor* zit, met het star schema (of een vergelijkbare gedenormaliseerde vorm) er weer bovenop gebouwd als presentatielaag.

Dit is bewust het laatste modelleerhoofdstuk in dit deel van het handbook, en het meest geavanceerd — Data Vault lost een specifiek probleem op dat vooral speelt bij grote, complexe organisaties met meerdere overlappende bronsystemen, strenge auditvereisten, of een historie van dure "big bang"-herontwerpen van hun warehouse. Als je bij een klein of middelgroot team werkt met een handvol bronnen, is een goed opgezet star schema vaak ruim voldoende — begrijpen wanneer je Data Vault **niet** nodig hebt, is minstens zo waardevol als begrijpen hoe het werkt.

## Het Probleem dat Data Vault Oplost

Stel je voor: een grote organisatie heeft een CRM voor sales, een apart ERP voor facturatie, en na een overname een tweede CRM van het overgenomen bedrijf — alle drie met een eigen, net iets andere definitie van "klant". In een traditioneel star schema-ontwerp zou je deze drie bronnen tijdens het laden al moeten samenvoegen tot één `dim_customer`, wat betekent dat je vooraf moet beslissen hoe conflicten worden opgelost (welke bron "wint" als namen verschillen?) — een beslissing die vaak nog niet definitief te nemen is, en die je bovendien dwingt te wachten tot alle drie de bronnen geladen zijn voordat je de dimensie kunt bouwen.

Data Vault lost dit op door de **identiteit** van een klant (de business key) volledig te scheiden van de **beschrijving** van die klant per bronsysteem. Elke bron krijgt zijn eigen, onafhankelijk te laden tabel met beschrijvende data; het samenvoegen tot één klantbeeld gebeurt pas later, in een aparte laag, met alle historie van alle bronnen nog intact en herleidbaar naar waar elk gegeven vandaan kwam. Dat is de kern van waar de rest van dit hoofdstuk om draait.

## De Drie Bouwstenen: Hubs, Links, Satellites

Elk Data Vault-model is opgebouwd uit precies drie soorten tabellen, elk met een strikt afgebakende verantwoordelijkheid — vergelijkbaar in geest met de striktheid van fact/dimension tables in het vorige hoofdstuk, maar met een andere verdeling.

**Hubs** bevatten uitsluitend unieke business keys — geen beschrijvende attributen, geen namen, geen adressen. Een `hub_customer` bevat simpelweg elke unieke klant-identifier die ooit is gezien, uit welke bron dan ook, samen met een hash key en metadata over wanneer en waar die voor het eerst opdook. Een hub verandert bijna nooit van vorm en groeit alleen door nieuwe rijen toe te voegen — nooit door bestaande rijen te wijzigen.

**Links** leggen relaties tussen hubs vast — vergelijkbaar met een bridging table voor veel-op-veel-relaties, maar hier het standaardmechanisme voor élke relatie, ook één-op-veel. Een `link_customer_order` legt vast dat een specifieke klant bij een specifieke order hoort, zonder zelf iets over die klant of order te zeggen.

**Satellites** bevatten alle beschrijvende, veranderende attributen, gekoppeld aan een hub of link, met volledige historie — functioneel vergelijkbaar met een SCD Type 2-dimensie uit het vorige hoofdstuk, maar hier de standaardvorm voor élk attribuut, niet een uitzonderingsgeval. Cruciaal: **elke bron krijgt zijn eigen satellite** op dezelfde hub. `sat_customer_crm_sales` en `sat_customer_crm_acquired` kunnen beide aan `hub_customer` hangen, elk met hun eigen attributen, hun eigen laadschema, en zonder dat het laden van de ene satellite ooit hoeft te wachten op de andere.

## Waarom Deze Scheiding? Load-onafhankelijkheid

Dit is de architectonische kern van Data Vault, en het antwoord op de vraag "waarom niet gewoon een dimensie": **elke tabel in een Data Vault-model kan volledig onafhankelijk en parallel geladen worden**, zonder te wachten op enige andere tabel.

Dit werkt omdat hubs, links en satellites gebruikmaken van **hash keys** in plaats van sequentiële surrogate keys. Een sequentiële surrogate key (zoals in het star schema-hoofdstuk) vereist een centrale sequence-generator — je kunt niet twee laadprocessen tegelijk nieuwe surrogate keys laten uitgeven zonder coördinatie. Een hash key wordt berekend uit de business key zelf (bijvoorbeeld `MD5(customer_id)`), volledig deterministisch en zonder enige centrale coördinatie nodig:

```
-- Hash key: deterministisch, geen sequence-generator nodig,
-- dus veilig parallel te berekenen door elk laadproces afzonderlijk
SELECT MD5(UPPER(TRIM(customer_id))) AS customer_hash_key
FROM staging.customers;
```

Omdat elk laadproces onafhankelijk dezelfde hash key berekent voor dezelfde business key, kunnen de CRM-sales-load, de ERP-load en de overgenomen-CRM-load tegelijkertijd draaien, elk zijn eigen hub-rijen en satellite-rijen wegschrijven, zonder ooit op elkaar te hoeven wachten of elkaars data te overschrijven. Dit is precies het probleem dat in het traditionele star schema-ontwerp een probleem was: je hoeft niet meer te wachten tot alle bronnen binnen zijn voordat je kunt beginnen met laden.

## Concreet Voorbeeld: Hub, Link en Satellite voor Customer & Order

```
-- HUB: alleen de business key, nooit gewijzigd na invoegen
CREATE TABLE hub_customer (
    customer_hash_key   CHAR(32) PRIMARY KEY,   -- MD5 van de business key
    customer_id         VARCHAR(20),             -- de business key zelf
    load_date           TIMESTAMP,
    record_source        VARCHAR(50)              -- welk bronsysteem leverde dit aan
);

CREATE TABLE hub_order (
    order_hash_key      CHAR(32) PRIMARY KEY,
    order_id            VARCHAR(20),
    load_date           TIMESTAMP,
    record_source        VARCHAR(50)
);

-- LINK: legt de relatie vast, zonder zelf attributen te bevatten
CREATE TABLE link_customer_order (
    link_hash_key        CHAR(32) PRIMARY KEY,   -- MD5 van beide hash keys samen
    customer_hash_key    CHAR(32) REFERENCES hub_customer(customer_hash_key),
    order_hash_key       CHAR(32) REFERENCES hub_order(order_hash_key),
    load_date            TIMESTAMP,
    record_source        VARCHAR(50)
);

-- SATELLITE: beschrijvende data MET historie, per bronsysteem apart
CREATE TABLE sat_customer_crm_sales (
    customer_hash_key    CHAR(32) REFERENCES hub_customer(customer_hash_key),
    load_date            TIMESTAMP,
    customer_name        VARCHAR(100),
    segment              VARCHAR(50),
    hash_diff            CHAR(32),               -- MD5 van alle attributen samen
    record_source        VARCHAR(50),
    PRIMARY KEY (customer_hash_key, load_date)
);
```

De `hash_diff`-kolom verdient uitleg: het is een MD5-hash van *alle* beschrijvende attributen in de satellite samen. Bij elke nieuwe load bereken je de hash-diff van de binnenkomende rij en vergelijk je die met de laatst bekende hash-diff voor die business key — zijn ze gelijk, dan is er niets veranderd en sla je een nieuwe rij simpelweg over; verschillen ze, dan voeg je een nieuwe satellite-rij toe met een nieuwe `load_date`. Dit is het Data Vault-equivalent van SCD Type 2, maar dan zonder `MERGE`-statements of `UPDATE`-logica — puur `INSERT`, wat het laadproces aanzienlijk eenvoudiger en veiliger maakt om parallel uit te voeren.

```
-- Nieuwe satellite-rij alleen invoegen als de hash-diff daadwerkelijk verschilt
INSERT INTO sat_customer_crm_sales (customer_hash_key, load_date, customer_name, segment, hash_diff, record_source)
SELECT
    s.customer_hash_key,
    CURRENT_TIMESTAMP(),
    s.customer_name,
    s.segment,
    MD5(CONCAT(s.customer_name, '|', s.segment)) AS hash_diff,
    'crm_sales'
FROM staging.customer_updates s
LEFT JOIN (
    SELECT customer_hash_key, hash_diff
    FROM sat_customer_crm_sales
    QUALIFY ROW_NUMBER() OVER (PARTITION BY customer_hash_key ORDER BY load_date DESC) = 1
) latest ON latest.customer_hash_key = s.customer_hash_key
WHERE latest.hash_diff IS NULL
   OR latest.hash_diff <> MD5(CONCAT(s.customer_name, '|', s.segment));
```

## Insert-Only: Waarom Dit Betrouwbaarheid Oplevert

Merk op dat geen van de bovenstaande statements een `UPDATE` of `DELETE` bevat — Data Vault-tabellen zijn in de kern **insert-only**. Dit heeft een directe consequentie die veel waard is in enterprise-omgevingen: elke rij die ooit is geladen, blijft voor altijd bestaan, precies zoals hij binnenkwam, met `record_source` en `load_date` die exact vastleggen waar en wanneer. Voor auditdoeleinden ("toon aan dat dit cijfer op 3 maart correct was volgens de data die we toen hadden") is dit vaak niet optioneel maar een compliance-vereiste — en het is precies waarom Data Vault populair is in de financiële sector en andere zwaar gereguleerde omgevingen.

## Data Vault versus Star Schema versus 3NF

| Aspect | 3NF (OLTP) | Star Schema | Data Vault |
|---|---|---|---|
| Geoptimaliseerd voor | Schrijfintegriteit | Leessnelheid, eenvoud | Traceerbaarheid, parallel laden |
| Wijzigingen bij bronwijziging | Grote impact | Matige impact | Minimale impact — nieuwe satellite |
| Directe bevraagbaarheid | Matig (veel joins) | Uitstekend | Slecht — vereist een laag erbovenop |
| Historie | Meestal geen | Via SCD Type 2 | Standaard, altijd, insert-only |
| Parallel laden | Beperkt | Beperkt (centrale surrogate keys) | Ja, via hash keys |
| Geschikt voor | Applicatiedatabases | BI-rapportage, dashboards | Enterprise-integratie van veel bronnen |

## Business Vault en de Information Mart-laag

Een cruciaal punt dat vaak wordt gemist: **een Data Vault-model is niet bedoeld om rechtstreeks door analisten of BI-tools bevraagd te worden.** De drie-tabellen-per-relatie-structuur, met attributen verspreid over meerdere satellites per bron, maakt zelfs simpele vragen omslachtig om direct te schrijven. Data Vault is een **integratie- en opslaglaag** — de "raw vault" — waar bovenop je een **business vault** (afgeleide, business-logica-verrijkte structuren, nog steeds in Data Vault-stijl) en uiteindelijk een **information mart-laag** bouwt, die er in de praktijk vaak precies uitziet als het star schema uit het vorige hoofdstuk.

Met andere woorden: Data Vault en dimensioneel modelleren zijn in de praktijk vaak geen alternatieven voor elkaar, maar complementair — Data Vault als robuuste, parallel-laadbare, volledig traceerbare basislaag, en een star schema als presentatielaag erbovenop, specifiek gebouwd voor de rapportagebehoeften die op dat moment relevant zijn. Dit sluit direct aan op de bronze/silver/gold-indeling uit Medallion Architecture: een raw vault past goed in de rol van een uitgebreide silver-laag, met information marts als gold.

**Point-in-Time (PIT) en Bridge-tabellen** zijn de gebruikelijke hulpmiddel om de overgang van raw vault naar information mart behapbaar te maken. Een hub met vijf satellites uit vijf verschillende bronnen vereist normaliter vijf aparte joins, elk met zijn eigen "wat was de laatst geldige rij op tijdstip X"-logica — omslachtig en foutgevoelig om telkens opnieuw te schrijven. Een PIT-tabel berekent dit één keer vooraf: voor elke combinatie van business key en relevant tijdstip legt hij vast welke `load_date` in elke satellite op dat moment de meest recente was, zodat de information mart-laag daarna simpele, voorspelbare joins tegen de PIT-tabel kan doen in plaats van tegen elke satellite los. Een bridge-tabel doet hetzelfde voor links: het lost meerdere-hops-relaties (klant → link → order → link → factuur) vooraf op tot directe, plat te bevragen paden.

## Best Practices

- **Gebruik Data Vault alleen als het probleem het rechtvaardigt** — meerdere overlappende bronsystemen, strenge auditvereisten, of een organisatie die regelmatig bronsystemen ziet komen en gaan. Voor een enkel bronsysteem is de overhead zelden de moeite waard.
- **Bouw altijd een presentatielaag erbovenop.** Verwacht niet dat analisten rechtstreeks tegen hubs, links en satellites queryen — een business vault of information mart is geen optioneel extraatje, het is waar Data Vault pas bruikbaar wordt.
- **Eén satellite per bron per hub**, nooit attributen van meerdere bronnen in één satellite mengen — dat is precies de load-onafhankelijkheid die Data Vault zijn kracht geeft.
- **Bereken hash keys consistent** (dezelfde normalisatie — hoofdletters, trimmen van spaties — in elk laadproces), anders berekent hetzelfde business key-record in twee verschillende bronnen per ongeluk twee verschillende hash keys.
- **Behandel hubs, links en satellites als insert-only.** Zodra je een `UPDATE` op een van deze tabellen overweegt, is dat een signaal dat je het patroon verkeerd toepast.

## Veelgemaakte Fouten

- **Data Vault toepassen bij een klein team met één of twee bronnen** — de overhead van drie tabellen per relatie levert dan geen enkel voordeel op ten opzichte van een direct star schema, en kost aanzienlijk meer ontwikkeltijd.
- **Attributen van meerdere bronnen in één satellite mengen**, waardoor je alsnog moet wachten tot alle bronnen binnen zijn voordat je kunt laden — precies het probleem dat Data Vault had moeten oplossen.
- **Vergeten een business vault/information mart-laag te bouwen**, waardoor analisten worden losgelaten op de ruwe hub/link/satellite-structuur en compleet vastlopen op de complexiteit.
- **Hash keys inconsistent berekenen** tussen laadprocessen (bijvoorbeeld wel of niet trimmen van whitespace), waardoor dezelfde entiteit meerdere hub-rijen krijgt.
- **Sequentiële surrogate keys gebruiken in plaats van hash keys** "omdat het vertrouwder aanvoelt" — dit ondermijnt direct de parallel-laadbaarheid die de hele reden is om voor Data Vault te kiezen.

## Performance Tips

- **Indexeer hash keys consequent** op elke hub, link en satellite — dit zijn de kolommen waarop vrijwel elke join in het model draait.
- **Gebruik de hash-diff-vergelijking om onnodige satellite-rijen te vermijden.** Zonder deze check groeit een satellite met elke load, ook als er niets veranderd is, wat de information mart-laag onnodig traag maakt om op te bouwen.
- **Beperk het aantal satellites per hub tot wat praktisch nodig is** — te veel granulaire satellites (één per los attribuut) maakt het samenvoegen in de presentatielaag onnodig complex zonder een navenant voordeel.
- **Bouw information marts als aparte, materialiseerde tabellen**, niet als views direct over de raw vault heen — anders betaal je bij elke dashboardquery opnieuw de kosten van het samenvoegen van meerdere satellites.

## Interviewvragen

**"Wat zijn de drie bouwstenen van Data Vault, en wat is de verantwoordelijkheid van elk?"** Let op: hubs (unieke business keys), links (relaties tussen hubs), satellites (beschrijvende, historische attributen) — en het besef dat elk een strikt afgebakende, niet-overlappende rol heeft.

**"Waarom gebruikt Data Vault hash keys in plaats van sequentiële surrogate keys?"** Let op: hash keys zijn deterministisch en vereisen geen centrale sequence-generator, waardoor meerdere laadprocessen volledig parallel en onafhankelijk kunnen draaien.

**"Wanneer zou je Data Vault kiezen boven een star schema, en wanneer juist niet?"** Let op: Data Vault bij meerdere overlappende bronsystemen, strenge auditvereisten, of frequent wisselende bronnen; een star schema (of zelfs direct 3NF) bij een enkel of klein aantal bronnen waar de extra complexiteit geen voordeel oplevert.

**"Wat is een hash-diff, en waar wordt die voor gebruikt?"** Let op: een hash van alle attributen in een satellite-rij samen, gebruikt om te detecteren of een nieuwe rij daadwerkelijk gewijzigde data bevat vóórdat je een nieuwe satellite-rij toevoegt — voorkomt onnodige rijgroei.

**"Waarom is een Data Vault-model niet geschikt om direct door BI-tools bevraagd te worden?"** Let op: het besef dat attributen verspreid zijn over meerdere satellites per hub, wat zelfs simpele queries omslachtig maakt, en dat een business vault/information mart-laag daarom altijd nodig is.

**"Hoe verhoudt Data Vault zich tot medallion architecture (bronze/silver/gold)?"** Let op: een raw vault functioneert vaak als een uitgebreide, sterk gestructureerde silver-laag, met information marts (vaak star schemas) als gold — complementair, niet concurrerend.

## Relevante Documentatie

- dbt Labs: Data Vault Modeling — hoe Data Vault-patronen zich vertalen naar dbt-projecten.
- Snowflake: Data Vault Architecture — platformspecifieke overwegingen voor hash keys en insert-only laadpatronen.
- Databricks: Building a Data Vault on the Lakehouse — hoe Delta Lake's ACID-garanties Data Vault-laadpatronen ondersteunen.

## Samenvatting

Data Vault 2.0 lost een specifiek, herkenbaar probleem op: hoe integreer je data uit meerdere, mogelijk conflicterende bronsystemen, volledig traceerbaar en parallel laadbaar, zonder vooraf te hoeven beslissen hoe conflicten worden opgelost? De drie bouwstenen — hubs voor identiteit, links voor relaties, satellites voor beschrijvende historie — scheiden precies wat zelden verandert (business keys) van wat vaak verandert (attributen), elk per bron apart, gekoppeld via deterministische hash keys die elke centrale coördinatie tussen laadprocessen overbodig maken. De prijs voor deze flexibiliteit is bevraagbaarheid: een raw Data Vault is niet bedoeld voor directe rapportage, en vereist altijd een business vault of information mart-laag erbovenop — vaak weer een star schema, wat laat zien dat de modelleeraanpakken uit dit en het vorige hoofdstuk elkaar aanvullen in plaats van uitsluiten. Met deze fundamenten (data modeling, star schema, Data Vault) op zak is het tijd om te kijken naar hoe data daadwerkelijk van bron naar deze modellen beweegt — het onderwerp van het volgende hoofdstuk, ETL vs ELT.

HOOFDSTUK 7

# ETL vs ELT

## Introductie

In Introduction to Data Engineering noemden we kort dat de verschuiving van ETL naar ELT samenviel met de opkomst van goedkope, elastische cloud-compute in data warehouses. Dat is de geschiedenis; dit hoofdstuk behandelt de techniek — wat er precies anders gebeurt tussen deze twee patronen, welke concrete voor- en nadelen daaruit volgen, en hoe je in de praktijk beslist welke aanpak bij welke situatie past. Het is geen achterhaalde discussie: zelfs op platformen die volledig voor ELT zijn ontworpen, zijn er legitieme, terugkerende redenen om alsnog voor ETL te kiezen, met name rond compliance.

## De Kernvraag: Waar Vindt Transformatie Plaats?

Beide patronen bestaan uit dezelfde drie stappen — extract, transform, load — het verschil zit in de volgorde en, belangrijker, in **waar** de transformatiestap fysiek plaatsvindt.

Bij **ETL (Extract, Transform, Load)** wordt data uit de bron gehaald, getransformeerd op een aparte, dedicated verwerkingsengine (historisch een on-premises ETL-server met tools als Informatica of SSIS, tegenwoordig soms een aparte compute-laag), en pas dan geladen in het uiteindelijke warehouse — in de definitieve, gemodelleerde vorm. Het warehouse ziet nooit de ruwe, ongetransformeerde data.

Bij **ELT (Extract, Load, Transform)** wordt data eerst, grotendeels ongewijzigd, in het warehouse geladen — dit is exact de bronze-laag uit Medallion Architecture — en pas daarna getransformeerd, met de eigen compute van het warehouse zelf, meestal via SQL (dbt) of Spark. Het warehouse ziet wél de ruwe data, tenminste tijdelijk.

Dit klinkt als een klein verschil in volgorde, maar de gevolgen raken bijna elk aspect van hoe je een pipeline bouwt, onderhoudt en beveiligt.

## ETL in Detail: Transform Before Load

Het transformeren gebeurt hier op een aparte engine, losstaand van het warehouse. Dit heeft twee directe consequenties.

**Voordeel: gevoelige data kan al gefilterd of gemaskeerd worden vóórdat ze het warehouse bereikt.** Als een organisatie wettelijk verplicht is om BSN's of medische gegevens nooit ongemaskeerd in een bepaald systeem te laten belanden, biedt ETL een harde garantie: die maskering gebeurt vóór het laden, dus de ruwe waarde bestaat op geen enkel moment in het warehouse — niet eens tijdelijk, niet eens in een tabel die "bronze" heet en toegang beperkt. Dit is precies waarom veel financiële instellingen en overheidsorganisaties, ondanks de moderne voorkeur voor ELT, bepaalde pijplijnen bewust als ETL blijven bouwen.

**Nadeel: transformatielogica leeft buiten het warehouse**, vaak in een proprietary tool met een grafische interface. Wijzigingen zijn lastig te versiebeheren (geen nette Git-diff van een drag-and-drop-canvas), lastig te reviewen in een pull request, en de aparte verwerkingsengine is extra infrastructuur die apart geschaald, gepatcht en onderhouden moet worden — precies het probleem dat in Introduction to Data Engineering werd beschreven als de aanleiding voor de opkomst van de rol "data engineer" met echte software-engineeringvaardigheden.

## ELT in Detail: Load Before Transform

Bij ELT verschuift transformatie naar ná het laden, uitgevoerd met de compute van het warehouse zelf.

**Voordeel: flexibiliteit en eenvoud.** Omdat de ruwe data al in het warehouse staat (de bronze-laag), kun je transformatielogica achteraf wijzigen, uitbreiden of zelfs volledig herzien zonder opnieuw te hoeven extraheren uit het bronsysteem — een principe dat we

in Introduction to Data Engineering al zagen als reden om bronze altijd te bewaren. Transformatie gebeurt bovendien in SQL (via dbt) of Spark, wat betekent dat het gewoon code is: Git-versiebeheerd, reviewbaar in pull requests, getest via CI/CD — zie CI/CD for Data Engineering.

**Nadeel: gevoelige data landt eerst ruw in het warehouse**, wat een compliance-overweging is die je expliciet moet adresseren (via kolomniveau-maskering direct in de silver-transformatie, en strikte toegangscontrole op bronze — zie Data Security) in plaats van dat het probleem automatisch is opgelost door de architectuur zelf. Daarnaast kan ELT duurder uitpakken als transformatie-compute niet bewust wordt beheerd: dezelfde zware aggregatiequery die je in een oude ETL-tool op een goedkope, dedicated server liet draaien, verbruikt nu warehouse-credits — relevant genoeg dat we er in Modern Data Platform Architecture een heel kostenvoorbeeld aan wijdden.

## Praktisch Voorbeeld: Dezelfde Pipeline, Twee Patronen

Stel: je wilt klantgegevens van een CRM naar een warehouse brengen, waarbij e-mailadressen gemaskeerd moeten worden voor niet-geautoriseerde gebruikers.

**De ETL-aanpak** — maskering gebeurt vóór het laden, in het extractiescript zelf:

```
# etl_style_extract.py — maskering gebeurt VOOR het laden
import hashlib

def mask_email(email: str) -> str:
    return hashlib.sha256(email.encode()).hexdigest()[:16] + "@masked.local"

def transform_before_load(raw_customers: list[dict]) -> list[dict]:
    transformed = []
    for c in raw_customers:
        transformed.append({
            "customer_id": c["id"],
            "email_masked": mask_email(c["email"]),
            "segment": c["segment"].upper(),
        })
    return transformed

# Pas NA transformatie wordt er geladen — het warehouse ziet nooit
# de ongemaskeerde e-mailadressen, ook niet tijdelijk.
load_to_warehouse(transform_before_load(extract_from_crm()))
```

**De ELT-aanpak** — ruwe data (inclusief het ongemaskeerde e-mailadres) landt eerst in bronze, met beperkte toegang, en maskering gebeurt in de silver-transformatie:

```
-- Stap 1 (Extract + Load): raw_customers.email bevat het ongemaskeerde adres.
-- Toegang tot raw.customers is beperkt tot het data-engineeringteam.

-- Stap 2 (Transform, in dbt): maskering gebeurt hier, bij het bouwen van silver
CREATE OR REPLACE TABLE silver.customers AS
SELECT
    customer_id,
    SHA2(email, 256) AS email_masked,
    UPPER(segment) AS segment
FROM raw.customers;
```

Functioneel is de output identiek. Het verschil zit in het risicoprofiel: bij de ELT-versie bestaat er, tussen stap 1 en stap 2, een tabel met ongemaskeerde e-mailadressen — beveiligd via toegangscontrole, maar aanwezig. Bij de ETL-versie bestaat die tabel nooit. Voor de meeste organisaties is toegangscontrole op bronze afdoende; voor sommige (denk aan medische data, financiële transactiedata onder strenge regelgeving) is "bestaat nooit" een harde vereiste die alleen ETL kan garanderen.

## Wanneer ETL Nog Steeds de Juiste Keuze Is

Ondanks dat de meeste nieuwe dataplatformen standaard voor ELT kiezen, is ETL geen verouderd patroon dat je nooit meer zou moeten overwegen. Concrete situaties waarin ETL nog steeds terecht de voorkeur krijgt:

**Wettelijk verplichte data-minimalisatie.** Wanneer regelgeving voorschrijft dat bepaalde ruwe data een systeem nooit mag bereiken, ook niet tijdelijk of achter toegangscontrole, is "transformeren vóór het laden" de enige architectuur die dat daadwerkelijk garandeert.

**Extreem grote, zelden gebruikte brondata.** Als een bron enorme volumes bevat waarvan je slechts een klein, vooraf bekend deel nodig hebt, kan filteren vóór het laden aanzienlijk goedkoper zijn dan alles laden en pas achteraf filteren — vooral als warehouse-opslagkosten of laadtijd een reële beperking vormen.

**Legacy-integraties met vaste, stabiele transformatie-eisen.** Een koppeling met een verouderd extern systeem waarvan het contract al jaren niet wijzigt, heeft weinig baat bij de flexibiliteit die ELT biedt — de stabiliteit van een bestaande ETL-tool omzetten naar ELT levert dan vooral migratierisico op zonder concreet voordeel.

**Contractuele beperkingen van een externe partner.** Sommige dataleveranciers (bijvoorbeeld in de zorg of financiële sector) staan in hun leveringsovereenkomst expliciet niet toe dat bepaalde velden ooit "at rest" in een systeem buiten hun eigen infrastructuur belanden, ook niet tijdelijk en ook niet versleuteld. In die situatie is transformeren (en desnoods volledig weglaten van het betreffende veld) vóór het laden niet optioneel maar contractueel verplicht — een variant van het compliance-argument, maar afkomstig van een externe partij in plaats van interne regelgeving.

## Hybride Aanpak: de Praktijk is Zelden Zuiver

De discussie hierboven suggereert een keuze tussen twee uitersten, maar in de praktijk gebruikt vrijwel elk volwassen dataplatform **beide patronen tegelijk**, voor verschillende bronnen binnen hetzelfde platform. Een typisch voorbeeld: Flowmetric (de case study uit Modern Data Platform Architecture) laadt de meeste bronnen — Stripe, het CRM, event-tracking — via ELT, omdat die data geen bijzondere gevoeligheid heeft en de flexibiliteit van SQL-transformatie in dbt de voorkeur verdient. Voor één specifieke bron — medische verzuimgegevens van het HR-systeem, onder strikte AVG-vereisten — kiest hetzelfde platform bewust voor een ETL-stap: een klein, apart Python-script dat BSN's en diagnosecodes al vóór het laden pseudonimiseert, zodat die gevoelige velden het warehouse nooit ongemaskeerd bereiken.

Dit is geen inconsistentie in de architectuur — het is precies de juiste toepassing van het principe uit dit hoofdstuk: **de keuze tussen ETL en ELT is een beslissing per databron, niet een platformbrede religieuze keuze.** Een architectuurdocument dat stelt "wij zijn een ELT-platform" mist de nuance die in de praktijk nodig is; een architectuurdocument dat per bron beargumenteert waarom voor ETL of ELT is gekozen, is bruikbaarder en eerlijker over de afwegingen die daadwerkelijk zijn gemaakt.

## Tooling en Kostenvergelijking

De keuze tussen ETL en ELT heeft ook een directe doorwerking op welke tools je gebruikt en hoe de kosten zich verhouden.

**ETL-tooling** — traditioneel Informatica, SSIS, Talend, of specifieke cloud-varianten zoals Azure Data Factory's mapping data flows (die transformatie op een aparte Spark-compute-laag uitvoeren, los van het doelwarehouse). De kosten zitten in twee aparte compute-lagen: de ETL-engine zelf (vaak licentiekosten plus infrastructuurkosten) én het warehouse waar het eindresultaat landt. Voor een klein aantal pipelines met bescheiden volumes is dit vaak overzichtelijk; bij tientallen pipelines wordt de aparte ETL-laag zelf een significante kostenpost én een aparte operationele verantwoordelijkheid (patches, schaalbaarheid, monitoring) naast het warehouse.

**ELT-tooling** — Fivetran of Airbyte voor de EL-stap (vaak geprijsd per verwerkte rij of per connector), gecombineerd met dbt voor de T-stap (waar de kosten in warehouse-compute zitten, niet in een aparte tool-licentie). Voor de meeste teams is dit financieel voordeliger zodra het aantal bronnen groeit, simpelweg omdat er geen aparte, los te schalen ETL-laag hoeft te worden onderhouden — de kosten schalen mee met wat het warehouse toch al kost. De valkuil is dat transformatiekosten dan minder zichtbaar worden als aparte regel, wat precies de reden is waarom workload-scheiding en kostmonitoring (zie Modern Data Platform Architecture) belangrijker worden naarmate een ELT-platform groeit.

```
# orchestration.yml — illustratief: hoe een hybride aanpak er
# in een Airflow/Dagster-achtige orchestrator uitziet
pipelines:
  - name: stripe_orders
    pattern: ELT
```

```
    steps: [fivetran_sync, dbt_run_staging, dbt_run_marts]

  - name: hr_absence_data
    pattern: ETL
    steps: [python_extract_and_pseudonymize, load_to_warehouse]
    note: "AVG-vereiste: BSN/diagnosecodes nooit ongemaskeerd in warehouse"
```

## Reverse ETL: een Verwante maar Andere Term

Een term die vaak voor verwarring zorgt: **reverse ETL** is geen derde variant van hetzelfde spectrum, maar een tegenovergestelde beweging — data die *van* het warehouse *terug* naar operationele tools stroomt (Salesforce, HubSpot, een supportsysteem), zodat bijvoorbeeld een sales-medewerker in het CRM een door het dataplatform berekende klantscore kan zien. Tools als Hightouch en Census zijn hier gespecialiseerd in. Dit hoort thuis in de serving-laag die we in Modern Data Platform Architecture bespraken, niet in de ingestion-laag — noem het niet "ELT achterstevoren", want de tussenliggende architectuur (extractie uit een warehouse, geen ruwe extractie uit een bron, geen medallion-lagen) is wezenlijk anders.

## Best Practices

- **Kies ELT als standaard voor nieuwe cloudplatformen**, tenzij een concrete, benoembare reden (compliance, extreem volume, stabiele legacy-integratie) voor ETL pleit.
- **Documenteer expliciet welke pipelines om compliance-redenen ETL zijn**, zodat die keuze niet per ongeluk wordt "gemoderniseerd" naar ELT door een toekomstige engineer die de reden niet kent.
- **Beperk toegang tot bronze/raw-tabellen strikt** wanneer je voor ELT kiest — dit is de architecturale compensatie voor het feit dat ruwe, mogelijk gevoelige data daar tijdelijk staat.
- **Behandel maskering als onderdeel van de silver-transformatie**, niet als een losse, makkelijk te vergeten nabewerkingsstap.
- **Verwar reverse ETL niet met ELT** in documentatie of architectuurdiagrammen — het zijn tegenovergestelde databewegingen met andere doelen.
- **Maak de ETL/ELT-keuze per bron expliciet en beargumenteerd**, bijvoorbeeld in een orchestratie-config zoals hierboven, zodat een nieuwe engineer in één oogopslag ziet waarom een specifieke pipeline afwijkt van het platformbrede standaardpatroon.

## Veelgemaakte Fouten

- **Automatisch voor ELT kiezen zonder de compliance-implicaties te overwegen** — vooral bij persoonsgegevens of financiële data, waar "het staat achter toegangscontrole" soms niet volstaat als wettelijke garantie.
- **ETL-tooling aanhouden puur uit gewoonte**, terwijl er geen concrete reden meer is die ELT in de weg staat — wat onnodige infrastructuur en onderhoud oplevert.
- **Maskering vergeten of te laat toepassen** in een ELT-pipeline, waardoor gevoelige data verder stroomafwaarts (silver, gold, of zelfs een BI-dashboard) terechtkomt dan bedoeld.
- **Reverse ETL bouwen als een aparte "omgekeerde ELT-pipeline"** in plaats van het te behandelen als onderdeel van de serving-laag, met eigen architectuuroverwegingen.

## Performance Tips

- **Bij ELT: monitor transformatie-compute apart van ingestion-compute** — zie het workload-scheidingsprincipe uit Modern Data Platform Architecture, zodat een zware transformatiejob niet onopgemerkt de warehouse-rekening laat oplopen.
- **Bij ETL: schaal de transformatie-engine onafhankelijk van het warehouse** — een veelgemaakte performance-fout is een ETL-server die niet meegroeit met toenemend datavolume, terwijl het warehouse zelf ruim voldoende capaciteit heeft.
- **Filter zo vroeg mogelijk in beide patronen**, ook bij ELT — "laad alles, filter later" betekent niet dat vroege, evident onnodige data (test-records, duidelijk corrupte rijen) per se moet worden meegenomen.

- **Meet de daadwerkelijke laadtijd versus transformatietijd** voordat je een architectuurwijziging voorstelt — in de praktijk is bij de meeste moderne cloudpipelines de transformatiestap, niet de extractie, de grootste kostenpost.

## Interviewvragen

**"Wat is het verschil tussen ETL en ELT, technisch gezien?"** Let op: waar transformatie fysiek plaatsvindt — op een aparte engine vóór het laden (ETL) versus met warehouse-compute ná het laden (ELT) — niet alleen de lettervolgorde.

**"Waarom is ELT de dominante aanpak geworden op moderne cloudplatformen?"** Let op: de ontkoppeling van storage en compute die cloud warehouses goedkoop maakt, waardoor transformeren met warehouse-compute niet langer een premium kost zoals bij on-premises systemen.

**"In welke situatie zou je bewust voor ETL kiezen op een modern cloudplatform?"** Let op: compliance-vereisten waarbij ruwe, gevoelige data een systeem nooit mag bereiken — niet "ETL is ouderwets, dus nooit meer gebruiken".

**"Wat is reverse ETL, en hoe verschilt het van ELT?"** Let op: reverse ETL stroomt van warehouse terug naar operationele tools (tegenovergestelde richting), hoort bij de serving-laag, en is geen variant van het extract/transform/load-spectrum zelf.

**"Hoe zou je gevoelige data maskeren in een ELT-pipeline, gegeven dat de ruwe data al in het warehouse staat?"** Let op: maskering als expliciete stap in de silver-transformatie, gecombineerd met strikte toegangscontrole op de bronze-laag als extra beschermingslaag.

**"Is het normaal dat een dataplatform zowel ETL- als ELT-pipelines heeft? Is dat geen inconsistente architectuur?"** Let op: het besef dat de keuze per databron wordt gemaakt op basis van gevoeligheid en compliance-eisen, niet platformbreed — een hybride aanpak is in de praktijk de norm, geen architectonische smet.

## Relevante Documentatie

- dbt Labs: ELT Explained — dbt Labs' eigen uitleg van waarom ELT hun tool-ontwerp bepaalt.
- Snowflake: Data Loading Overview — hoe laadpatronen praktisch werken op een ELT-georiënteerd platform.
- Databricks: What is ETL? — Databricks' perspectief vanuit een lakehouse-architectuur.

## Samenvatting

ETL en ELT verschillen in waar transformatie plaatsvindt: vóór het laden op een aparte engine (ETL), of ná het laden met de compute van het warehouse zelf (ELT). Cloud-economics — goedkope, elastische warehouse-compute — hebben ELT tot de dominante aanpak gemaakt, met als bijkomend voordeel dat transformatielogica gewone, versiebeheerde code wordt in plaats van een proprietary tool-configuratie. Toch blijft ETL de juiste keuze in specifieke situaties, vooral wanneer compliance vereist dat gevoelige ruwe data een systeem nooit bereikt, ook niet tijdelijk. Reverse ETL is een gerelateerd maar wezenlijk ander concept — de omgekeerde databeweging, van warehouse naar operationele tools. Met dit onderscheid helder, is het volgende hoofdstuk de logische vervolgstap: hoe je die ELT-transformatie in de praktijk structureert via medallion architecture.

HOOFDSTUK 8

# Medallion Architectuur

## Introductie

Bronze, silver en gold zijn inmiddels in bijna elk hoofdstuk van dit handbook voorbijgekomen — als de laag waar ruwe data landt, als de plek waar deduplicatie gebeurt, als het patroon achter de case study in Modern Data Platform Architecture. Dit hoofdstuk herhaalt die basisdefinities niet, maar gaat een niveau dieper: wat mag er precies in elke laag gebeuren en wat nooit, hoe vertaalt dit patroon zich naar een concrete dbt-projectstructuur en materialization-strategie, hoe test je elke laag anders, en hoe verwerk je een AVG-verwijderverzoek door een laag heen die per ontwerp onveranderlijk hoort te zijn?

## Het Contract per Laag: Wat Hoort Waar

De definities "ruw", "schoon" en "business-klaar" zijn een startpunt, maar te vaag om als engineeringrichtlijn te dienen. Een preciezer contract, geformuleerd als wat wel en niet is toegestaan:

**Bronze — toegestaan:** ongewijzigd opslaan van brondata, technische metadata toevoegen (`_ingested_at`, `_source_system`), append-only schrijven. **Nooit toegestaan:** rijen filteren op basis van businesslogica, waarden transformeren of casten, rijen updaten of verwijderen (behalve voor het uitzonderingsgeval van wettelijk verplichte verwijdering, zie verderop). Zodra iemand in een bronze-transformatie een `WHERE`-clausule op business-logica tegenkomt (`WHERE status != 'test'`), is dat een teken dat die logica in silver hoort.

**Silver — toegestaan:** deduplicatie, type-casting, het combineren van meerdere bronnen tot één betrouwbare representatie per entiteit, basale validatie (rijen met een ontbrekende primary key verwijderen of quarantaine plaatsen), kolomniveau-maskering van gevoelige velden. **Nooit toegestaan:** business-specifieke aggregaties of KPI-berekeningen — silver blijft op transactie-/entiteitsniveau, niet op rapportageniveau.

**Gold — toegestaan:** aggregaties, joins tussen meerdere silver-entiteiten, herberekening naar de exacte vorm die een dashboard of downstream-systeem nodig heeft — inclusief de fact/dimension-structuren uit Star Schema & Dimensional Modeling. **Nooit toegestaan:** rechtstreeks lezen uit bronze, ook niet als "snelkoppeling" voor een dringende rapportagevraag — elke gold-tabel bouwt voort op silver, zodat datakwaliteitsregels uit silver gegarandeerd worden toegepast voordat data gold bereikt.

## Implementatie: Schema's, Catalogen en Lakehouse-zones

Dit contract is techniek-agnostisch, maar de fysieke implementatie verschilt per platform.

Op **Snowflake** wordt dit meestal uitgedrukt als aparte schema's binnen één database (`raw.orders`, `staging.orders`, `analytics.orders`) of, bij sterkere isolatie-eisen, aparte databases per laag met specifieke rolgebaseerde toegang per database. Op **Databricks** wordt dit typisch vertaald naar Unity Catalog-catalogi of -schema's (`bronze.orders`, `silver.orders`, `gold.orders`), vaak gekoppeld aan fysiek gescheiden opslaglocaties in de onderliggende cloud storage. Op **Microsoft Fabric** verschijnt dit als aparte Lakehouses of als mappenstructuur binnen één Lakehouse (`Bronze/orders`, `Silver/orders`, `Gold/orders`), met OneLake als onderliggende gedeelde opslaglaag.

In alle drie de gevallen geldt hetzelfde onderliggende principe: **toegangscontrole volgt de laag, niet de tabel.** Het data-engineeringteam krijgt schrijftoegang tot bronze en silver; analisten krijgen alleen leestoegang tot gold — een directe toepassing van de governance-principes uit Modern Data Platform Architecture, nu concreet vertaald naar namespace-ontwerp.

## dbt-projectstructuur: Staging, Intermediate, Marts

Wie een dbt-project opzet, ontdekt al snel dat dbt zijn eigen naamgevingsconventie hanteert — `staging`, `intermediate`, `marts` — die niet toevallig één-op-één overeenkomt met bronze/silver/gold, maar er wel een specifieke extra nuance aan toevoegt:

```
models/
├── staging/           # 1 model per bron-tabel, minimale transformatie
│   ├── stg_orders.sql       # hernoemen, casten — nog dicht bij bronze
│   └── stg_customers.sql
├── intermediate/       # combineren van meerdere staging-modellen
│   └── int_orders_enriched.sql   # dit is waar silver echt silver wordt
└── marts/              # business-klare, vaak per domein gegroepeerd
    ├── finance/
    │   └── fct_revenue.sql
    └── sales/
        └── dim_customer.sql
```

`staging` is dun bronze-naar-silver-vertaalwerk: kolommen hernoemen naar een consistente conventie, types casten, verder niets — bewust minimalistisch, zodat elk staging-model maximaal overzichtelijk blijft en één-op-één een bron-tabel volgt. `intermediate` is waar de echte silver-logica zit: deduplicatie, het combineren van meerdere staging-modellen, validatie. `marts` is gold: per business-domein georganiseerd, vaak in de fact/dimension-vorm uit het star schema-hoofdstuk. Deze indeling is dbt's eigen conventie (zie de dbt Labs-structuurgids) en niet verplicht, maar wordt inmiddels breed genoeg gebruikt dat het als de facto standaard binnen dbt-projecten geldt, naast medallion architecture als de facto standaard op platformniveau.

## Materialization-strategie per Laag

Een veelgemaakte, kostbare misvatting is elke laag op dezelfde manier materialiseren. In de praktijk verschilt de optimale keuze sterk per laag:

- **Bronze/staging** — meestal een `view`, of bij grotere volumes een `incremental table`. Omdat staging-modellen zo dun zijn (alleen hernoemen/casten), is het materialiseren als view vaak goedkoop genoeg en bespaart het onnodige opslag van een bijna-identieke kopie van bronze.
- **Silver/intermediate** — meestal een `table`, of bij groeiende historie een `incremental table` met een duidelijke unique key — dit is waar de zwaardere deduplicatie- en combinatielogica zit, die je niet bij elke downstream-query opnieuw wilt laten berekenen.
- **Gold/marts** — bijna altijd een `table`, soms `incremental` voor zeer grote fact tables. Dit zijn de tabellen die BI-tools rechtstreeks bevragen; ze moeten voorspelbaar snel zijn, wat een view (die de onderliggende query elke keer herberekent) zelden kan garanderen.

```
# dbt_project.yml — materialization-strategie per laag, per folder
models:
  my_project:
    staging:
      +materialized: view
    intermediate:
      +materialized: table
    marts:
      +materialized: table
      finance:
        +materialized: incremental
        +unique_key: order_id
```

## Testing Strategy per Laag

Net als materialization verschilt ook wat je test fundamenteel per laag — een teststrategie die "overal dezelfde tests" toepast, verspilt compute op tests die in die specifieke laag niets betekenisvols opleveren.

**Bronze** — test vooral **aanwezigheid en versheid**: is de verwachte data daadwerkelijk geland, en niet ouder dan verwacht? Inhoudelijke tests (uniciteit, referentiële integriteit) zijn hier grotendeels zinloos — bronze bevat per definitie nog rommel, dat is precies waarom de

laag bestaat.

**Silver** — test **structurele correctheid**: `not_null` op verplichte velden, `unique` op de entiteits-sleutel, `relationships`-tests die refereren naar andere silver-tabellen. Dit is de laag waar Data Quality & Testing het zwaarst wordt toegepast, omdat dit de laatste plek is voordat data zich vermengt in complexere gold-berekeningen.

**Gold** — test **business-logica-correctheid**: komt de som van de regels overeen met het totaal, telt een KPI op tot een bekend controlegetal, wijkt een dag-op-dag-verandering niet onverklaarbaar sterk af? Dit zijn vaak custom, domeinspecifieke tests in plaats van dbt's generieke ingebouwde tests.

```
# schema.yml — teststrategie geconcentreerd per laag
models:
  - name: stg_orders          # bronze/staging: alleen versheid
    tests:
      - dbt_utils.recency:
          datepart: hour
          field: _ingested_at
          interval: 26

  - name: int_orders_enriched  # silver/intermediate: structuur
    columns:
      - name: order_id
        tests: [not_null, unique]
      - name: customer_id
        tests:
          - relationships:
              to: ref('int_customers')
              field: customer_id

  - name: fct_revenue          # gold/marts: business-logica
    tests:
      - dbt_utils.expression_is_true:
          expression: "total_revenue >= 0"
```

## Streaming Data binnen Medallion Architecture

Het patroon is ontworpen met batch in gedachten, maar werkt met aanpassingen ook voor streaming-bronnen (zie de batch-versus-streaming-afweging uit Introduction to Data Engineering). Bronze wordt dan een continu appendende tabel die events opslaat zoals ze binnenkomen — via Kafka, Kinesis, of Databricks' Auto Loader — zonder enige transformatie, precies zoals bij batch. Silver draait dan als een micro-batch- of streaming-job die continu of elke paar minuten dedupliceert en valideert (Spark Structured Streaming en Snowflake's Dynamic Tables zijn hier twee platformspecifieke implementaties van). Gold blijft meestal periodiek — de meeste dashboards hebben geen sub-minuut-verse aggregaties nodig, zelfs als de onderliggende bronze/silver-lagen wel continu bijwerken, wat opnieuw de "kies batch tenzij een concrete eis anders vereist"-vuistregel uit hoofdstuk 1 bevestigt.

## Backfills en het Recht op Vergetelheid

Een technisch probleem dat nog niet aan bod kwam: als bronze per ontwerp onveranderlijk is, hoe verwerk je dan een AVG-verwijderverzoek van een klant die vraagt om al zijn data te laten wissen? De ruwe, onveranderlijke bronze-laag lijkt dit principieel tegen te spreken.

De gangbare oplossing is een van twee patronen. **Tombstoning** voegt een verwijderingsmarkering toe (een nieuwe bronze-rij die aangeeft "dit record is verwijderd op moment X") in plaats van de bestaande rij te wijzigen — bronze blijft technisch append-only, maar silver-transformaties respecteren de tombstone en sluiten het record uit van elke laag erna. **Crypto-shredding** is grondiger: persoonsgegevens worden bij het schrijven naar bronze versleuteld met een unieke sleutel per klant; een verwijderverzoek vernietigt simpelweg die sleutel, waarna de versleutelde data in bronze voor altijd onleesbaar blijft zonder dat de rij zelf ooit fysiek is gewijzigd — technisch nog steeds "append-only", functioneel onherstelbaar gewist. Voor de meeste organisaties is crypto-shredding de robuustere aanpak specifiek omdat het geen enkele uitzondering op de immutability-regel van bronze vereist.

## Wanneer Drie Lagen Niet Genoeg Zijn

Sommige teams introduceren een vierde laag, vaak "platinum" genoemd, voor sterk verrijkte of ML-feature-klare data die verder gaat dan een standaard gold-aggregatie — bijvoorbeeld voorberekende features voor een aanbevelingsmodel die zowel zware berekening als domeinspecifieke feature-engineering vereisen. Dit is geen teken dat het medallion-patroon tekortschiet, maar een natuurlijke uitbreiding ervan: dezelfde scheiding van verantwoordelijkheden, één laag verder doorgevoerd voor een specifiek gebruiksdoel. Voer dit alleen in wanneer een concrete behoefte (zoals ML-featurestores) dat rechtvaardigt — drie lagen dekken de overgrote meerderheid van platforms afdoende, en een ongebruikte vierde laag is vooral extra onderhoud zonder voordeel.

## Lineage Zichtbaar Maken over de Lagen Heen

Een vraag die zodra een platform groter wordt onvermijdelijk opduikt: "welke bronze-tabellen voeden uiteindelijk dit ene getal in het directiedashboard?" Zonder expliciete lineage is dit archeologie — het handmatig terugvolgen van elke join en elke `ref()` door tientallen modellen. dbt lost dit grotendeels automatisch op: omdat elk model zijn afhankelijkheden expliciet declareert via `ref()` in plaats van hardcoded tabelnamen, kan dbt de volledige dependency-graph genereren en visualiseren (`dbt docs generate`), van elke gold-tabel helemaal terug tot de bronze-bron. Op Databricks vervult Unity Catalog een vergelijkbare rol, automatisch afgeleid uit daadwerkelijk uitgevoerde queries in plaats van uit projectconfiguratie.

Dit is niet alleen handig voor debugging — het is een directe governance-vereiste zodra een organisatie moet kunnen aantonen waar een gerapporteerd cijfer vandaan komt (relevant voor financiële audits, en direct gekoppeld aan de lineage-verplichtingen uit Data Governance). Een praktische vuistregel: als je niet binnen enkele minuten met tooling — niet met tribale kennis van één senior engineer — kunt aantonen welke bronze-tabellen een specifieke gold-metric voeden, ontbreekt er een stuk volwassenheid in hoe de medallion-lagen zijn gedocumenteerd, ongeacht hoe correct de onderliggende SQL is.

## Best Practices

- **Handhaaf het contract per laag strikt**, ook onder tijdsdruk — een "snelle" businessregel die stiekem in staging sluipt, ondermijnt de herbouwbaarheid die de hele reden is om lagen te scheiden.
- **Materialiseer bewust per laag** (view voor dunne staging, table voor silver/gold), niet uniform — zie de materialization-tabel hierboven.
- **Concentreer teststypes per laag**: versheid in bronze, structuur in silver, business-logica in gold — in plaats van overal dezelfde generieke tests toe te passen.
- **Kies crypto-shredding boven tombstoning** voor persoonsgegevens waar mogelijk, omdat het geen uitzondering op bronze's immutability vereist.
- **Voer een vierde laag alleen in bij een concrete, benoembare behoefte** (zoals ML-featurestores), niet omdat "meer lagen netter klinkt".
- **Genereer en publiceer lineage-documentatie als onderdeel van de CI/CD-pipeline** (bijvoorbeeld `dbt docs generate` bij elke deploy), zodat lineage nooit veroudert ten opzichte van de daadwerkelijke modelstructuur.

## Veelgemaakte Fouten

- **Businesslogica die in bronze sluipt**, meestal als "tijdelijke" oplossing voor een dringende deadline die nooit meer wordt teruggedraaid — waardoor bronze zijn functie als betrouwbaar vangnet verliest.
- **Elke laag hetzelfde materialiseren** (bijvoorbeeld alles als `table`), wat onnodige opslag- en compute-kosten oplevert voor dunne staging-modellen die prima als view kunnen draaien.
- **Generieke tests overal toepassen** in plaats van teststrategie te concentreren op wat elke laag daadwerkelijk moet garanderen.
- **Persoonsgegevens direct in bronze laten staan zonder verwijderstrategie**, tot het moment dat het eerste AVG-verzoek binnenkomt en er geen ontworpen mechanisme is om het af te handelen.

- **Rechtstreeks vanuit gold-modellen naar bronze verwijzen** "voor een extra kolom die nog niet in silver zit" — een sluipweg die de datakwaliteitsgaranties van silver omzeilt.

## Performance Tips

- **Gebruik incrementele materialisatie in silver en gold** zodra tabellen groeien — volledige herberekening bij elke run wordt onbetaalbaar naarmate historie opbouwt, zoals eerder beargumenteerd in Introduction to Data Engineering.
- **Beperk het aantal kolommen dat door elke laag stroomt** tot wat daadwerkelijk nodig is verderop — een brede bronze-tabel hoeft niet elke kolom één-op-één door te geven aan silver als downstream niets ermee doet.
- **Plan silver-transformaties niet vaker dan nodig** — als gold toch maar eens per uur ververst, heeft een silver-laag die elke vijf minuten herberekent geen enkel voordeel, alleen extra compute-kosten.
- **Monitor de laadtijd per laag apart** — een trage bronze-naar-silver-stap wijst op een ander probleem (deduplicatielogica, ontbrekende clustering) dan een trage silver-naar-gold-stap (vaak join- of aggregatiecomplexiteit).

## Interviewvragen

**"Wat mag wel en niet in de bronze-laag gebeuren?"** Let op: alleen ongewijzigd opslaan plus technische metadata; nooit businesslogica, filtering op inhoud, of updates/verwijderingen (behalve via een expliciet ontworpen mechanisme zoals tombstoning).

**"Hoe vertaalt bronze/silver/gold zich naar een dbt-projectstructuur?"** Let op: staging (dun, dicht bij bronze), intermediate (silver — deduplicatie, combinatie), marts (gold — business-klaar, vaak per domein) — en het besef dat deze indeling dbt's eigen conventie is, niet een verplicht onderdeel van medallion architecture zelf.

**"Waarom zou je bronze als view materialiseren maar gold als table?"** Let op: staging-modellen zijn dun genoeg dat herberekenen bij elke query goedkoop is; gold-tabellen worden vaak en direct door BI-tools bevraagd en moeten voorspelbaar snel zijn, wat een gematerialiseerde tabel garandeert en een view niet.

**"Hoe verwerk je een AVG-verwijderverzoek als bronze onveranderlijk hoort te zijn?"** Let op: tombstoning (een verwijdermarkering toevoegen die latere lagen respecteren) of crypto-shredding (versleutelde opslag waarvan de sleutel wordt vernietigd) — beide zonder de append-only-garantie van bronze te breken.

**"Hoe pas je medallion architecture toe op een streaming-bron?"** Let op: bronze als continu appendende ruwe stream, silver als (micro-)batch-deduplicatie, gold meestal nog steeds periodiek — niet elke laag hoeft even vers te zijn als de bron zelf binnenkomt.

**"Wanneer zou je een vierde laag (platinum) toevoegen aan medallion architecture?"** Let op: alleen bij een concrete behoefte zoals ML-featurestores die verder gaan dan standaard business-aggregaties — niet standaard, en het besef dat drie lagen de meerderheid van platforms afdoende dekken.

## Relevante Documentatie

- Databricks: The Medallion Architecture — de oorspronkelijke, meest geciteerde beschrijving van het patroon.
- dbt Labs: How We Structure Our dbt Projects — de staging/intermediate/marts-conventie in detail.
- Snowflake: Data Governance and Schema Design — hoe schema-scheiding en toegangscontrole in de praktijk samenkomen.

## Samenvatting

Medallion architecture is meer dan drie namen voor drie mate van "schoonheid" — het is een strikt contract over wat elke laag wel en niet mag doen, met directe consequenties voor hoe je namespaces inricht, hoe je materialiseert, en hoe je test. dbt's eigen staging/intermediate/marts-conventie vertaalt dit contract naar een concrete projectstructuur; materialization- en teststrategie horen per laag te verschillen, niet uniform te zijn. Het patroon werkt met aanpassingen ook voor streaming-bronnen, en zelfs het

ogenschijnlijk tegenstrijdige probleem van AVG-verwijdering in een onveranderlijke bronze-laag heeft ontworpen oplossingen (tombstoning, crypto-shredding) die de kernprincipes intact laten. Met dit fundament — modellering, transformatiepatronen, en nu de architectuur die ze samenbrengt — is het tijd voor het volgende hoofdstuk: dbt zelf, het gereedschap waarmee de meeste van deze patronen in de praktijk worden gebouwd.

HOOFDSTUK 9

# dbt Fundamenten

## Introductie

dbt is in vrijwel elk voorgaand hoofdstuk als transformatietool genoemd — voor medallion architecture, voor SCD Type 2 via snapshots, als het ELT-gereedschap bij uitstek. Dit hoofdstuk behandelt dbt als tool zelf, in de diepte die eerdere hoofdstukken bewust oversloegen: wat dbt precies wel en niet doet, hoe Jinja en macro's SQL herbruikbaar maken, hoe de dependency graph exact wordt opgebouwd, hoe incrementele modellen technisch werken, en het verschil tussen de twee soorten tests die dbt aanbiedt.

## Wat dbt Precies Is (en Wat het Niet Is)

De meest voorkomende misvatting bij mensen die net met dbt beginnen: dbt is geen database, geen orchestrator, en verplaatst zelf geen data tussen systemen. **dbt is een compiler.** Je schrijft SQL met Jinja-templating erin; dbt compileert dat naar pure, uitvoerbare SQL en stuurt die naar je warehouse om uit te voeren. Het warehouse doet al het rekenwerk — dbt orkestreert alleen *welke* SQL-statements in *welke volgorde* worden uitgevoerd, gebaseerd op de afhankelijkheden die jij declareert.

Dit verklaart meteen waarom dbt zo goed past bij het ELT-patroon uit ETL vs ELT: omdat dbt zelf geen compute levert, profiteert het volledig van de compute die het warehouse toch al biedt, zonder een aparte verwerkingslaag te introduceren. Het verklaart ook waarom dbt niets doet aan extractie of laden — dat is het domein van Fivetran, Airbyte, of custom scripts; dbt begint pas zodra data al, ruw, in je warehouse staat.

## De Kern-Bouwstenen

**Models** zijn `.sql`-bestanden met een `SELECT`-statement — de basiseenheid van transformatie. dbt compileert elk model naar een `CREATE TABLE` of `CREATE VIEW`-statement, afhankelijk van de materialization.

**Sources** zijn declaraties van externe, ruwe tabellen die dbt niet zelf beheert — meestal je bronze-laag, geladen door een los ingestion-proces. Door bronnen expliciet te declareren in `sources.yml` in plaats van tabelnamen hard te coderen, kan dbt ook freshness-checks op die bronnen uitvoeren.

**Seeds** zijn kleine, statische CSV-bestanden die je als tabel wilt laden — een mapping van landcodes naar regio's, een lijst met feestdagen. Niet bedoeld voor operationele data, wel handig voor kleine referentietabellen die zelden wijzigen en onder versiebeheer horen.

**Snapshots** — al aangehaald in Data Vault 2.0 als dbt's ingebouwde mechanisme voor SCD Type 2 — leggen periodiek de staat van een brontabel vast en bouwen daar automatisch `valid_from`/`valid_to`-historie van op, zonder dat je zelf `MERGE`-logica hoeft te schrijven.

```
-- models/staging/stg_orders.sql — een model verwijst naar een source, niet naar een tabelnaam
SELECT
    id AS order_id,
    customer_id,
    CAST(order_date AS DATE) AS order_date,
    amount
FROM {{ source('raw', 'orders') }}
# models/staging/sources.yml — de source expliciet gedeclareerd, met freshness-check
sources:
  - name: raw
    tables:
      - name: orders
        loaded_at_field: _ingested_at
        freshness:
          warn_after: {count: 12, period: hour}
          error_after: {count: 24, period: hour}
```

## Jinja & Macro's: Waarom SQL Alleen Niet Genoeg Is

Puur SQL heeft geen manier om logica te herhalen zonder te copy-pasten. dbt lost dit op door elk model door een Jinja-templating-engine te halen vóórdat het naar het warehouse gaat — dezelfde templating-taal die achter veel Python-webframeworks zit. Dit opent de deur naar **macro's**: herbruikbare, parametriseerbare stukken SQL-logica, functioneel vergelijkbaar met een functie in een programmeertaal.

```
-- macros/cents_to_euros.sql — een herbruikbare macro
{% macro cents_to_euros(column_name) %}
    ROUND({{ column_name }} / 100.0, 2)
{% endmacro %}
-- gebruik in een model — wordt bij compile-time uitgeschreven naar pure SQL
SELECT
    order_id,
    {{ cents_to_euros('amount_cents') }} AS amount_euros
FROM {{ ref('stg_orders') }}
```

Dit compileert naar exact `ROUND(amount_cents / 100.0, 2) AS amount_euros` — de macro voegt geen runtime-overhead toe, het is puur een manier om dezelfde SQL-uitdrukking op meerdere plekken consistent te hergebruiken, zodat een wijziging in de rekenregel maar op één plek hoeft te gebeuren. Grotere macro's kunnen loops (`{% for %}`), conditionele logica (`{% if %}`) en zelfs dynamisch kolommen genereren bevatten — krachtig, maar met de waarschuwing dat te veel Jinja-magie een model net zo onleesbaar maakt als de GUI-tools die dbt juist probeerde te vervangen.

## De Dependency Graph in Detail: ref() en source()

`ref()` en `source()` zijn de twee functies die de dependency graph (DAG) van een dbt-project vormen, en het is de moeite waard precies te begrijpen wat ze doen. `{{ ref('stg_orders') }}` verwijst niet naar een letterlijke tabelnaam — het verwijst naar het *model* `stg_orders`, en dbt vertaalt dat op compile-time naar de daadwerkelijke, volledig gekwalificeerde tabelnaam in de doelomgeving (`dev_jan.stg_orders` in een developer-sandbox, `analytics.stg_orders` in productie). Dit is precies wat een volgende sectie over environments mogelijk maakt: hetzelfde model-bestand, andere gecompileerde tabelnaam, zonder één regel code te wijzigen.

Belangrijker: omdat elk model expliciet declareert welke andere modellen en sources het gebruikt via `ref()`/`source()`, kan dbt bij elke run automatisch de juiste uitvoeringsvolgorde afleiden — een model dat `{{ ref('stg_orders') }}` bevat, wordt gegarandeerd pas gebouwd nadat `stg_orders` zelf gebouwd is, zonder dat jij ooit een expliciete volgorde hoeft te specificeren zoals in een handmatig onderhouden Airflow-DAG. Dit is het mechanisme achter de "dbt leidt de dependency graph automatisch af"-uitspraak uit Modern Data Platform Architecture — nu concreet: het is puur statische analyse van welke `ref()`/`source()`-aanroepen in de gecompileerde SQL van elk model voorkomen.

## Incrementele Modellen in Diepte

Een `materialized='table'`-model herberekent bij elke run de volledige tabel vanaf nul — correct, maar onbetaalbaar zodra een tabel miljarden rijen bevat. Een `materialized='incremental'`-model verwerkt bij elke run alleen nieuwe of gewijzigde rijen, via het `is_incremental()`-macro dat dbt automatisch beschikbaar stelt:

```
-- models/marts/fct_orders.sql
{{ config(
    materialized='incremental',
    unique_key='order_id',
    incremental_strategy='merge'
) }}

SELECT order_id, customer_id, order_date, amount
FROM {{ ref('stg_orders') }}

{% if is_incremental() %}
  -- deze WHERE-clausule bestaat alleen bij incrementele runs, niet bij een full-refresh
  WHERE order_date > (SELECT MAX(order_date) FROM {{ this }})
{% endif %}
```

`{{ this }}` verwijst naar de tabel die het model zelf produceert — hier gebruikt om het watermark-patroon uit Introduction to Data Engineering direct in dbt te implementeren. De `incremental_strategy` bepaalt wat er met de nieuwe rijen gebeurt zodra ze zijn geselecteerd, en dit verschilt fundamenteel per strategie:

- **`merge`** — het standaardgedrag op de meeste moderne warehouses (Snowflake, Databricks, BigQuery): nieuwe rijen met een bestaande `unique_key` worden bijgewerkt, nieuwe worden ingevoegd — functioneel identiek aan de `MERGE`-patronen uit SQL for Data Engineers, maar door dbt gegenereerd.
- **`delete+insert`** — verwijdert eerst alle rijen die matchen op `unique_key`, voegt dan de nieuwe rijen in. Nuttig op platformen zonder native `MERGE`-ondersteuning, of wanneer een enkele `unique_key`-waarde meerdere rijen in de nieuwe batch vertegenwoordigt (`merge` staat dat niet toe).
- **`append`** — voegt simpelweg alle nieuwe rijen toe, zonder ooit te controleren op duplicaten. Alleen veilig wanneer je zeker weet dat de bron nooit dezelfde rij twee keer aanlevert — in de praktijk zeldzamer dan beginners verwachten.

Een incrementeel model kan altijd geforceerd volledig herbouwd worden met `dbt run --full-refresh` — essentieel na een wijziging in de transformatielogica zelf, omdat de incrementele `WHERE`-clausule anders alleen nieuwe rijen met de nieuwe logica zou verwerken, terwijl bestaande rijen de oude, inmiddels onjuiste berekening zouden behouden.

## Testing: Generic versus Singular Tests

dbt onderscheidt twee soorten tests, met een net iets andere toepassing dan de laag-specifieke teststrategie uit Medallion Architecture — dit gaat over de *vorm* van de test, niet over *welke laag* je test.

**Generic tests** zijn herbruikbare, parametriseerbare tests die je in `schema.yml` aanroept zonder zelf SQL te schrijven — `not_null`, `unique`, `relationships`, `accepted_values` zijn ingebouwd, en packages als `dbt_utils` en `dbt_expectations` voegen tientallen extra toe (bijvoorbeeld: controleer of een kolom altijd binnen een bepaalde range valt).

**Singular tests** zijn custom `.sql`-bestanden in de `tests/`-map die een specifieke, vaak business-specifieke bewering controleren — het bestand moet **nul rijen** teruggeven om te slagen; elke teruggegeven rij is een gefaalde test-case:

```
-- tests/assert_positive_revenue.sql — een singular test
-- Slaagt als deze query GEEN rijen teruggeeft
SELECT order_id, revenue
FROM {{ ref('fct_revenue') }}
WHERE revenue < 0
```

De vuistregel: gebruik generic tests voor structurele garanties (bestaat de rij, is de sleutel uniek, valt de waarde binnen een verwachte set) en singular tests voor businessregels die specifiek genoeg zijn dat geen generieke test ze kan uitdrukken — zoals "omzet mag nooit negatief zijn" of "het totaal van orderregels moet gelijk zijn aan het ordertotaal".

## Environments: Dev, Staging, Prod

Omdat `ref()` op compile-time wordt opgelost naar de juiste omgeving, kan hetzelfde dbt-project zonder wijziging tegen verschillende doelen draaien, geconfigureerd via `profiles.yml`:

```
# profiles.yml — dezelfde modellen, andere schema's per target
my_project:
  target: dev
  outputs:
    dev:
      type: snowflake
      schema: dev_jan
      threads: 4
    prod:
      type: snowflake
      schema: analytics
      threads: 8
```

Een engineer die lokaal ontwikkelt, draait tegen `dev` — een persoonlijk schema waar experimenteren geen risico vormt voor productiedata. CI/CD (zie CI/CD for Data Engineering) draait dezelfde modellen tegen `prod`. Geen enkel model-bestand hoeft te weten in welke omgeving het draait — dat is precies het punt.

## dbt Core versus dbt Cloud

**dbt Core** is de open-source command-line tool: je schrijft modellen, draait `dbt run`/`dbt test` lokaal of via je eigen orchestrator (Airflow, Dagster), en beheert alles zelf. **dbt Cloud** is de gehoste variant van dbt Labs: een web-IDE, ingebouwde scheduling, hosted documentatie, en een job-orchestrator specifiek voor dbt-runs — praktisch een managed laag bovenop dezelfde dbt Core-engine. De keuze is vergelijkbaar met de "beheer het zelf versus laat het managen"-afweging die overal in dit handbook terugkomt: dbt Core geeft volledige controle en geen licentiekosten, dbt Cloud bespaart operationele last in ruil voor een abonnement.

## Packages: Niet Alles Zelf Bouwen

dbt heeft een eigen packagesysteem (`packages.yml`), vergelijkbaar met pip voor Python of npm voor JavaScript, waarmee je herbruikbare macro's en tests van de community importeert in plaats van zelf te herschrijven. **dbt_utils** is het meest gebruikte pakket: datum-manipulatie, surrogate key-generatie (`dbt_utils.generate_surrogate_key`, relevant voor de hash keys uit Data Vault 2.0), en generieke tests zoals `dbt_utils.recency` die we eerder in Medallion Architecture gebruikten voor freshness-checks. **dbt_expectations** breidt dit uit met tientallen statistische en distributie-tests, geïnspireerd op het Python-framework Great Expectations. Voordat je een macro zelf schrijft voor iets generieks (surrogate keys, datumreeksen genereren, snelle string-manipulatie), is de kans groot dat dbt_utils het al oplost — zelf het wiel opnieuw uitvinden is hier zelden de moeite waard.

```
# packages.yml
packages:
  - package: dbt-labs/dbt_utils
    version: [">=1.0.0", "<2.0.0"]
```

## Best Practices

- **Behandel elk model als een pure functie van zijn `ref()`/`source()`-inputs** — geen verborgen afhankelijkheden buiten wat expliciet gedeclareerd is, anders breekt de automatisch afgeleide uitvoeringsvolgorde.
- **Gebruik `merge` als standaard incremental_strategy**, en wijk daar alleen bewust van af (delete+insert, append) met een concrete reden.
- **Test structurele garanties generiek, businessregels singulier** — probeer geen complexe businesslogica in een generic test te proppen die daar niet voor bedoeld is.
- **Documenteer wanneer een `--full-refresh` verplicht is** (na wijziging van incrementele logica) in de PR-conventie van je team, zodat dit niet wordt vergeten.
- **Houd macro's leesbaar** — een macro met drie geneste `{% if %}`-blokken is vaak minder onderhoudbaar dan drie aparte, iets minder DRY modellen.

## Veelgemaakte Fouten

- **Tabelnamen hardcoderen in plaats van `ref()`/`source()` te gebruiken** — werkt in dev, breekt zodra hetzelfde model in een andere omgeving met andere schema-namen draait.
- **`append` gebruiken als incremental_strategy zonder te verifiëren dat de bron nooit duplicaten aanlevert** — leidt tot langzaam groeiende, onopgemerkte duplicatie.
- **Vergeten een `--full-refresh` te draaien na een wijziging in incrementele logica**, waardoor oude en nieuwe rijen met inconsistente berekeningen naast elkaar blijven bestaan.

- **Te veel logica in Jinja-macro's stoppen** in plaats van in leesbare SQL, wat modellen onbegrijpelijk maakt voor iemand die geen Jinja-expert is.
- **Singular tests gebruiken voor iets dat een generic test met een package als dbt_utils al had kunnen afdekken** — onnodig maatwerk voor een opgelost probleem.

## Performance Tips

- **Gebruik incrementele modellen zodra een tabel merkbaar traag wordt om volledig te herbouwen** — er is geen vaste rijgrens, meet de daadwerkelijke build-tijd.
- **Beperk `threads` in `profiles.yml` niet onnodig laag** — dbt bouwt onafhankelijke modellen parallel; te weinig threads laat warehouse-compute onbenut.
- **Vermijd macro's die per rij worden geëvalueerd** in plaats van eenmalig bij compile-time — Jinja draait vóór de SQL naar het warehouse gaat, dus complexe macro-logica kost geen runtime-overhead, maar een macro die per ongeluk N losse queries genereert in plaats van één wel.
- **Gebruik `dbt build` in plaats van los `dbt run` + `dbt test`** in CI/CD — het bouwt en test elk model direct na elkaar in dependency-volgorde, zodat een gefaalde test een downstream-model kan blokkeren vóór het draait, in plaats van pas achteraf te ontdekken dat het op foute data is gebouwd.

## Interviewvragen

**"Wat doet dbt precies, en wat doet het niet?"** Let op: dbt compileert SQL met Jinja-templating en orkestreert de uitvoeringsvolgorde; het levert zelf geen compute en doet niets aan extractie of laden.

**"Wat is het verschil tussen `ref()` en `source()`?"** Let op: `ref()` verwijst naar een ander dbt-model (intern beheerd), `source()` verwijst naar een externe, ruwe tabel die dbt niet zelf bouwt — beide dragen bij aan de automatisch afgeleide dependency graph.

**"Leg de drie incremental_strategy-opties uit: merge, delete+insert, append."** Let op: merge voor update-of-insert op een unique key (het standaardgeval), delete+insert wanneer een unique key meerdere rijen per batch vertegenwoordigt, append alleen wanneer duplicatie gegarandeerd niet kan optreden.

**"Wanneer zou je een `--full-refresh` moeten draaien op een incrementeel model?"** Let op: na elke wijziging in de transformatielogica van het model zelf, omdat de incrementele `WHERE`-clausule anders alleen nieuwe rijen met de bijgewerkte logica zou verwerken.

**"Wat is het verschil tussen een generic en een singular test in dbt?"** Let op: generic tests zijn herbruikbaar en parametriseerbaar via schema.yml (not_null, unique, relationships); singular tests zijn custom SQL-bestanden voor specifieke businessregels, die slagen als ze nul rijen teruggeven.

**"Hoe zorgt dbt ervoor dat hetzelfde project tegen zowel een dev- als een productieomgeving kan draaien zonder codewijziging?"** Let op: `ref()` wordt op compile-time opgelost naar de juiste, volledig gekwalificeerde tabelnaam op basis van het actieve `target` in `profiles.yml` — modellen zelf bevatten nooit een hardcoded schema.

## Relevante Documentatie

- dbt Labs: Incremental Models — officiële uitleg van alle incremental_strategy-opties per adapter.
- dbt Labs: Jinja and Macros — de volledige Jinja-syntax die dbt ondersteunt.
- dbt Labs: Testing — generic versus singular tests in detail.
- dbt Labs: About dbt Core setup — het verschil tussen dbt Core en dbt Cloud.

## Samenvatting

dbt is een SQL-compiler met een dependency-resolutiemechanisme, geen database en geen orchestrator — het genereert en ordent SQL, het warehouse voert het uit. Jinja en macro's maken SQL herbruikbaar op een manier die pure SQL niet kan; `ref()` en `source()` bouwen automatisch de dependency graph op basis van statische analyse van elk model. Incrementele modellen, met de juiste `incremental_strategy`, zijn de sleutel tot betaalbare verwerking naarmate tabellen groeien, en de twee testsoorten (generic voor structuur, singular voor businesslogica) dekken samen de teststrategie die in het vorige hoofdstuk per laag werd uitgewerkt. Met dbt als gereedschap in de vingers, is het tijd om de platforms te bekijken waar dit gereedschap tegen draait — te beginnen met Snowflake in het volgende hoofdstuk.

HOOFDSTUK 10

# Snowflake

## Introductie

Snowflake is in eerdere hoofdstukken al meermaals aangehaald — micro-partities en MPP in Data Warehousing Fundamentals, warehouse-sizing en auto-suspend in Modern Data Platform Architecture, Dynamic Tables als declaratief alternatief voor dbt. Dit hoofdstuk herhaalt die basis niet, maar behandelt wat Snowflake echt onderscheidt van andere platformen: multi-cluster warehouses voor concurrency, Time Travel en zero-copy cloning, de manier waarop het semi-gestructureerde data behandelt, en hoe Dynamic Tables mechanisch werken onder de motorkap.

## Virtual Warehouses: Voorbij Sizing Alleen

Modern Data Platform Architecture introduceerde al waarom je meerdere warehouses per workload wilt (BI versus batch) en waarom auto-suspend kosten drukt. Twee aspecten die daar nog niet aan bod kwamen, zijn cruciaal om Snowflake goed te gebruiken.

**T-shirt sizing is exponentieel, niet lineair.** Elke stap omhoog (XS → S → M → L → XL, tot 6XL) verdubbelt zowel de rekenkracht als het creditverbruik per uur. Een M-warehouse kost dus 4× zoveel credits per uur als een XS, niet 2×. Dit maakt overdimensioneren duur op een manier die niet intuïtief is als je alleen naar de naamgeving kijkt — de sprong van M naar L "klinkt" als een kleine stap, maar verdubbelt de kosten opnieuw.

**Multi-cluster warehouses lossen een ander probleem op dan grootte: concurrency.** Een groter warehouse (hogere T-shirt-maat) maakt individuele queries sneller door meer parallelle rekenkracht per query. Een multi-cluster warehouse (`MIN_CLUSTER_COUNT`/`MAX_CLUSTER_COUNT`) lost op wat er gebeurt als te veel gebruikers *tegelijk* queries indienen — in plaats van dat queries in een wachtrij komen te staan, start Snowflake automatisch een extra cluster van dezelfde grootte op om de piekbelasting op te vangen, en schaalt weer terug zodra de drukte voorbij is:

```
CREATE WAREHOUSE bi_reporting
  WAREHOUSE_SIZE = 'SMALL'
  MIN_CLUSTER_COUNT = 1
  MAX_CLUSTER_COUNT = 4      -- schaalt automatisch tot 4 clusters bij piekdrukte
  SCALING_POLICY = 'STANDARD'
  AUTO_SUSPEND = 60
  AUTO_RESUME = TRUE;
```

De vuistregel: als queries traag zijn omdat ze individueel te veel data verwerken, vergroot het warehouse (verticaal schalen). Als queries traag zijn omdat te veel gebruikers gelijktijdig hetzelfde, kleinere warehouse belasten, voeg je clusters toe (horizontaal schalen) — twee verschillende knoppen voor twee verschillende problemen, vaak verward.

## Clustering Keys: Hoe Pruning Echt Werkt

Data Warehousing Fundamentals introduceerde micro-partities; hier de praktische toepassing. Snowflake verdeelt elke tabel automatisch in micro-partities van 50-500MB (gecomprimeerd), en houdt voor elke partitie metadata bij over de min/max-waarden per kolom. Een query met `WHERE order_date = '2026-08-14'` kan dan hele partities overslaan zonder ze te lezen, simpelweg omdat de metadata al aangeeft dat die partitie geen rijen met die datum bevat — dit heet **pruning**, en het gebeurt volledig automatisch, zonder dat je er iets voor hoeft te configureren.

Het probleem ontstaat wanneer data niet van nature gesorteerd binnenkomt op de kolom waarop je meestal filtert. Als rijen in willekeurige volgorde worden ingeladen, bevat vrijwel elke micro-partitie een beetje van elke datum, waardoor pruning niets oplevert — elke query moet dan alsnog vrijwel alle partities lezen. Een expliciete **clustering key** vertelt Snowflake om data periodiek te herordenen zodat gerelateerde waarden (dezelfde datum, dezelfde klant) fysiek bij elkaar in dezelfde partities terechtkomen:

```
ALTER TABLE fact_sales CLUSTER BY (order_date);

-- Clustering-kwaliteit inspecteren
SELECT SYSTEM$CLUSTERING_INFORMATION('fact_sales', '(order_date)');
```

Reclustering kost zelf credits (Snowflake doet dit automatisch op de achtergrond zodra clustering-kwaliteit afneemt), dus het is geen gratis knop — de vuistregel is: overweeg een expliciete clustering key pas bij tabellen boven enkele honderden GB's waar querypatronen consistent op dezelfde kolom filteren, niet bij elke tabel standaard.

## Time Travel en Zero-Copy Cloning

Dit zijn twee functies zonder directe tegenhanger bij de meeste concurrenten, en beide zijn het waard om standaard in je workflow op te nemen.

**Time Travel** laat je data bevragen zoals die er op een eerder moment uitzag, tot 90 dagen terug (afhankelijk van je Snowflake-edition), zonder dat je zelf snapshots hoefde te maken:

```
-- Query de tabel zoals die er gisteren om deze tijd uitzag
SELECT * FROM fact_sales AT (OFFSET => -86400);

-- Of herstel een per ongeluk gewiste tabel binnen de retentieperiode
CREATE TABLE fact_sales_recovered CLONE fact_sales
  AT (TIMESTAMP => '2026-08-13 09:00:00'::TIMESTAMP);
```

Dit is direct bruikbaar als vangnet tegen het klassieke "iemand draaide een `DELETE` zonder `WHERE`"-scenario — geen aparte back-uprestore-procedure nodig, gewoon een query.

**Zero-copy cloning** maakt een volledige, onafhankelijke kopie van een tabel, schema of zelfs een hele database, ogenblikkelijk en zonder extra opslagkosten totdat er daadwerkelijk wijzigingen op de kloon plaatsvinden (copy-on-write):

```
-- Een volledige kopie van productiedata voor een dev-omgeving, in seconden, vrijwel gratis
CREATE DATABASE dev_jan CLONE analytics_prod;
```

Dit verandert de economie van dev/staging-omgevingen fundamenteel: in plaats van een verouderde, handmatig ververste testset te onderhouden, kan elke engineer op elk moment een verse, volledige kloon van productiedata maken om lokaal tegen te ontwikkelen — relevant voor de `dev`/`prod`-scheiding uit dbt Fundamentals, nu met een concreet mechanisme om dev-data actueel te houden zonder een aparte ETL-stap.

## Semi-gestructureerde Data: VARIANT en FLATTEN

Niet alle brondata is netjes tabulair — API-responses en event-tracking leveren vaak geneste JSON. Snowflake behandelt dit via het `VARIANT`-type, dat elke JSON-structuur ongewijzigd kan opslaan terwijl je er nog steeds relationeel op kunt filteren en indexeren, zonder vooraf een strikt schema te hoeven definiëren:

```
CREATE TABLE raw_events (
    event_id STRING,
    payload VARIANT      -- ruwe, geneste JSON
);

-- Dot-notatie om direct in de JSON-structuur te filteren
SELECT
    event_id,
    payload:user_id::STRING AS user_id,
    payload:properties:plan::STRING AS plan_name
FROM raw_events
WHERE payload:event_type::STRING = 'subscription_upgraded';

-- FLATTEN om een geneste array uit te klappen naar losse rijen
SELECT
    e.event_id,
    item.value:sku::STRING AS product_sku
FROM raw_events e,
LATERAL FLATTEN(input => e.payload:items) item;
```

Dit is precies de bronze-laag-behandeling voor semi-gestructureerde bronnen uit Medallion Architecture in de praktijk: JSON landt ongewijzigd als `VARIANT` in bronze, en de silver-transformatie parseert met dot-notatie en `FLATTEN` de specifieke velden die je nodig hebt naar een nette, tabulaire vorm.

## Dynamic Tables: TARGET_LAG Mechanisch Uitgelegd

Eerdere hoofdstukken noemden Dynamic Tables als Snowflake's declaratieve alternatief voor handmatig geplande dbt-runs — hier de mechaniek. Je declareert een query en een `TARGET_LAG`; Snowflake berekent zelf, op basis van hoe lang de query duurt en hoe vaak de brondata verandert, hoe vaak hij moet verversen om dat doel te halen — en bij een keten van afhankelijke Dynamic Tables rekent Snowflake automatisch terug hoe vers elke laag moet zijn om de eindlaag op tijd te halen:

```
CREATE DYNAMIC TABLE silver_orders
  TARGET_LAG = '15 minutes'
  WAREHOUSE = transform_wh
  AS
SELECT
    order_id, customer_id, CAST(order_date AS DATE) AS order_date, amount
FROM raw_orders
QUALIFY ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY _ingested_at DESC) = 1;
```

Het verschil met een incrementeel dbt-model op een schema (bijvoorbeeld elk uur via een orchestrator getriggerd) is dat een Dynamic Table zelf beslist wanneer verversen nodig is — bij weinig brondatawijzigingen ververst hij minder vaak dan het `TARGET_LAG` toestaat, wat compute bespaart, terwijl een vast dbt-schema altijd op tijd draait ongeacht of er daadwerkelijk iets veranderd is. De keerzijde is minder directe controle: je stuurt een doel, niet een exact schema.

## Snowpark: Python Naast SQL

Snowflake was oorspronkelijk een puur SQL-platform, maar **Snowpark** brengt Python (en Java/Scala) rechtstreeks in het platform, uitgevoerd op dezelfde warehouse-compute in plaats van in een externe Python-omgeving die data heen en weer moet sturen. Dit is relevant voor precies het soort werk waar SQL minder natuurlijk voor is — complexere featureberekeningen voor een ML-model, of logica die beter als imperatieve code dan als declaratieve query past:

```
from snowflake.snowpark import Session
from snowflake.snowpark.functions import col, when

session = Session.builder.configs(connection_params).create()

orders_df = session.table("silver.orders")

enriched = orders_df.with_column(
    "order_size_segment",
    when(col("amount") > 1000, "large")
    .when(col("amount") > 100, "medium")
    .otherwise("small")
)

enriched.write.mode("overwrite").save_as_table("gold.orders_segmented")
```

Dit compileert en draait binnen Snowflake's eigen compute-omgeving, zonder data naar een externe Python-cluster te hoeven verplaatsen — een directe manier om de Python-vaardigheden uit Python for Data Engineers in te zetten zonder de architecturale eenvoud van "alles blijft in het warehouse" op te geven die Snowflake's belangrijkste aantrekkingskracht is.

## Secure Data Sharing

Een functie zonder directe parallel bij traditionele warehouses: Snowflake laat je data delen met een ander Snowflake-account — of zelfs publiceren op de Snowflake Marketplace — **zonder de data daadwerkelijk te kopiëren**. De ontvanger bevraagt een read-only view op jouw storage, real-time, zonder ETL-pijplijn ertussen:

```
CREATE SHARE sales_data_share;
GRANT USAGE ON DATABASE analytics TO SHARE sales_data_share;
GRANT SELECT ON TABLE analytics.gold.revenue_summary TO SHARE sales_data_share;
ALTER SHARE sales_data_share ADD ACCOUNTS = ('partner_account_identifier');
```

Voor organisaties die regelmatig data uitwisselen met partners, leveranciers of dochterondernemingen elimineert dit een hele categorie aan traditionele "exporteer, verstuur, importeer"-pijplijnen, inclusief het synchronisatieprobleem dat daarbij hoort — de ontvanger ziet altijd de actuele data, niet een export van gisteren.

## Kostenbeheersing met Resource Monitors

Het warehouse-sizingvoorbeeld in Modern Data Platform Architecture liet zien hoeveel een verkeerd beheerd warehouse kan kosten; **resource monitors** zijn het concrete Snowflake-mechanisme om dat automatisch te bewaken in plaats van achteraf de rekening te controleren. Een resource monitor bewaakt het creditverbruik van een of meerdere warehouses binnen een periode, en kan bij het bereiken van een drempel automatisch waarschuwen — of het warehouse hard stoppen:

```
CREATE RESOURCE MONITOR monthly_transform_budget
  WITH CREDIT_QUOTA = 500
  FREQUENCY = MONTHLY
  START_TIMESTAMP = IMMEDIATELY
  TRIGGERS
    ON 75 PERCENT DO NOTIFY
    ON 100 PERCENT DO SUSPEND;

ALTER WAREHOUSE nightly_batch_wh SET RESOURCE_MONITOR = monthly_transform_budget;
```

Dit soort harde grens is precies wat het verschil maakt tussen "we ontdekken een kostenprobleem aan het eind van de maand" en "het systeem voorkomt het probleem voordat het zich voordoet" — een concrete, automatiseerbare invulling van de kostengovernance-principes uit Modern Data Platform Architecture.

## Best Practices

- **Scheid warehouses per workload en gebruik multi-cluster alleen voor concurrency-problemen**, niet als eerste antwoord op trage individuele queries — grotere warehouse-sizing lost dat op.
- **Overweeg clustering keys pas bij tabellen die daadwerkelijk groot genoeg zijn** om ervan te profiteren; op kleinere tabellen kost reclustering meer credits dan het oplevert.
- **Gebruik zero-copy clones structureel voor dev/staging-omgevingen** in plaats van verouderde, handmatig onderhouden testsets.
- **Bewaar ruwe JSON als VARIANT in bronze, parseer expliciet in silver** — niet andersom, om dezelfde reden dat je in Introduction to Data Engineering nooit vroegtijdig moet casten.
- **Zet resource monitors in** op elk warehouse of account om onverwachte creditverbruik-pieken automatisch te signaleren of te blokkeren vóórdat de rekening onaangenaam verrast.
- **Gebruik Snowpark voor logica die zich slecht in SQL laat uitdrukken**, maar val niet standaard terug op Python voor werk dat een simpele `SELECT` met een `CASE`-expressie net zo goed had opgelost.

## Veelgemaakte Fouten

- **Eén groot warehouse voor alle workloads gebruiken** in plaats van workloads te scheiden — precies de fout die het kostenvoorbeeld in Modern Data Platform Architecture illustreerde.
- **Clustering keys blindelings op elke grote tabel zetten** zonder te verifiëren dat querypatronen daadwerkelijk consistent op die kolom filteren — reclustering-kosten zonder queryvoordeel.
- **Time Travel-retentie verwarren met een back-upstrategie** — Time Travel beschermt tegen recente fouten binnen de retentieperiode, niet tegen langetermijnarchivering of rampenherstel buiten die periode.
- **`SELECT *` op VARIANT-kolommen zonder specifieke paden** in productiequeries, waardoor je de voordelen van kolomgeoriënteerde pruning misloopt die dot-notatie op specifieke JSON-paden wel biedt.

- **Geen resource monitors instellen**, waardoor een per ongeluk oneindig draaiende query of een verkeerd geconfigureerde multi-cluster-warehouse ongemerkt credits verbrandt.

## Performance Tips

- **Gebruik `SYSTEM$CLUSTERING_INFORMATION` om clustering-kwaliteit te meten** vóór je aanneemt dat een clustering key nodig is of nog effectief is.
- **Vermijd query's die `VARIANT`-kolommen volledig casten** (`payload::STRING`) wanneer je maar een paar velden nodig hebt — gebruik gerichte dot-notatie zodat Snowflake alleen de relevante sub-structuur hoeft te verwerken.
- **Monitor `TARGET_LAG` versus daadwerkelijke verversingsfrequentie** bij Dynamic Tables — een consistent te laag ingesteld doel dwingt onnodig frequente verversingen af.
- **Gebruik het Query Profile** (in Snowsight) bij elke onverwacht trage query vóór je gokt naar een oplossing — het toont exact welk deel van de query (scan, join, aggregatie) de tijd kost.

## Interviewvragen

**"Wat is het verschil tussen groter maken van een warehouse en multi-cluster warehousing?"** Let op: grootte (T-shirt-sizing) versnelt individuele queries via meer rekenkracht; multi-cluster lost concurrency op door extra clusters te starten bij gelijktijdige belasting — twee verschillende problemen.

**"Hoe werkt pruning in Snowflake, en wanneer zou je een expliciete clustering key toevoegen?"** Let op: automatische min/max-metadata per micro-partitie, clustering keys alleen zinvol bij grote tabellen met consistente filterpatronen op een specifieke kolom.

**"Wat is Time Travel, en waar zou je het voor gebruiken?"** Let op: bevragen of herstellen van data zoals die op een eerder moment was, binnen de retentieperiode — nuttig tegen recente fouten, geen vervanging voor langetermijnback-ups.

**"Leg zero-copy cloning uit, en waarom is het goedkoper dan een traditionele kopie."** Let op: copy-on-write — de kloon deelt aanvankelijk dezelfde onderliggende micro-partities als het origineel en kost pas opslag zodra er daadwerkelijk wijzigingen op plaatsvinden.

**"Hoe zou je geneste JSON-data verwerken in Snowflake?"** Let op: opslaan als VARIANT in bronze, dot-notatie en FLATTEN gebruiken om specifieke velden en geneste arrays te parsen naar een tabulaire silver-structuur.

**"Wat is een Dynamic Table, en hoe verschilt het van een incrementeel dbt-model op een vast schema?"** Let op: een Dynamic Table met TARGET_LAG bepaalt zelf de verversingsfrequentie op basis van daadwerkelijke brondatawijzigingen, terwijl een vast dbt-schema altijd op het ingestelde tijdstip draait, ongeacht of er iets veranderd is.

**"Hoe zou je onverwacht hoge Snowflake-kosten proactief voorkomen in plaats van achteraf ontdekken?"** Let op: resource monitors met credit-quota's en triggers (notify/suspend), gecombineerd met workload-scheiding per warehouse en bewuste sizing — niet pas reageren nadat de maandelijkse rekening al hoog bleek.

**"Wanneer zou je Snowpark gebruiken in plaats van standaard SQL?"** Let op: voor logica die zich beter leent voor imperatieve Python-code dan voor declaratieve SQL (complexere featureberekeningen, iteratieve logica), zonder de data buiten Snowflake's eigen compute te hoeven verplaatsen.

## Relevante Documentatie

- Snowflake: Virtual Warehouses — sizing, multi-cluster en auto-suspend in detail.
- Snowflake: Understanding Micro-partitions and Clustering — de officiële uitleg van pruning en clustering keys.
- Snowflake: Time Travel — retentieperiodes per edition en syntax.
- Snowflake: Dynamic Tables — TARGET_LAG-mechaniek en beperkingen.

## Samenvatting

Voorbij de basisconcepten die eerdere hoofdstukken al behandelden, onderscheidt Snowflake zich door een aantal specifieke functies: multi-cluster warehouses die concurrency oplossen los van query-snelheid, clustering keys die pruning bewust sturen wanneer data niet natuurlijk gesorteerd binnenkomt, en Time Travel plus zero-copy cloning die dev-workflows en incidentherstel fundamenteel vereenvoudigen ten opzichte van traditionele back-upstrategieën. VARIANT en FLATTEN maken semi-gestructureerde data een eersteklas burger in plaats van een uitzondering die eerst plat geslagen moet worden, en Dynamic Tables brengen declaratieve verversingslogica direct in het platform. Resource monitors geven een concreet, automatiseerbaar antwoord op de kostengovernance die eerder alleen conceptueel werd behandeld. In het volgende hoofdstuk verleggen we de aandacht naar Databricks — een platform met een andere set uitgangspunten, gebouwd rond Spark en het lakehouse-concept in plaats van een pure warehouse-architectuur.

HOOFDSTUK 11

# Databricks

## Introductie

Modern Data Platform Architecture vergeleek Databricks al met Snowflake en Fabric op workload-niveau: sterk voor ML/Python en complexe Spark-transformaties. Delta Lake, de tabellaag die ACID-garanties aan het lakehouse toevoegt, krijgt een eigen volledige behandeling in Delta Lake; PySpark zelf in PySpark. Dit hoofdstuk vult het gat ertussen in: de platformmechanica van Databricks zelf — hoe clusters werken, hoe Unity Catalog governance regelt, hoe Delta Live Tables declaratieve pipelines mogelijk maakt, en wat Photon en MLflow toevoegen.

## Drie Soorten Compute: Clusters en SQL Warehouses

Een veelvoorkomende bron van verwarring bij nieuwkomers: Databricks kent niet één, maar drie verschillende compute-vormen, elk voor een ander doel.

**All-purpose clusters** zijn bedoeld voor interactief, exploratief werk — een engineer die in een notebook stap voor stap data verkent, met de mogelijkheid om cellen los uit te voeren en tussentijds resultaten te bekijken. Ze blijven draaien totdat je ze expliciet stopt of een inactiviteitstimeout verstrijkt, en zijn relatief duur om continu te laten draaien.

**Job clusters** worden automatisch aangemaakt bij het starten van een geplande job, en **automatisch weer afgebroken zodra de job klaar is** — geen doorlopende kosten voor idle-tijd, in tegenstelling tot all-purpose clusters. Voor elke productiepipeline die op een schema draait (via Databricks Workflows) is dit de juiste keuze; het is functioneel vergelijkbaar met Snowflake's auto-suspend, maar dan als het standaardgedrag in plaats van een instelling die je expliciet aanzet.

**SQL warehouses** zijn een aparte compute-laag, specifiek geoptimaliseerd voor SQL-workloads vanuit BI-tools (Power BI, Tableau) of Databricks SQL zelf — met eigen auto-scaling en auto-stop, losstaand van de Spark-clusters die notebooks en jobs gebruiken. Ze bieden een warehouse-achtige ervaring (vergelijkbaar met een Snowflake virtual warehouse) bovenop dezelfde onderliggende lakehouse-data.

```
// job_cluster_config.json — ephemeral cluster, alleen actief tijdens de job
{
  "new_cluster": {
    "spark_version": "15.4.x-scala2.12",
    "node_type_id": "Standard_DS3_v2",
    "autoscale": { "min_workers": 2, "max_workers": 8 },
    "autotermination_minutes": 20
  }
}
```

De vuistregel: **all-purpose voor ontwikkelen, job clusters voor productie, SQL warehouses voor BI** — het door elkaar gebruiken (bijvoorbeeld een productiepipeline op een permanent draaiend all-purpose cluster) is de meest voorkomende, makkelijk te vermijden kostenfout op dit platform.

## Notebooks en Workflows: de Ontwikkelervaring

Databricks' ontwikkelervaring is bewust notebook-gedreven, wat een aantal concrete gewoontes met zich meebrengt die op andere platformen minder centraal staan. Een notebook combineert code-cellen (Python, SQL, Scala of R, zelfs gemengd binnen één notebook via `%sql`/`%python`-magic-commands) met markdown-cellen voor documentatie, en wordt cel voor cel interactief uitgevoerd — ideaal voor exploratie, maar met een addertje: de volgorde waarin cellen zijn *uitgevoerd* kan afwijken van de volgorde waarin ze in het notebook *staan*, wat reproduceerbaarheidsproblemen geeft als je niet consequent van boven naar beneden werkt.

`%run` laat een notebook een ander notebook aanroepen, functioneel vergelijkbaar met een Python `import`, en wordt vaak gebruikt om gedeelde hulpfuncties (connectielogica, herbruikbare transformaties) in een apart notebook te isoleren. **Widgets** (`dbutils.widgets`) maken een notebook parametriseerbaar — een datumparameter, een omgevingsnaam — zodat hetzelfde notebook zowel interactief als via een geplande job met andere inputs kan draaien, zonder de code zelf aan te passen.

**Databricks Workflows** orkestreert vervolgens meerdere notebooks of scripts als taken (*tasks*) in een DAG, met afhankelijkheden, retries en alerting — Databricks' eigen antwoord op orchestratie, functioneel overlappend met Airflow uit Introduction to Data Engineering maar dan native in het platform, zonder aparte infrastructuur:

```
// workflow_definition.json (vereenvoudigd)
{
  "name": "daily_orders_pipeline",
  "tasks": [
    { "task_key": "bronze_ingest", "notebook_task": { "notebook_path": "/pipelines/bronze_orders" } },
    { "task_key": "silver_transform", "depends_on": [{ "task_key": "bronze_ingest" }],
      "notebook_task": { "notebook_path": "/pipelines/silver_orders" } }
  ],
  "schedule": { "quartz_cron_expression": "0 0 3 * * ?", "timezone_id": "Europe/Amsterdam" }
}
```

## Databricks Repos: Git-integratie voor Notebooks

Notebooks zijn van nature lastig te versiebeheren — een los `.ipynb`-bestand met celuitvoer erin geeft rommelige, moeilijk leesbare Git-diffs. **Databricks Repos** koppelt een workspace-map direct aan een Git-repository, zodat notebooks als broncode (niet als uitvoerbundel) worden opgeslagen, met normale branches, commits en pull requests — de brug die notebook-gedreven ontwikkeling verenigbaar maakt met de CI/CD-principes uit CI/CD for Data Engineering, inclusief het draaien van Workflows tegen een specifieke branch voor pull-request-validatie vóórdat er naar `main` wordt gemerged.

## Instance Pools: Clusterstart Versnellen

Een operationeel detail dat de moeite waard is te kennen: het opstarten van een nieuw cluster (het aanvragen en provisionen van virtuele machines bij de onderliggende cloudprovider) kan enkele minuten duren — vervelend voor job clusters die je juist kort en efficiënt wilt laten draaien. **Instance pools** houden een voorraad al-opgestarte, maar ongebruikte virtuele machines paraat, zodat een nieuw cluster daaruit kan putten in seconden in plaats van minuten. Dit kost een beetje continue infrastructuur (de pool zelf), in ruil voor aanzienlijk snellere opstarttijden bij frequent draaiende job clusters — een afweging die vooral de moeite waard is bij pipelines die vaak, kort en op strikte tijdschema's draaien.

## Unity Catalog: het Drieledige Namespace

Unity Catalog is Databricks' governance-laag, en de kern ervan is een namespace-structuur met drie niveaus in plaats van de gebruikelijke twee: `catalog.schema.table` in plaats van alleen `schema.table`. Dit extra niveau bestaat specifiek om de medallion-scheiding uit Medallion Architecture fysiek af te dwingen op organisatieniveau, over meerdere workspaces heen:

```
-- Drie catalogi, elk met eigen toegangsrechten -- niet drie schema's binnen één catalogus
CREATE CATALOG bronze_catalog;
CREATE CATALOG silver_catalog;
CREATE CATALOG gold_catalog;

-- Rechten per laag, exact zoals de governance-principes uit Modern Data Platform Architecture
GRANT USAGE, SELECT ON CATALOG bronze_catalog TO `data-engineering-team`;
GRANT USAGE, SELECT ON CATALOG gold_catalog TO `analytics-team`;
```

Unity Catalog is ook waar Databricks automatische **lineage** aanbiedt — niet afgeleid uit projectconfiguratie zoals dbt's `ref()`-gebaseerde graph, maar uit daadwerkelijk uitgevoerde queries. Elke keer dat een query data uit tabel A leest en naar tabel B schrijft, registreert Unity Catalog die relatie automatisch, zichtbaar als graph in de UI — bruikbaar naast, niet in plaats van, de dbt-lineage uit Medallion Architecture wanneer dbt tegen Databricks draait.

## Delta Live Tables: Declaratieve Pipelines met Expectations

Waar Snowflake's Dynamic Tables (zie Snowflake) een `TARGET_LAG` declareren, declareert **Delta Live Tables (DLT)** een pipeline van afhankelijke tabellen mét ingebouwde datakwaliteitsregels, **expectations** genoemd — een direct antwoord op de teststrategie-per-laag uit Medallion Architecture, maar dan uitgevoerd tijdens het laden zelf in plaats van als losse test-stap achteraf:

```
import dlt
from pyspark.sql.functions import col

@dlt.table(comment="Ruwe orders, ongewijzigd geland")
def bronze_orders():
    return spark.readStream.format("cloudFiles").option("cloudFiles.format", "json").load("/raw/orders")

@dlt.table(comment="Schone, gededupliceerde orders")
@dlt.expect_or_drop("valid_order_id", "order_id IS NOT NULL")
@dlt.expect("positive_amount", "amount > 0")   # loggen, niet blokkeren
def silver_orders():
    return (
        dlt.read_stream("bronze_orders")
        .dropDuplicates(["order_id"])
        .withColumn("amount", col("amount").cast("decimal(10,2)"))
    )
```

`expect_or_drop` verwijdert rijen die niet aan de voorwaarde voldoen stilzwijgend uit de output (met telling in de UI); `expect` logt overtredingen zonder de rij te verwijderen; een derde variant, `expect_or_fail`, laat de hele pipeline-run falen bij een overtreding — drie niveaus van streng, gekozen per regel naargelang hoe kritiek die specifieke garantie is. Dit is precies de bronze/silver-teststrategie uit Medallion Architecture, maar dan native in het laadmechanisme verweven in plaats van als aparte `schema.yml`-tests erna.

## Photon: de Vectorized Query-engine

**Photon** is Databricks' native, in C++ herschreven query-executie-engine, een drop-in-vervanging voor de standaard JVM-gebaseerde Spark-engine voor SQL- en DataFrame-operaties. Het voordeel zit in **vectorized processing**: in plaats van rij voor rij te verwerken, verwerkt Photon batches van rijen tegelijk met CPU-instructies die daarvoor zijn geoptimaliseerd (SIMD), wat vooral bij aggregatie- en join-zware SQL-workloads een aanzienlijk snelheidsvoordeel oplevert ten opzichte van standaard Spark — vaak zonder dat je zelf iets aan je query hoeft te veranderen, simpelweg door Photon in te schakelen op het cluster. Het is geen vervanging voor goed queryontwerp (dezelfde principes uit SQL for Data Engineers blijven gelden), maar een engine-niveau versnelling die bovenop die principes komt.

## MLflow: Experimenttracking als Eerste-klas Burger

Waar Snowflake ML-workloads via Snowpark toegankelijk maakt zonder ze centraal te positioneren, is **MLflow** — oorspronkelijk door Databricks ontwikkeld, inmiddels open-source — diep geïntegreerd in het platform en direct relevant voor het grensvlak tussen data engineering en data science dat in AI for Data Engineers verder wordt uitgewerkt. MLflow legt automatisch experimenten vast: welke parameters, welke dataset-versie (vaak gekoppeld aan een specifieke Delta Lake-versie via Time Travel-achtige functionaliteit), welke metrics, en welk uiteindelijk model-artefact bij elke trainingsrun hoorden — waardoor "welk model draait er precies in productie, en met welke data is het getraind" een reproduceerbare vraag wordt in plaats van tribale kennis.

```
import mlflow

with mlflow.start_run():
    mlflow.log_param("max_depth", 5)
    model = train_model(training_df, max_depth=5)
    mlflow.log_metric("accuracy", evaluate(model, test_df))
    mlflow.sklearn.log_model(model, "churn_model")
```

Voor een data engineer die primair aan pipelines werkt, is het relevant om te weten dát dit bestaat en hoe het aansluit op Delta Lake-versies — de diepgaande ML-workflow zelf valt buiten de scope van dit handbook.

## Best Practices

- **Gebruik job clusters voor elke productiepipeline**, nooit een permanent draaiend all-purpose cluster — de kostenbesparing is direct en aanzienlijk.
- **Structureer Unity Catalog-catalogi rond medallion-lagen** (of een vergelijkbare governance-indeling), niet rond individuele teams, zodat toegangscontrole de datastroom volgt in plaats van de organisatiestructuur.
- **Kies het expectation-niveau per regel bewust** (`expect`, `expect_or_drop`, `expect_or_fail`) — niet alles verdient dezelfde striktheid.
- **Schakel Photon standaard in** op clusters die primair SQL/DataFrame-workloads draaien, tenzij een specifieke, zelden voorkomende operatie niet door Photon wordt ondersteund.
- **Gebruik MLflow vanaf de eerste trainingsrun**, niet pas zodra een model naar productie gaat — retroactief reproduceerbaarheid toevoegen aan oude experimenten is vaak niet meer mogelijk.
- **Koppel elke productienotebook aan Databricks Repos**, zodat wijzigingen via pull requests lopen in plaats van rechtstreeks in de workspace-UI bewerkt te worden.
- **Gebruik instance pools voor frequent draaiende job clusters** met strikte tijdvensters, waar de paar minuten clusterstarttijd anders een merkbaar deel van het beschikbare tijdvenster opsoupeert.

## Veelgemaakte Fouten

- **All-purpose clusters gebruiken voor geplande productiejobs** — een van de meest voorkomende en makkelijkst te verhelpen kostenlekken op dit platform.
- **Eén platte Unity Catalog-catalogus voor alles**, zonder onderscheid tussen bronze/silver/gold of tussen teams, waardoor toegangscontrole net zo grof wordt als geen governance.
- **Alle expectations op `expect_or_fail` zetten** uit voorzichtigheid, waardoor elke kleine datakwaliteitsafwijking de hele pipeline blokkeert in plaats van gericht te reageren naar ernst.
- **Photon uitschakelen "voor de zekerheid"** zonder te meten of het daadwerkelijk een compatibiliteitsprobleem veroorzaakt — in de meeste gevallen is het een zuivere performancewinst zonder nadeel.
- **MLflow overslaan tot het "nodig" lijkt**, waarna niemand meer kan reconstrueren met welke data en parameters een model in productie is getraind.
- **Notebooks rechtstreeks in de productieworkspace bewerken** zonder Repos/Git-koppeling, waardoor wijzigingen niet reviewbaar zijn en een fout direct productie raakt zonder pull-request-stap ertussen.
- **Widgets vergeten bij het herbruikbaar maken van een notebook**, waardoor omgevingsspecifieke waarden (datums, schema-namen) hardcoded in de cellen blijven staan in plaats van als parameter te worden meegegeven.

## Performance Tips

- **Zet `autotermination_minutes` laag op all-purpose clusters** die voor ontwikkelwerk worden gebruikt — vergeten clusters die dagenlang idle doordraaien zijn een stille kostenpost.
- **Gebruik autoscaling (`min_workers`/`max_workers`) in plaats van een vaste clustergrootte** voor jobs met wisselende datavolumes, zodat je niet permanent voor piekcapaciteit betaalt.
- **Monitor DLT-expectationstatistieken actief** — een geleidelijk oplopend percentage gedropte rijen is vaak een vroeg signaal van een verslechterende databron, lang voordat het een zichtbaar rapportageprobleem wordt.
- **Vergelijk clusterprestaties met en zonder Photon** op je zwaarste periodieke jobs — de winst is workload-afhankelijk, en meten is betrouwbaarder dan aannemen.
- **Zet instance pools in voor pipelines met een strak tijdvenster**, waar clusterstarttijd een merkbaar deel van de beschikbare verwerkingstijd anders zou opeisen.
- **Houd `%run`-ketens ondiep.** Een notebook dat via `%run` tien andere notebooks aanroept die op hun beurt weer andere aanroepen, wordt net zo moeilijk te doorgronden als een diep geneste macro-structuur in dbt.

## Interviewvragen

**"Wat is het verschil tussen een job cluster, een all-purpose cluster en een SQL warehouse?"** Let op: job clusters zijn ephemeral en gekoppeld aan een specifieke geplande run, all-purpose clusters blijven draaien voor interactief werk, SQL warehouses zijn een aparte, BI-geoptimaliseerde compute-laag — en het besef dat het door elkaar gebruiken ervan de meest voorkomende kostenfout is.

**"Waarom heeft Unity Catalog drie namespace-niveaus (catalog.schema.table) in plaats van twee?"** Let op: het extra catalogus-niveau maakt het mogelijk om medallion-lagen (of teams) fysiek te scheiden met eigen toegangsrechten, over meerdere workspaces heen.

**"Wat zijn expectations in Delta Live Tables, en welke drie niveaus van striktheid bestaan er?"** Let op: `expect` (loggen), `expect_or_drop` (rij verwijderen), `expect_or_fail` (hele pipeline laten falen) — met het besef dat de keuze per regel afhangt van hoe kritiek die specifieke garantie is.

**"Wat doet Photon, en wanneer levert het het meeste voordeel op?"** Let op: vectorized, CPU-geoptimaliseerde query-executie die vooral bij aggregatie- en join-zware SQL/DataFrame-workloads een merkbaar verschil maakt, zonder dat de query zelf hoeft te veranderen.

**"Waarom zou een data engineer iets van MLflow moeten weten, ook als hij zelf geen modellen bouwt?"** Let op: het besef dat MLflow experimenten koppelt aan specifieke datasetversies (vaak via Delta Lake), wat reproduceerbaarheid van modellen mogelijk maakt — relevant voor iedereen die de data levert waarop modellen trainen.

**"Wat is het reproduceerbaarheidsprobleem met notebooks, en hoe voorkom je het?"** Let op: cellen kunnen out-of-order worden uitgevoerd tijdens interactief werk, waardoor de zichtbare volgorde niet de daadwerkelijke uitvoeringsvolgorde hoeft te zijn — consequent van boven naar beneden werken (of "Run All" gebruiken vóór oplevering) voorkomt dit.

**"Wat is het voordeel van Databricks Repos ten opzichte van los notebookbeheer?"** Let op: notebooks worden als broncode (niet als uitvoerbundel met celresultaten) opgeslagen in Git, met normale branches en pull requests — noodzakelijk om notebook-ontwikkeling te combineren met CI/CD.

**"Wat lossen instance pools op, en wanneer zijn ze de moeite waard?"** Let op: ze houden vooraf opgestarte virtuele machines paraat om clusterstarttijd te verkorten van minuten naar seconden — vooral waardevol bij job clusters die frequent en kort draaien op een strak schema.

## Relevante Documentatie

- Databricks: Compute (Clusters) — het volledige onderscheid tussen cluster-types.
- Databricks: Unity Catalog — namespace-structuur, lineage en governance.
- Databricks: Delta Live Tables — expectations en pipeline-declaratie in detail.
- Databricks: Photon — ondersteunde workloads en activering.
- MLflow Documentation — experimenttracking, model registry en deployment.

## Samenvatting

Databricks' platformmechanica draait om drie herkenbare, elk-hun-eigen-doel compute-vormen (job clusters, all-purpose clusters, SQL warehouses), een drieledig Unity Catalog-namespace dat governance direct aan de medallion-indeling koppelt, en Delta Live Tables die datakwaliteitsregels — expectations — rechtstreeks in het laadproces verweven in plaats van als losse teststap. Photon versnelt SQL/DataFrame-workloads op engine-niveau zonder queryherschrijving, en MLflow maakt de brug naar machine learning reproduceerbaar door experimenten aan specifieke dataversies te koppelen. Notebooks als broncode via Databricks Repos en versnelde clusterstart via instance pools maken de operationele kant van het platform compleet. Met Snowflake en Databricks nu beide behandeld, richt het volgende hoofdstuk zich op het derde grote platform in dit handbook: Microsoft Fabric, met zijn eigen unified-platformbenadering.

HOOFDSTUK 12

# Microsoft Fabric

## Introductie

Modern Data Platform Architecture plaatste Fabric al naast Snowflake en Databricks: de sterkste keuze bij diepe Microsoft-ecosysteemintegratie. Dit hoofdstuk gaat voorbij die vergelijking en behandelt wat Fabric architectonisch fundamenteel anders maakt dan de andere twee platformen: één gedeelde opslaglaag (OneLake) onder meerdere gespecialiseerde workloads, in plaats van één enkele compute-engine met één opslagvorm.

## OneLake: "OneDrive voor Data"

Waar Snowflake en Databricks elk hun eigen opslagabstractie hanteren (Snowflake's interne storage, Databricks' Delta-tabellen op cloud storage), heeft Fabric daar één laag onder getrokken: **OneLake**, één enkele, tenant-brede opslaglaag waar letterlijk elk Fabric-workload — Lakehouse, Data Warehouse, KQL Database, Power BI-semantische modellen — zijn data in dezelfde onderliggende Delta Parquet-bestanden opslaat. Microsoft noemt dit zelf "OneDrive voor data": één account, één opslagplek, ongeacht welke tool je gebruikt om ernaar te kijken.

De praktische consequentie is groot: data die je in een Lakehouse landt, hoeft niet gekopieerd of geëxporteerd te worden om in een Data Warehouse of vanuit Power BI bevraagd te worden — hetzelfde fysieke Delta-bestand wordt door meerdere engines gelezen, elk met hun eigen sterkte (SQL-analytics in Warehouse, Spark-notebooks in Lakehouse, tijdreeksanalyse in KQL Database). Dit is een wezenlijk andere architectuur dan "één platform, één engine" — Fabric is bewust **één opslaglaag, meerdere gespecialiseerde engines**.

## Shortcuts: Data Verbinden Zonder te Kopiëren

**Shortcuts** zijn Fabric's mechanisme om data die *buiten* OneLake staat — in Azure Data Lake Storage, Amazon S3, of een ander Fabric-item — te ontsluiten alsof die zich lokaal in je Lakehouse bevindt, zonder de data daadwerkelijk te verplaatsen of te dupliceren:

```
Lakehouse "Sales"
├── Tables/
│   └── orders                    (echte, lokale Delta-tabel)
└── Shortcuts/
    └── external_weather_data     → wijst naar ADLS Gen2, elders in de organisatie
```

Een shortcut is een verwijzing, geen kopie — een query tegen `external_weather_data` leest live uit de brondata, wat betekent dat wijzigingen aan de bron direct zichtbaar zijn zonder een aparte syncpijplijn. Dit is vergelijkbaar in geest met Snowflake's Secure Data Sharing uit Snowflake (data ontsluiten zonder kopiëren), maar dan bewust breder toepasbaar: ook tussen verschillende Fabric-workspaces onderling, en ook naar cloudopslag buiten Microsoft's eigen ecosysteem. Voor organisaties met data verspreid over meerdere teams of zelfs meerdere clouds, vermijdt dit een hele categorie ETL-pijplijnen wiens enige doel is "kopieer data van A naar B zodat B het kan bevragen".

## Het Workload-Portfolio: Eén Opslag, Meerdere Motoren

Fabric's onderscheidende architectuurkeuze wordt het duidelijkst in hoe de verschillende workloads zich tot elkaar verhouden, allemaal bovenop dezelfde OneLake-data:

**Lakehouse** — de Spark-/notebook-gedreven werkomgeving, vergelijkbaar met een Databricks-lakehouse: ideaal voor de bronze/silver-transformatiestappen uit Medallion Architecture, met Python/PySpark-notebooks als primair gereedschap.

**Data Warehouse** — een volledig T-SQL-gebaseerde, warehouse-achtige ervaring bovenop diezelfde OneLake-data, voor teams die liever in SQL werken dan in notebooks — functioneel dichter bij Snowflake's ervaring, maar wijzend naar exact dezelfde onderliggende Delta-bestanden als het Lakehouse.

**KQL Database** — geoptimaliseerd voor tijdreeks- en logdata (Kusto Query Language, oorspronkelijk uit Azure Data Explorer), relevant voor observability-data en telemetrie op een schaal en snelheid waarvoor standaard SQL minder geschikt is — zie de raakvlakken met Monitoring & Observability.

**Power BI** — niet langer een los rapportagetool die data importeert, maar een workload die rechtstreeks tegen OneLake kan lezen (zie Direct Lake hieronder).

Het punt is niet dat elk team alle vier de workloads gebruikt — de meeste platforms gebruiken er twee of drie — maar dat de keuze tussen bijvoorbeeld Lakehouse en Data Warehouse een **ontwikkelervaring-keuze** is (notebooks versus SQL), niet een **datakopie-keuze**, omdat beide dezelfde OneLake-data delen.

## Direct Lake: Power BI Zonder Import of Refresh

Traditionele Power BI-modellen werken op een van twee manieren: **Import** (data wordt gekopieerd in een gecomprimeerd, in-memory model, snel te bevragen maar vereist periodieke refreshes die zelf tijd en compute kosten) of **DirectQuery** (elke visual stuurt een live query naar de bron, altijd actueel maar merkbaar trager per interactie). **Direct Lake** is Fabric's derde optie, en het genereert het meeste enthousiasme van de drie: Power BI leest de Delta-tabellen in OneLake **rechtstreeks in het eigen geheugengeoptimaliseerde formaat**, zonder de data eerst te hoeven importeren én zonder de latency-nadelen van DirectQuery.

De praktische consequentie: zodra een silver- of gold-tabel in OneLake wordt bijgewerkt (bijvoorbeeld door een nachtelijke Spark-transformatie), ziet een Direct Lake-rapport die wijziging vrijwel onmiddellijk, zonder dat er een aparte Power BI-refresh-taak hoeft te draaien of ingepland te worden — de traditionele "wanneer ververst het dashboard" vraag uit eerdere hoofdstukken vervalt voor dit specifieke pad grotendeels, omdat er geen aparte kopie meer bestaat die kan verouderen.

## Capacity-Based Billing: een Fundamenteel Ander Kostenmodel

Waar Snowflake credits per warehouse verbruikt en Databricks DBU's per cluster, koopt Fabric **capaciteit** (F-SKUs, bijvoorbeeld F64, F128) die wordt **gedeeld over alle workloads binnen een tenant** — Lakehouse-Spark-jobs, Warehouse-queries, Power BI-refreshes en Dataflows putten allemaal uit dezelfde capaciteitspool, in plaats van elk hun eigen, apart afgerekende compute te hebben.

Dit heeft een directe consequentie voor kostenbeheer die afwijkt van de per-warehouse-scheiding uit Modern Data Platform Architecture: in plaats van workloads te scheiden over aparte, individueel beheerde compute-eenheden, beheer je op Fabric vooral de **totale capaciteitsgrootte** en houd je in de gaten welke workload binnen die gedeelde pool het meeste verbruikt via de Capacity Metrics-app. Fabric biedt hier wel **smoothing** (piekgebruik uitgesmeerd over een venster in plaats van directe doorbelasting) en **bursting** (tijdelijk boven de capaciteitsgrens draaien, gecompenseerd door latere rustigere periodes) — mechanismen zonder directe tegenhanger bij de andere twee platformen, specifiek ontworpen voor een gedeeld-capaciteitsmodel.

## Mirroring: Zero-ETL Replicatie van Externe Databases

Een functie die direct voortbouwt op het shortcut-principe, maar dan voor operationele databases in plaats van bestandsopslag: **Mirroring** repliceert continu en vrijwel real-time de inhoud van een externe database — Azure SQL Database, Azure Cosmos DB, en zelfs Snowflake zelf — naar OneLake, in Delta-formaat, zonder dat je zelf een ingestion-pipeline hoeft te bouwen of te onderhouden. In de kern is dit een ingebouwde, beheerde change-data-capture-koppeling die de handmatige extractiestap uit Introduction to Data Engineering grotendeels overbodig maakt voor specifiek ondersteunde bronnen.

De praktische waarde zit in de combinatie met shortcuts en Direct Lake: een gemirrorde tabel verschijnt automatisch als bevraagbare Delta-tabel in OneLake, direct beschikbaar voor Spark-notebooks, SQL-warehousequeries én Direct Lake Power BI-rapporten — zonder ETL, zonder handmatige planning, zonder een aparte Fivetran- of Airbyte-koppeling. De beperking is dat Mirroring alleen werkt voor de

specifiek ondersteunde bronsystemen; voor alles daarbuiten blijft een reguliere ingestion-pipeline (Data Factory, of een externe tool) nodig.

## Dataflows Gen2: Low-Code Transformatie

Naast notebook- en SQL-gebaseerde transformatie biedt Fabric **Dataflows Gen2**, gebouwd op dezelfde Power Query-engine die achter Excel's "Data ophalen en transformeren" en Power BI's query-editor zit — een visuele, low-code interface voor transformaties, met de mogelijkheid om ook aangepaste M-code (Power Query's eigen taal) te schrijven voor complexere logica. Dit is functioneel het dichtst bij de klassieke ETL-tools uit ETL vs ELT, maar dan native in Fabric en schrijvend naar OneLake in plaats van naar een apart doelsysteem. Voor teams met sterke Power Query/Excel-vaardigheden maar minder SQL- of Python-ervaring is dit een toegankelijke ingang tot transformatie — met als afweging dat complexe, testbare transformatielogica zich doorgaans beter in versiebeheerde dbt-modellen of notebooks laat uitdrukken dan in een visuele canvas, om dezelfde reden als bij traditionele ETL-tools besproken in Introduction to Data Engineering.

## Codevoorbeeld: Shortcut en Notebook Samen

Een concreet voorbeeld van hoe de bouwstenen hierboven samenkomen: een shortcut ontsluit externe brondata, een notebook transformeert die naar silver, en de resulterende tabel is direct bruikbaar door elke andere workload op OneLake.

```
# Fabric-notebook (PySpark) — leest via een shortcut, schrijft naar silver
df_bronze = spark.read.format("delta").load("Tables/external_orders")  # shortcut-tabel

df_silver = (
    df_bronze
    .dropDuplicates(["order_id"])
    .withColumn("order_date", F.to_date("order_date"))
)

df_silver.write.format("delta").mode("overwrite").save("Tables/silver_orders")
# Vanaf hier: bevraagbaar via SQL Warehouse, Power BI (Direct Lake) en andere notebooks,
# zonder enige aparte kopieerstap.
// Data Factory-pipeline in Fabric: triggert het notebook op een schema
{
  "name": "silver_orders_pipeline",
  "activities": [
    { "name": "Run_Silver_Notebook", "type": "TridentNotebook",
      "notebook": { "referenceName": "transform_silver_orders" } }
  ],
  "triggers": [{ "type": "ScheduleTrigger", "recurrence": { "frequency": "Hour", "interval": 1 } }]
}
```

## Git-integratie op Workspace-niveau

Fabric ondersteunt het koppelen van een volledige workspace aan een Git-repository (Azure DevOps of GitHub), waarna items — notebooks, pipelines, Lakehouse-definities, rapporten — als broncode worden gesynchroniseerd in plaats van alleen binnen de Fabric-portal te bestaan. Dit is vergelijkbaar met Databricks Repos uit Databricks, maar dan toegepast op de bredere set Fabric-item-types, en het is de basis waarop een fatsoenlijke CI/CD-flow (zie CI/CD for Data Engineering) voor Fabric-items wordt gebouwd — zonder deze koppeling blijven wijzigingen beperkt tot handmatige export/import tussen workspaces.

## Best Practices

- **Gebruik shortcuts in plaats van kopieerpijplijnen** wanneer data al ergens anders in Delta/Parquet-vorm beschikbaar is — dit bespaart zowel opslag als synchronisatiecomplexiteit.
- **Kies de workload (Lakehouse/Warehouse) op basis van teamvaardigheden**, niet uit een aanname dat data gedupliceerd moet worden om beide te kunnen gebruiken — ze delen dezelfde OneLake-data toch al.

- **Zet Power BI-rapporten op Direct Lake waar mogelijk**, zeker voor dashboards op gold-tabellen die al periodiek door een pipeline worden ververst — een aparte Import-refresh is dan overbodige complexiteit.
- **Monitor capaciteitsverbruik via de Capacity Metrics-app**, niet per workload afzonderlijk, omdat het gedeelde model juist vraagt om zicht op de totale pool in plaats van geïsoleerde metingen.
- **Koppel elke productieworkspace aan Git** vanaf het begin, niet pas nadat de eerste ongewenste, onomkeerbare wijziging in de portal is gemaakt.
- **Overweeg Mirroring vóór een custom ingestion-pipeline** wanneer de bron een ondersteund systeem is (Azure SQL, Cosmos DB, Snowflake) — het bespaart bouw- en onderhoudswerk dat anders in een eigen CDC-koppeling zou gaan zitten.

## Veelgemaakte Fouten

- **Data onnodig kopiëren tussen Lakehouse en Warehouse** uit gewoonte van andere platformen, terwijl een shortcut of de gedeelde OneLake-laag dat overbodig maakt.
- **Alle workloads even zwaar laten draaien zonder capaciteitsplanning**, waardoor een piek in Power BI-refreshes onverwacht Spark-jobs vertraagt omdat ze dezelfde gedeelde capaciteitspool delen.
- **Complexe, kritieke transformatielogica in Dataflows Gen2 bouwen** puur omdat het laagdrempelig aanvoelt, zonder de testbaarheid en versiebeheerbaarheid van een dbt-model of notebook te overwegen.
- **Direct Lake verwarren met DirectQuery** — Direct Lake leest OneLake's eigen geoptimaliseerde formaat rechtstreeks in geheugen, met prestaties dichter bij Import dan bij traditionele DirectQuery, wat vaak wordt onderschat.
- **Workspaces niet aan Git koppelen**, waardoor wijzigingen in pipelines of notebooks alleen terug te vinden zijn via Fabric's eigen versiegeschiedenis, zonder de reviewbaarheid van een pull request.
- **Een custom ingestion-pipeline bouwen voor een bron die Mirroring al ondersteunt**, waardoor onnodig onderhoudswerk ontstaat voor iets dat native en beheerd beschikbaar was.

## Performance Tips

- **Gebruik shortcuts strategisch, niet overal** — elke shortcut naar externe opslag introduceert een netwerkafhankelijkheid; voor zeer frequent bevraagde data kan een lokale kopie soms toch sneller zijn ondanks het synchronisatienadeel.
- **Houd Direct Lake-tabellen binnen de door Microsoft gedocumenteerde grootte- en kolomlimieten** — buiten die grenzen valt een rapport automatisch terug op DirectQuery-achtig gedrag, met de bijbehorende prestatie-impact.
- **Plan zware Spark-notebook-runs buiten piekuren van Power BI-gebruik** binnen dezelfde capaciteit, om te voorkomen dat gedeelde capaciteit een van beide workloads laat wachten.
- **Gebruik V-Order (Fabric's Delta-tabeloptimalisatie) bewust** voor tabellen die primair via Direct Lake of SQL worden bevraagd — het optimaliseert de fysieke bestandslayout specifiek voor leesprestaties op deze workloads.
- **Beperk het aantal actieve shortcuts naar dezelfde externe bron** waar mogelijk — elke shortcut voegt een eigen verbindings- en metadata-laag toe, en een enkele, gedeelde shortcut is doorgaans beter te onderhouden dan meerdere losse verwijzingen naar dezelfde data vanuit verschillende Lakehouses.

## Interviewvragen

**"Wat is OneLake, en waarom is het architectonisch anders dan hoe Snowflake of Databricks opslag regelen?"** Let op: één gedeelde, tenant-brede opslaglaag onder alle Fabric-workloads, in plaats van een opslagvorm gekoppeld aan één specifieke engine — meerdere gespecialiseerde motoren op dezelfde data, niet één motor met eigen opslag.

**"Wat is een shortcut in Fabric, en welk probleem lost het op?"** Let op: een verwijzing naar externe of andere-workspace-data zonder te kopiëren, vergelijkbaar in geest met Snowflake's Secure Data Sharing maar breder toepasbaar — voorkomt onnodige synchronisatiepijplijnen.

**"Leg uit wat Direct Lake is, en hoe het verschilt van Import en DirectQuery."** Let op: rechtstreeks lezen van OneLake's geoptimaliseerde Delta-formaat in geheugen, zonder importstap (dus geen verouderde kopie) en zonder de latency van traditionele DirectQuery.

**"Hoe verschilt Fabric's kostenmodel van Snowflake's warehouse-credits of Databricks' DBU's?"** Let op: gedeelde capaciteit (F-SKU) over alle workloads binnen een tenant, met smoothing en bursting, in plaats van per-workload afgerekende, geïsoleerde compute.

**"Wanneer zou je Dataflows Gen2 kiezen boven een notebook- of SQL-gebaseerde transformatie?"** Let op: laagdrempelige, visuele transformatie voor teams met sterke Power Query-vaardigheden en minder complexe logica — met de kanttekening dat kritieke, testbare businesslogica zich vaak beter leent voor versiebeheerde code.

**"Waarom zou je een Fabric-workspace aan Git koppelen?"** Let op: items worden dan als broncode gesynchroniseerd met reviewbare pull requests, in plaats van alleen binnen de portal te bestaan met beperkte, minder transparante versiegeschiedenis.

**"Wat is Mirroring, en waarom is het anders dan een reguliere ingestion-pipeline?"** Let op: een beheerde, vrijwel real-time replicatie van specifiek ondersteunde externe databases (Azure SQL, Cosmos DB, Snowflake) rechtstreeks naar OneLake als Delta-tabellen, zonder zelf een CDC-pipeline te hoeven bouwen — met als beperking dat het alleen voor ondersteunde bronnen werkt.

## Relevante Documentatie

- Microsoft Fabric: OneLake Overview — de officiële uitleg van de gedeelde opslaglaag.
- Microsoft Fabric: OneLake Shortcuts — configuratie en beperkingen in detail.
- Microsoft Fabric: Direct Lake Overview — grootte- en kolomlimieten, en fallback-gedrag.
- Microsoft Fabric: Capacity and Billing — F-SKU's, smoothing en bursting in detail.

## Samenvatting

Microsoft Fabric onderscheidt zich niet primair door een sterkere individuele engine, maar door een architectuurkeuze: één gedeelde opslaglaag (OneLake) onder meerdere gespecialiseerde workloads — Lakehouse, Data Warehouse, KQL Database, Power BI — die dezelfde onderliggende Delta-data delen in plaats van kopiëren. Shortcuts verlengen dit principe naar externe opslag, Direct Lake elimineert de traditionele import/refresh-cyclus voor Power BI, en het capacity-based billingmodel met smoothing/bursting vervangt de per-warehouse-scheiding van andere platformen door gedeeld capaciteitsbeheer. Mirroring bouwt hierop voort door zelfs de ingestion-stap voor ondersteunde bronnen native te beheren, zonder een eigen CDC-pipeline te hoeven bouwen. Met alle drie de grote platformen nu behandeld — Snowflake, Databricks, Fabric — verschuift de aandacht in het volgende hoofdstuk naar een meer gespecialiseerd onderdeel van het Microsoft-ecosysteem: Azure Data Factory, dat ook buiten Fabric zelfstandig wordt ingezet.

HOOFDSTUK 13

# Azure Data Factory

## Introductie

Azure Data Factory (ADF) is in eerdere hoofdstukken vooral zijdelings genoemd — als ingestion-tool in Introduction to Data Engineering, als ETL-stijl transformatietool in ETL vs ELT. Fabric's ingebouwde Data Factory-ervaring (zie Microsoft Fabric) deelt hetzelfde objectmodel en dezelfde pipeline-JSON-structuur als standalone ADF, maar richt zich op OneLake in plaats van op willekeurige Azure-resources als doel — wat in dit hoofdstuk volgt, is dus relevant voor beide, met standalone ADF als uitgangspunt omdat dat de meest volledige, meest gebruikte vorm is buiten Fabric-specifieke scenario's.

## Het Objectmodel: Pipelines, Activities, Datasets, Linked Services

ADF's structuur bestaat uit vier concepten die strikt van elkaar te onderscheiden zijn, en waar beginners regelmatig verwarring over hebben.

**Linked services** zijn verbindingsconfiguraties — hoe ADF met een specifiek systeem praat (een Azure SQL-database, een Snowflake-account, een SFTP-server), inclusief authenticatie. Een linked service bevat geen data-specifieke informatie, alleen de verbinding zelf.

**Datasets** verwijzen naar een specifieke data-representatie binnen een linked service — een tabel, een map met CSV-bestanden, een specifieke API-endpoint. Eén linked service (bijvoorbeeld een Azure SQL-database) kan meerdere datasets hebben (verschillende tabellen binnen die database).

**Activities** zijn de daadwerkelijke uitvoerbare stappen — een Copy Activity die data van dataset A naar dataset B verplaatst, een Notebook Activity die een Databricks-notebook aanroept, een Lookup Activity die een enkele waarde ophaalt om verderop in de pipeline te gebruiken.

**Pipelines** groeperen activities in een logische, geordende (of vertakte) reeks, met afhankelijkheden tussen activities — functioneel een DAG, vergelijkbaar met een Airflow-DAG uit Introduction to Data Engineering, maar dan gedeclareerd als JSON in plaats van als Python-code.

```
{
  "name": "orders_ingestion_pipeline",
  "activities": [
    {
      "name": "Copy_Orders_to_Bronze",
      "type": "Copy",
      "inputs": [{ "referenceName": "SourceOrdersDataset", "type": "DatasetReference" }],
      "outputs": [{ "referenceName": "BronzeOrdersDataset", "type": "DatasetReference" }]
    },
    {
      "name": "Trigger_Silver_Transform",
      "type": "ExecuteDataFlow",
      "dependsOn": [{ "activity": "Copy_Orders_to_Bronze", "dependencyConditions": ["Succeeded"] }],
      "dataFlow": { "referenceName": "SilverOrdersDataFlow", "type": "DataFlowReference" }
    }
  ]
}
```

## Copy Activity: het Werkpaard

De **Copy Activity** is verreweg de meest gebruikte activity in ADF, en verdient een preciezer begrip dan "het kopieert data". Bij het configureren kies je een **source** (waar data vandaan komt) en een **sink** (waar het naartoe gaat), elk met eigen instellingen — een SQL-

source ondersteunt bijvoorbeeld een custom query in plaats van een hele tabel, een sink naar een warehouse ondersteunt keuzes tussen `insert`, `upsert`, of eerst de doeltabel leegmaken.

De prestaties van een Copy Activity worden bepaald door **Data Integration Units (DIU's)** — een abstracte maat voor de toegewezen rekenkracht en netwerkbandbreedte, vergelijkbaar in geest met warehouse-sizing uit Snowflake maar dan voor het kopieerproces zelf. Meer DIU's toewijzen (of ADF automatisch laten schalen) versnelt grote kopieertaken, tot een punt waarop de bron of het doel zelf de beperkende factor wordt — meer DIU's helpen dan niet meer, net zoals een groter warehouse niets oplevert als de brondatabase zelf de bottleneck is.

## Integration Runtimes: waar Voert ADF Eigenlijk Uit?

Een concept zonder directe tegenhanger bij de andere platformen in dit handboek: de **Integration Runtime (IR)** bepaalt *waar* de daadwerkelijke dataverplaatsing en -berekening plaatsvindt, en er zijn drie varianten.

**Azure IR** is de standaard, volledig door Microsoft beheerde compute in de cloud — geschikt voor cloud-naar-cloud-scenario's, wat de meerderheid van moderne pipelines dekt.

**Self-hosted IR** draait op infrastructuur die jij zelf beheert — een virtuele machine binnen je eigen netwerk, of zelfs on-premises — specifiek nodig wanneer ADF toegang moet krijgen tot een bronsysteem dat niet publiek vanaf het internet bereikbaar is (een on-premises database achter een firewall, een intern SFTP-systeem). Dit is de meest voorkomende reden dat een organisatie een self-hosted IR nodig heeft: niet performance, maar netwerktoegang.

**Azure-SSIS IR** host SQL Server Integration Services-pakketten binnen ADF, specifiek bedoeld voor organisaties die bestaande SSIS-investeringen (het klassieke on-premises ETL-tool-landschap uit Introduction to Data Engineering) willen migreren naar de cloud zonder de pakketten volledig te herschrijven — een brug-technologie voor legacy-migraties.

## Mapping Data Flows: Visueel, maar Spark Eronder

ETL vs ELT noemde Mapping Data Flows al als voorbeeld van transformatie op een aparte compute-laag. De mechaniek erachter: wat je visueel opbouwt als een reeks transformatiestappen (filter, join, aggregate, derived column) compileert ADF bij uitvoering naar een Spark-job, uitgevoerd op een cluster dat ADF voor je beheert — je schrijft nooit zelf Spark-code, maar de onderliggende uitvoering is wel degelijk Spark, met dezelfde partitionerings- en shuffle-overwegingen die ook in PySpark aan bod komen.

```
Bron (Azure SQL: orders)
  → Filter (amount > 0)
  → Derived Column (order_month = month(order_date))
  → Aggregate (group by order_month, category; sum(amount))
  → Sink (Data Lake: gold/revenue_by_month)
```

Dit compileert tot een Spark-job met dezelfde onderliggende principes als een handgeschreven PySpark-transformatie, maar met een visuele interface eromheen — waardevol voor teams die de flexibiliteit van Spark willen zonder zelf Spark-code te onderhouden, met dezelfde afweging als bij Fabric's Dataflows Gen2 uit Microsoft Fabric: toegankelijk voor eenvoudigere logica, minder testbaar en versiebeheerbaar dan code voor complexere, kritieke transformaties.

## Triggers: Voorbij een Simpel Schema

Een schedule trigger (elke nacht om 3 uur) dekt de meeste gevallen, maar twee andere triggertypes lossen specifieke problemen op die een simpel schema niet kan.

**Tumbling window triggers** verdelen tijd in aaneengesloten, niet-overlappende vensters (elk uur, elke dag) en garanderen dat elk venster precies één keer wordt verwerkt, inclusief automatische afhandeling van **backfills**: als je een pipeline vandaag activeert met een startdatum van drie maanden geleden, voert ADF automatisch alle tussenliggende vensters achtereenvolgens uit, elk met zijn eigen venster-specifieke parameters (`WindowStart`, `WindowEnd`) beschikbaar binnen de pipeline. Dit lost het probleem op dat een gewone

schedule trigger niet heeft: gegarandeerde, herhaalbare verwerking per exact tijdvak, inclusief retries per individueel venster zonder de hele historie opnieuw te hoeven draaien.

**Event-based triggers** starten een pipeline zodra een specifieke gebeurtenis plaatsvindt — meestal het aankomen van een nieuw bestand in Azure Blob Storage — in plaats van te wachten op een vast tijdstip. Dit is relevant wanneer een externe partij bestanden op onvoorspelbare tijdstippen aanlevert: de pipeline reageert direct in plaats van te wachten tot het volgende geplande moment, wat de latency tussen aankomst en verwerking aanzienlijk verkort.

```
{
  "name": "hourly_tumbling_window",
  "type": "TumblingWindowTrigger",
  "typeProperties": {
    "frequency": "Hour",
    "interval": 1,
    "startTime": "2026-08-01T00:00:00Z",
    "delay": "00:05:00",
    "maxConcurrency": 5
  }
}
```

## Parameterisatie: Eén Pipeline, Meerdere Omgevingen

Net als `ref()` in dbt (zie dbt Fundamentals) compile-time de juiste omgeving oplost, laat ADF pipelines en datasets **parametriseren** zodat dezelfde pipeline-definitie tegen dev, test en productie kan draaien zonder te dupliceren:

```
{
  "name": "CopyOrdersParameterized",
  "parameters": {
    "SourceSchema": { "type": "string", "defaultValue": "dev_staging" }
  },
  "activities": [
    {
      "name": "Copy_Orders",
      "type": "Copy",
      "typeProperties": {
        "source": { "type": "AzureSqlSource", "sqlReaderQuery": "SELECT * FROM @{pipeline().parameters.SourceSchema}.orders" }
      }
    }
  ]
}
```

Gecombineerd met **dynamic content** (expressies zoals `@pipeline().parameters.X`, `@utcnow()`, `@concat()`) kunnen zelfs bestandspaden, querytekst en connectiedetails runtime worden samengesteld — de sleutel om te voorkomen dat je voor elke nieuwe bron of omgeving een volledig nieuwe pipeline moet bouwen in plaats van bestaande, geparametriseerde pipelines te hergebruiken.

## Foutafhandeling en Monitoring

Elke activity in ADF ondersteunt native **retry policies** — een aantal pogingen en een interval ertussen, ingesteld per activity in plaats van pipeline-breed — direct relevant voor de tijdelijke-fout-afhandeling uit Introduction to Data Engineering:

```
{
  "name": "Copy_Orders_to_Bronze",
  "type": "Copy",
  "policy": { "retry": 3, "retryIntervalInSeconds": 30, "timeout": "02:00:00" }
}
```

Voorbij retries op activity-niveau biedt ADF **failure paths**: een activity kan een andere activity triggeren specifiek bij falen (via een `Failed`-dependency-conditie), wat expliciete foutafhandelingslogica mogelijk maakt — bijvoorbeeld een Web Activity die een Slack- of Teams-melding stuurt zodra een kritieke Copy Activity faalt, los van de reguliere pipeline-flow. Voor monitoring op langere termijn integreert ADF met **Azure Monitor**: pipeline-runs, activity-duur en foutpercentages zijn hier centraal te bevragen en te alerteren, wat de basis vormt voor het soort observability dat in Monitoring & Observability verder wordt uitgewerkt.

## CI/CD voor ADF: het adf_publish-Patroon

ADF-pipelines worden in de Git-gekoppelde weergave opgeslagen als JSON-bestanden per object (pipelines, datasets, triggers) in een **collaboration branch** (meestal `main`), maar dat is niet wat daadwerkelijk naar Azure wordt gedeployed. Bij het klikken op "Publish" genereert ADF automatisch **Azure Resource Manager (ARM) templates** en commit die naar een aparte, automatisch beheerde `adf_publish`-branch — dít is wat een CI/CD-pipeline (zie CI/CD for Data Engineering) daadwerkelijk deployt naar test- of productieomgevingen, met omgevingsspecifieke parameters (connectiestrings, schema-namen) los meegegeven via een parameters-bestand.

```
main (collaboration branch)          adf_publish (auto-gegenereerd)
├── pipeline/orders_pipeline.json  →  ├── ARMTemplateForFactory.json
├── dataset/bronze_orders.json     →  ├── ARMTemplateParametersForFactory.json
└── trigger/hourly_trigger.json    →  └── (gegenereerd bij elke Publish-actie)
```

Dit tweeledige model — leesbare JSON per object voor development en review, gecompileerde ARM-templates voor deployment — is wezenlijk anders dan hoe dbt of Databricks Repos code behandelen (waar het broncodebestand zelf ook het deploybare artefact is), en is een veelvoorkomende bron van verwarring bij teams die voor het eerst ADF's CI/CD-flow opzetten.

## Best Practices

- **Gebruik tumbling window triggers voor elke pipeline waar exacte, herhaalbare tijdvak-verwerking en backfill-ondersteuning nodig zijn** — een gewone schedule trigger biedt dat niet native.
- **Reserveer self-hosted IR specifiek voor netwerktoegang-scenario's**, niet als standaardkeuze — Azure IR is eenvoudiger te onderhouden zodra publieke of peered cloud-connectiviteit volstaat.
- **Parametriseer pipelines vanaf het begin**, ook als er nog maar één omgeving is — een pipeline die pas achteraf parametriseerbaar gemaakt moet worden, kost aanzienlijk meer herwerk dan er vooraf rekening mee houden.
- **Gebruik Mapping Data Flows voor matig complexe, wijzigende transformaties**; verplaats kritieke, uitgebreid geteste businesslogica naar code (dbt, notebooks) zodra de complexiteit toeneemt.
- **Monitor DIU-verbruik en doorlooptijd samen**, niet DIU's geïsoleerd — meer DIU's toewijzen aan een Copy Activity die door de bron wordt beperkt, is verspild budget.
- **Configureer failure paths voor kritieke pipelines**, niet alleen retries — een fout die na alle retries alsnog faalt, moet iemand actief bereiken, niet stil in een logboek verdwijnen.
- **Begrijp het onderscheid tussen de collaboration branch en `adf_publish`** voordat je een CI/CD-pipeline opzet — deployen vanuit de verkeerde branch is een veelgemaakte, verwarrende eerste fout.

## Veelgemaakte Fouten

- **Linked services, datasets en activities door elkaar husselen** conceptueel — bijvoorbeeld authenticatie-instellingen op datasetniveau proberen te zetten in plaats van op de linked service, wat hergebruik onnodig lastig maakt.
- **Self-hosted IR gebruiken wanneer Azure IR had volstaan**, wat onnodige onderhoudslast (patches, beschikbaarheid van de VM zelf) toevoegt zonder functioneel voordeel.
- **Geen tumbling window trigger gebruiken voor tijdvak-gevoelige verwerking**, waardoor backfills handmatig en foutgevoelig moeten worden uitgevoerd in plaats van automatisch per venster.
- **Pipelines hardcoderen per omgeving** (een aparte pipeline-kopie voor dev en prod) in plaats van te parametriseren, wat wijzigingen dubbel werk maakt en de kans op configuratie-drift tussen omgevingen vergroot.
- **Complexe businesslogica permanent in Mapping Data Flows laten groeien** zonder ooit te heroverwegen of het inmiddels beter als geteste, versiebeheerde code past.

## Performance Tips

- **Gebruik parallelle kopieeractiviteiten** (`parallelCopies`) bij het verplaatsen van veel, onafhankelijke bestanden of partities, in plaats van sequentieel te kopiëren.
- **Beperk het aantal actieve tumbling window-vensters tijdens een backfill** via `maxConcurrency`, zodat een grote historische inhaalslag niet de bron of het doelsysteem overbelast.
- **Gebruik staged copy** (eerst naar Blob Storage, dan naar het uiteindelijke doel) bij trage of instabiele directe verbindingen — dit isoleert netwerkfouten van het uiteindelijke laadproces.
- **Cache Lookup Activity-resultaten** die binnen dezelfde pipeline-run herhaald nodig zijn, in plaats van dezelfde lookup-query meerdere keren uit te voeren.

## Interviewvragen

**"Wat is het verschil tussen een linked service en een dataset in ADF?"** Let op: een linked service is de verbinding naar een systeem (met authenticatie); een dataset verwijst naar een specifieke data-representatie binnen die verbinding — één linked service kan meerdere datasets bedienen.

**"Wanneer zou je een self-hosted integration runtime nodig hebben in plaats van Azure IR?"** Let op: primair voor netwerktoegang tot systemen die niet publiek bereikbaar zijn (on-premises, achter een firewall), niet voor extra performance.

**"Wat is een tumbling window trigger, en welk probleem lost het op dat een schedule trigger niet oplost?"** Let op: gegarandeerde, exact-één-keer-verwerking per aaneengesloten tijdvak met automatische, parametriseerbare backfill-ondersteuning — een schedule trigger heeft dat mechanisme niet ingebouwd.

**"Hoe werkt Mapping Data Flows onder de motorkap?"** Let op: de visuele transformatiestappen compileren naar een Spark-job op door ADF beheerde compute — geen eigen executie-engine, maar een visuele laag boven Spark.

**"Hoe zou je dezelfde pipeline tegen zowel een dev- als een productieomgeving laten draaien zonder te dupliceren?"** Let op: parameterisatie van pipelines en datasets, gecombineerd met dynamic content-expressies, zodat omgevingsspecifieke waarden runtime worden ingevuld in plaats van hardcoded.

**"Wat bepalen Data Integration Units (DIU's), en wanneer helpt meer DIU's toewijzen niet meer?"** Let op: DIU's bepalen de toegewezen rekenkracht/bandbreedte voor een Copy Activity; zodra de bron of het doelsysteem zelf de beperkende factor is, levert meer DIU's geen extra snelheid op.

**"Leg het adf_publish-patroon uit — waarom wordt niet direct vanuit de main-branch gedeployed?"** Let op: de main-branch bevat leesbare JSON per object voor development en review; Publish genereert daaruit ARM-templates in een aparte, automatisch beheerde branch, en dát is wat CI/CD daadwerkelijk naar omgevingen deployt.

**"Hoe zou je ervoor zorgen dat een team direct op de hoogte is als een kritieke pipeline faalt?"** Let op: een failure path met een Web Activity die een melding stuurt (Slack/Teams), gecombineerd met retry policies op activity-niveau voor tijdelijke fouten, en Azure Monitor-integratie voor structurele alerting.

## Relevante Documentatie

- Azure Data Factory: Concepts and Terminology — pipelines, activities, datasets en linked services in detail.
- Azure Data Factory: Integration Runtime — de drie IR-varianten en wanneer je welke kiest.
- Azure Data Factory: Tumbling Window Trigger — configuratie, backfills en concurrency.
- Azure Data Factory: Mapping Data Flows — transformatietypes en de onderliggende Spark-executie.

## Samenvatting

ADF's objectmodel — linked services voor verbindingen, datasets voor data-representaties, activities voor uitvoerbare stappen, pipelines voor de DAG die ze samenbrengt — vormt de basis voor alles wat erop volgt. Copy Activity en DIU's regelen ruwe dataverplaatsing; integration runtimes bepalen *waar* dat gebeurt, met self-hosted IR specifiek voor netwerktoegang tot afgeschermde systemen. Mapping Data Flows biedt een visuele laag boven Spark voor teams die de kracht van Spark willen zonder zelf Spark-code te schrijven. Tumbling window triggers lossen het backfill- en exact-eenmaal-verwerkingsprobleem op dat een simpel schema niet aankan, en parameterisatie maakt pipelines herbruikbaar over omgevingen — hetzelfde principe als `ref()` in dbt, hier toegepast op orchestratie in plaats van transformatie. Retry policies en failure paths regelen foutafhandeling op activity-niveau; het adf_publish-patroon bepaalt hoe pipelines daadwerkelijk hun weg naar productie vinden, wezenlijk anders dan de directe code-als-artefact-benadering van dbt of Databricks Repos. Met de drie platformen en hun bijbehorende ingestion-tooling nu behandeld, verlegt het volgende hoofdstuk de aandacht naar de programmeertaal die door alle drie heen loopt: Python.

HOOFDSTUK 14

# Python voor Data Engineers

## Introductie

Python is in vrijwel elk voorgaand hoofdstuk al zijdelings gebruikt — extractiescripts in Introduction to Data Engineering, Snowpark in Snowflake, MLflow in Databricks. Dit hoofdstuk behandelt Python zelf, specifiek zoals het in data engineering wordt gebruikt: niet als taal om zware dataverwerking mee te doen (dat is het domein van SQL binnen het warehouse, of PySpark voor gedistribueerde verwerking — het onderwerp van het volgende hoofdstuk), maar als **verbindende taal**: API's aanroepen, bestanden verwerken, orchestratielogica schrijven, en de lijm tussen systemen die geen gedeelde taal spreken.

Dit is geen algemene Python-cursus — als je al basiskennis hebt van functies, loops en dictionaries, behandelt dit hoofdstuk specifiek de patronen die in productiepipelines steeds terugkomen en die generieke Python-tutorials zelden behandelen.

## Waarom Python de Standaard Werd

Python's dominantie in data engineering is geen toeval: de leesbare syntax verlaagt de drempel voor teams die niet primair uit software engineers bestaan, het ecosysteem aan libraries (requests, pandas, SQLAlchemy, elke cloud-SDK) dekt vrijwel elke integratiebehoefte zonder zelf iets te hoeven bouwen, en het is de gemeenschappelijke taal tussen data engineering, data science en ML engineering — relevant voor de vervagende grens tussen die rollen die in Introduction to Data Engineering werd besproken. Geen enkele van deze redenen gaat over ruwe uitvoeringssnelheid — Python is trager dan Java of Scala — maar in data engineering is ontwikkelsnelheid en integratiegemak vrijwel altijd belangrijker dan de laatste milliseconden performance, omdat de zware rekenkracht toch al gedelegeerd wordt aan het warehouse of Spark-cluster.

## Virtual Environments en Dependency Management

Een pipeline die op je eigen laptop werkt maar in productie faalt omdat een library-versie verschilt, is een van de meest voorkomende en makkelijkst te voorkomen operationele problemen. Een **virtual environment** isoleert de Python-packages van één project van die van andere projecten en van het systeem-Python:

```
python -m venv .venv
source .venv/bin/activate       # Windows: .venv\Scripts\activate
pip install -r requirements.txt
```

`requirements.txt` legt exacte versies vast, zodat "het werkt bij mij" wordt vervangen door "het werkt met precies deze set versies, overal":

```
requests==2.32.3
pandas==2.2.2
pyarrow==17.0.0
```

Modernere projecten gebruiken steeds vaker `pyproject.toml` met tools als Poetry of uv, die naast dependency-versies ook een reproduceerbare lock-file bijhouden — functioneel hetzelfde doel (reproduceerbaarheid), met betere afhankelijkheidsresolutie dan een handmatig onderhouden `requirements.txt`. Voor CI/CD (zie CI/CD for Data Engineering) is dit niet optioneel: zonder vastgelegde versies kan een pipeline die vorige week werkte, vandaag falen puur omdat een dependency een nieuwe, incompatibele versie uitbracht.

## Robuuste API-extractie: Paginering en Retries

Een realistisch extractiescript tegen een externe API moet met twee dingen omgaan die tutorials vaak overslaan: **paginering** (de meeste API's geven data in brokken van bijvoorbeeld 100 records terug, niet alles in één keer) en **tijdelijke fouten** (netwerkhaperingen, rate limits, korte uitval) — exact de "retries zijn de norm, niet de uitzondering"-les uit Introduction to Data Engineering, hier concreet in code.

```
import time
import requests
from requests.exceptions import RequestException

def fetch_all_orders(api_url: str, api_key: str, max_retries: int = 3) -> list[dict]:
    """Haalt alle orders op via paginering, met exponential backoff bij fouten."""
    all_orders = []
    page = 1

    while True:
        for attempt in range(max_retries):
            try:
                response = requests.get(
                    api_url,
                    headers={"Authorization": f"Bearer {api_key}"},
                    params={"page": page, "page_size": 100},
                    timeout=30,
                )
                response.raise_for_status()
                break
            except RequestException as e:
                if attempt == max_retries - 1:
                    raise
                wait = 2 ** attempt   # exponential backoff: 1s, 2s, 4s
                time.sleep(wait)

        data = response.json()
        if not data["results"]:
            break

        all_orders.extend(data["results"])
        page += 1

    return all_orders
```

Merk op: de `for attempt in range(max_retries)`-lus vangt alleen *tijdelijke* fouten op en herprobeert; als alle pogingen falen, wordt de fout alsnog doorgegeven (`raise`) in plaats van stilzwijgend te falen — een pipeline die een aanhoudende fout negeert, is gevaarlijker dan een pipeline die crasht en een alert triggert.

## Bestandsformaten in de Praktijk

Data engineers werken voortdurend met bestandsformaten, en de keuze heeft directe gevolgen voor de rest van de pipeline. **CSV** is universeel leesbaar maar heeft geen ingebouwd schema — elke kolom is tekst totdat iemand expliciet cast, precies het "cast nooit te vroeg"-probleem uit Data Warehousing Fundamentals. **JSON** behoudt structuur en geneste data, maar is rij-georiënteerd en dus inefficiënt voor analytische verwerking op schaal. **Parquet** is het formaat waar de meeste bronze-lagen in moderne pipelines op landen: kolomgeoriënteerd (dezelfde reden als in Data Warehousing Fundamentals besproken), zelfbeschrijvend qua schema, en sterk gecomprimeerd.

```
import pandas as pd

# CSV inlezen, EXPLICIET met dtype-controle -- niet blind vertrouwen op auto-detectie
df = pd.read_csv("orders.csv", dtype={"order_id": str, "customer_id": str})

# Wegschrijven als Parquet: kolomgeoriënteerd, gecomprimeerd, schema-bewust
df.to_parquet("orders.parquet", engine="pyarrow", compression="snappy")
```

De vuistregel: gebruik CSV/JSON voor uitwisseling met externe systemen die het vereisen, maar converteer zo vroeg mogelijk in de pipeline naar Parquet zodra data intern verder stroomt — elke stap die daarna nog CSV gebruikt, betaalt onnodig de kosten van een rij-georiënteerd, ongecomprimeerd formaat.

## Type Hints: Belangrijker in Pipelines dan Elders

Python is dynamisch getypeerd, maar **type hints** (sinds Python 3.5) laten je functiesignaturen expliciet documenteren zonder de dynamische aard van de taal te verliezen:

```
def calculate_revenue(orders: list[dict[str, float]], tax_rate: float = 0.21) -> float:
    return sum(order["amount"] for order in orders) * (1 + tax_rate)
```

In een data pipeline is dit specifiek waardevoller dan in bijvoorbeeld een webapplicatie: een pipeline verwerkt vaak data van buiten je controle (een API die stilletjes een veldtype wijzigt, een CSV met een onverwachte lege waarde), en een tool als **mypy** die type hints statisch controleert, vangt een categorie fouten (een functie die een string krijgt waar een `float` werd verwacht) vóórdat de pipeline draait, in plaats van pas bij een crash — of erger, bij stilzwijgend foute berekeningen — middenin een nachtelijke run.

## Foutafhandelingspatronen voor Pipelines

Een pipeline-specifiek patroon dat generieke Python-tutorials zelden behandelen: onderscheid maken tussen fouten die het waard zijn om te herproberen en fouten die dat niet zijn, via **custom exceptions**:

```
class RetryableError(Exception):
    """Tijdelijke fout: netwerktimeout, rate limit -- opnieuw proberen heeft zin."""
    pass

class FatalError(Exception):
    """Structurele fout: ongeldige credentials, corrupt schema -- opnieuw proberen helpt niet."""
    pass

def process_batch(batch: list[dict]) -> None:
    try:
        validate_schema(batch)      # kan FatalError opwerpen
        load_to_warehouse(batch)    # kan RetryableError opwerpen
    except FatalError:
        raise                        # nooit automatisch herproberen
    except RetryableError:
        retry_with_backoff(process_batch, batch)
```

Dit voorkomt de veelgemaakte fout van een generieke `except Exception` die elke fout hetzelfde behandelt — een verkeerd wachtwoord verdient een onmiddellijke, duidelijke melding, geen drie herhaalde pogingen die toch nooit gaan slagen. Combineer dit met **context managers** (`with`-statements) voor resources die altijd netjes afgesloten moeten worden, ook bij een fout halverwege:

```
from contextlib import contextmanager

@contextmanager
def warehouse_connection(conn_string: str):
    conn = connect(conn_string)
    try:
        yield conn
    finally:
        conn.close()   # gegarandeerd uitgevoerd, ook als er een exception optreedt

with warehouse_connection(CONN_STRING) as conn:
    load_data(conn, batch)
```

## Logging in Plaats van print()

Een `print()`-statement verdwijnt zodra een pipeline in productie draait zonder interactieve terminal; het Python `logging`-module legt berichten gestructureerd vast, met niveaus (`INFO`, `WARNING`, `ERROR`) die filtering en routing naar monitoringtools mogelijk maken — de basis waarop de observability-principes uit Monitoring & Observability daadwerkelijk gebouwd worden:

```
import logging

logging.basicConfig(
    level=logging.INFO,
```

```
    format="%(asctime)s [%(levelname)s] %(name)s: %(message)s"
)
logger = logging.getLogger("orders_pipeline")

logger.info(f"Extractie gestart: {len(orders)} orders opgehaald")
logger.warning(f"{skipped_count} rijen overgeslagen wegens ontbrekende customer_id")
logger.error(f"Laden mislukt na {max_retries} pogingen: {e}")
```

Het verschil met `print()` is niet alleen esthetisch: `logging` laat je output routeren naar een bestand, een centrale logaggregator, of een monitoringtool, met niveau-gebaseerde filtering (`WARNING` en hoger naar een alertingkanaal, `INFO` alleen naar een archief) — precies het mechanisme dat het mogelijk maakt om te weten dat een pipeline een probleem had, zonder dat iemand continu meekijkt.

## Data Valideren met Dataclasses en Pydantic

Type hints alleen worden niet afgedwongen tijdens runtime — een functie die een `float` verwacht, crasht niet automatisch als je er een string aan doorgeeft, tenzij je dat expliciet controleert. Voor data die van buiten je controle komt (een API-response, een CSV-rij) is runtime-validatie wél nodig, en **Pydantic** is hiervoor de gangbare library geworden: je definieert een model met getypeerde velden, en Pydantic valideert en converteert binnenkomende data automatisch, met een duidelijke foutmelding zodra iets niet klopt:

```
from pydantic import BaseModel, field_validator

class Order(BaseModel):
    order_id: str
    customer_id: str
    amount: float
    status: str

    @field_validator("amount")
    @classmethod
    def amount_must_be_positive(cls, v: float) -> float:
        if v <= 0:
            raise ValueError("amount moet positief zijn")
        return v

# Valideert en cast automatisch; werpt een duidelijke ValidationError bij een probleem
raw_record = {"order_id": "123", "customer_id": "456", "amount": "99.50", "status": "paid"}
order = Order.model_validate(raw_record)   # amount wordt automatisch naar float gecast
```

Dit is de Python-equivalent van de data contracts uit Modern Data Platform Architecture en de generic/singular tests uit dbt Fundamentals, maar dan toegepast op het moment dat data een Python-pipeline binnenkomt, vóórdat het ooit een warehouse bereikt — een vroege, expliciete validatielaag in plaats van erop vertrouwen dat foute data zich pas stroomafwaarts in een dbt-test manifesteert. Standaardbibliotheek-`dataclasses` bieden een lichtere, valideringsvrije variant voor situaties waar je alleen structuur wilt vastleggen zonder de overhead van runtime-validatie.

## Best Practices

- **Gebruik altijd een virtual environment met vastgelegde versies**, ook voor kleine scripts — de kosten zijn laag, de voorkomen problemen zijn kostbaar.
- **Converteer zo vroeg mogelijk naar Parquet** zodra data intern verder stroomt, en bewaar CSV/JSON alleen waar externe systemen dat vereisen.
- **Voeg type hints toe aan elke functie die pipeline-data verwerkt**, en draai mypy in CI zodat typefouten vóór productie worden gevangen.
- **Onderscheid retryable en fatale fouten expliciet** met custom exceptions, in plaats van elke fout hetzelfde te behandelen.
- **Gebruik `logging`, nooit `print()`, in elk stuk code dat ooit ongezien in productie zal draaien.**
- **Valideer externe data expliciet met Pydantic (of vergelijkbaar) zodra die een pipeline binnenkomt**, in plaats van te vertrouwen dat foute data pas later, dieper in het platform, wordt opgevangen.

## Veelgemaakte Fouten

- **Geen `requirements.txt` of vergelijkbaar vastleggen**, waardoor een pipeline die vandaag werkt volgende week kan breken door een stille dependency-update.
- **CSV blijven gebruiken tot diep in de pipeline** in plaats van vroeg naar Parquet te converteren, wat onnodige I/O- en opslagkosten oplevert.
- **Een brede `except Exception: pass`** die elke fout — tijdelijk of fataal — stilzwijgend negeert, wat een pipeline "laat slagen" terwijl hij feitelijk data verliest.
- **API-paginering vergeten**, waardoor een extractiescript alleen de eerste pagina ophaalt en niemand merkt dat 95% van de data ontbreekt totdat een analist vraagt waarom de cijfers zo laag zijn.
- **`print()`-statements in productiecode laten staan** in plaats van naar `logging` te migreren, waardoor er geen bruikbare, doorzoekbare historie van wat een pipeline deed overblijft.

## Performance Tips

- **Gebruik `pandas` of `pyarrow` voor bestandsverwerking, niet handmatige regel-voor-regel string-parsing** — de geoptimaliseerde, in C geïmplementeerde routines zijn een veelvoud sneller dan pure Python-loops.
- **Batch API-aanroepen waar de bron dat toelaat** (grotere `page_size`) in plaats van veel kleine, sequentiële requests — minder round-trips betekent minder overhead.
- **Vermijd het volledig inladen van zeer grote bestanden in geheugen** — gebruik chunked reading (`pd.read_csv(..., chunksize=...)`) of stap over naar Spark zodra volumes de grens van comfortabel in-memory verwerken overschrijden, het onderwerp van het volgende hoofdstuk.
- **Cache dure, herhaalde berekeningen of lookups** binnen een run expliciet, in plaats van dezelfde bewerking meerdere keren uit te voeren.

## Interviewvragen

**"Waarom is Python de dominante taal in data engineering, ondanks dat het trager is dan Java of Scala?"** Let op: leesbaarheid, ecosysteem, en het feit dat zware rekenkracht toch wordt gedelegeerd aan het warehouse of Spark — ontwikkelsnelheid weegt zwaarder dan ruwe executiesnelheid.

**"Hoe zou je een extractiescript schrijven dat robuust omgaat met een gepagineerde API en tijdelijke netwerkfouten?"** Let op: een paginering-lus die doorgaat tot een lege pagina, gecombineerd met retries met exponential backoff die alleen tijdelijke fouten opvangen, niet fatale.

**"Waarom zou je Parquet verkiezen boven CSV in een data pipeline?"** Let op: kolomgeoriënteerd (efficiënter voor analytische verwerking), schema-bewust, en sterk gecomprimeerd — dezelfde voordelen als columnar storage in een warehouse, hier toegepast op bestandsniveau.

**"Wat is het verschil tussen een retryable en een fatale fout, en waarom moet je pipeline-code dat onderscheid maken?"** Let op: tijdelijke fouten (netwerktimeout) verdienen automatische herpoging; structurele fouten (verkeerde credentials, corrupt schema) verdienen dat niet — alles hetzelfde behandelen leidt tot zinloze retries of gemiste directe alerts.

**"Waarom is `logging` beter dan `print()` in productiepipelines?"** Let op: `logging` biedt niveaus, routing naar bestanden/monitoringtools, en filtering — `print()`-output verdwijnt zodra een pipeline zonder interactieve terminal draait.

**"Type hints worden niet afgedwongen tijdens runtime — hoe valideer je dan toch data die van buiten komt?"** Let op: het besef dat statische type hints en runtime-validatie twee verschillende dingen zijn, en dat een library als Pydantic expliciete, runtime-afgedwongen validatie met duidelijke foutmeldingen toevoegt voor data van onbetrouwbare bronnen.

**"Wat is het verschil tussen een dataclass en een Pydantic-model?"** Let op: dataclasses leggen alleen structuur vast zonder validatie; Pydantic-modellen valideren en casten actief tijdens runtime, met een duidelijke `ValidationError` bij een probleem.

## Relevante Documentatie

- Python: venv — Virtual Environments — officiële documentatie van Python's ingebouwde virtual-environment-tool.
- Requests: HTTP for Humans — de meest gebruikte Python-library voor API-aanroepen.
- Apache Arrow: Reading and Writing Parquet — pyarrow's Parquet-ondersteuning in detail.
- Python: logging — Logging Facility — officiële documentatie van het logging-framework.
- mypy Documentation — statische type-checking voor Python.
- Pydantic Documentation — runtime-validatie en -parsing van data-modellen.

## Samenvatting

Python's rol in data engineering is die van verbindende taal: API's aanroepen, bestanden verwerken, orchestratielogica schrijven — niet de zware dataverwerking zelf, die aan het warehouse of aan Spark wordt gedelegeerd. Virtual environments met vastgelegde dependency-versies voorkomen het klassieke "werkt bij mij niet in productie"-probleem; robuuste API-extractie vereist expliciete paginering en retries met onderscheid tussen tijdelijke en fatale fouten; Parquet is het standaardformaat zodra data intern stroomt. Type hints, Pydantic-validatie en `logging` zijn geen esthetische keuzes maar directe hulpmiddelen om fouten vroeg te vangen — bij voorkeur al bij binnenkomst van externe data, niet pas wanneer een dbt-test diep in silver iets onverwachts opmerkt — en pipelines observeerbaar te maken. Met deze Python-fundamenten op zak is de volgende stap PySpark: dezelfde taal, maar dan ingezet voor gedistribueerde verwerking op een schaal die één machine niet meer aankan.

HOOFDSTUK 15

# PySpark

## Introductie

Python for Data Engineers sloot af met de grens van wat één machine comfortabel aankan. PySpark is Python's interface naar Apache Spark, het gedistribueerde verwerkingsraamwerk dat werk over meerdere machines verdeelt zodra data te groot wordt voor één machine — of zelfs wanneer dat niet het geval is, simpelweg omdat parallellisatie sneller is. Eerdere hoofdstukken toonden al PySpark-syntax (Medallion Architecture, Databricks); dit hoofdstuk legt uit *waarom* die code zich gedraagt zoals hij doet — de mechaniek onder de syntax, zonder welke PySpark-performance-problemen onbegrijpelijk blijven.

## Lazy Evaluation: Transformations versus Actions

Het eerste en meest verwarrende concept voor nieuwkomers: PySpark-code voert niets uit op het moment dat je hem schrijft. Operaties zijn onderverdeeld in twee categorieën.

**Transformations** (`filter`, `select`, `withColumn`, `groupBy`, `join`) beschrijven *wat* er moet gebeuren, maar voeren niets uit — ze bouwen alleen een uitvoeringsplan op, een DAG van bewerkingen, precies zoals dbt's `ref()`-graph uit dbt Fundamentals declareert wat moet gebeuren zonder het meteen uit te voeren.

**Actions** (`count`, `collect`, `show`, `write`) triggeren de daadwerkelijke uitvoering — pas op dat moment analyseert Spark het volledige opgebouwde plan en voert het uit.

```
df_filtered = df.filter(col("amount") > 0)      # transformation: gebeurt nog niets
df_grouped = df_filtered.groupBy("category").sum("amount")  # nog steeds niets
df_grouped.show()                                 # ACTION: nu pas voert Spark alles uit
```

Dit verklaart een veelvoorkomende verwarring: code die een `filter` bevat met een fout erin, geeft pas een foutmelding bij de `show()` op de laatste regel, niet bij de `filter()` zelf — Spark heeft tot dat moment alleen een plan gebouwd, nooit data aangeraakt. Het verklaart ook waarom `df.count()` twee keer aanroepen in dezelfde sessie de hele berekening twee keer opnieuw uitvoert, tenzij je expliciet cachet (verderop in dit hoofdstuk) — lazy evaluation onthoudt geen tussenresultaten uit zichzelf.

## Partities, Executors en Tasks: Hoe Werk Wordt Verdeeld

Een Spark DataFrame is logisch verdeeld in **partities** — stukken van de data die onafhankelijk van elkaar verwerkt kunnen worden. Een cluster bestaat uit meerdere **executors** (worker-processen, elk met eigen CPU-cores en geheugen, verdeeld over de machines uit Databricks of een vergelijkbaar platform), en Spark's scheduler wijst elke partitie toe als een **task** aan een beschikbare executor-core. Meer partities dan cores betekent dat sommige tasks moeten wachten; te weinig partities (bijvoorbeeld één partitie voor een enorme dataset) betekent dat maar één core werkt terwijl de rest van het cluster stilstaat — een directe, praktische reden om partitieaantal bewust te beheren, uitgewerkt verderop.

```
print(df.rdd.getNumPartitions())   # hoeveel stukken is deze DataFrame nu verdeeld?
```

Dit is de PySpark-specifieke uitwerking van het MPP-principe (Massively Parallel Processing) uit Data Warehousing Fundamentals — dezelfde onderliggende gedachte (werk verdelen over meerdere rekenknopen), hier zichtbaar en direct beïnvloedbaar op DataFrame-niveau in plaats van verborgen achter een warehouse se automatische micro-partitionering.

## Shuffles: de Duurste Operatie in Spark

Dit is het belangrijkste performance-concept in PySpark, en het verklaart waarom sommige operaties dramatisch trager zijn dan andere die er syntactisch vergelijkbaar uitzien. Een **shuffle** is het herverdelen van data over partities heen — nodig zodra een bewerking rijen met dezelfde sleutel bij elkaar moet brengen, ook al stonden ze oorspronkelijk op verschillende executors. Dit vereist het serialiseren,

over het netwerk versturen, en op schijf schrijven van tussenresultaten — kostbaar op een manier die puur in-partition-bewerkingen niet zijn.

`groupBy().agg()`, `join()` (behalve broadcast joins, zie verderop), en `distinct()` triggeren allemaal een shuffle, omdat ze rijen met dezelfde key op dezelfde executor moeten samenbrengen om correct te kunnen aggregeren of matchen. `filter()`, `select()`, `withColumn()` shuffelen nooit — elke rij kan onafhankelijk, binnen zijn eigen partitie, worden verwerkt.

## Wide versus Narrow Transformations

Dit onderscheid formaliseert precies het shuffle-verschil hierboven. **Narrow transformations** (filter, select, withColumn, map) hebben geen shuffle nodig — elke output-partitie hangt af van precies één input-partitie, wat volledig parallelle, onafhankelijke verwerking mogelijk maakt zonder enige communicatie tussen executors. **Wide transformations** (groupBy, join, distinct, orderBy) vereisen wél een shuffle — een output-partitie kan data nodig hebben uit meerdere, mogelijk alle input-partities.

```
# Narrow: elke rij onafhankelijk verwerkbaar, geen shuffle
df_clean = df.filter(col("amount") > 0).withColumn("amount_eur", col("amount") / 100)

# Wide: vereist een shuffle om alle rijen per categorie bij elkaar te brengen
df_summary = df_clean.groupBy("category").agg(F.sum("amount_eur").alias("total"))
```

De praktische consequentie: ketens van narrow transformations zijn vrijwel gratis om toe te voegen — Spark voert ze uit binnen dezelfde partitie zonder extra netwerk- of schijf-I/O. Elke wide transformation in een pipeline is een bewust punt waar je de kosten ervan zou moeten overwegen, vooral als er meerdere na elkaar voorkomen.

## De Catalyst Optimizer: Waarom "Naïeve" Code Vaak Prima Is

Voordat een DataFrame-plan daadwerkelijk wordt uitgevoerd, gaat het door Spark's **Catalyst optimizer**, die het logische plan analyseert en herschrijft naar een efficiëntere fysieke uitvoeringsvorm — vergelijkbaar in geest met hoe een warehouse-queryplanner een SQL-query herschrijft vóór uitvoering. Concreet betekent dit onder andere **predicate pushdown** (een `filter()` na een `join()` in je code geschreven, wordt vaak vóór de join uitgevoerd als dat correct en sneller is) en **kolomsnoei** (alleen kolommen die daadwerkelijk gebruikt worden, worden ingelezen — relevant bij kolomgeoriënteerde formaten als Parquet uit Python for Data Engineers).

```
df.filter(col("category") == "electronics").explain()
```

`.explain()` toont het fysieke uitvoeringsplan zoals Catalyst het daadwerkelijk gaat uitvoeren — onmisbaar bij het debuggen van onverwacht trage queries, omdat het laat zien of een verwachte optimalisatie (pushdown, partition pruning) daadwerkelijk plaatsvond. Dit betekent niet dat queryvolgorde er nooit toe doet — Catalyst is goed, niet perfect — maar het verklaart waarom in de praktijk "logisch geschreven" code vaak al redelijk geoptimaliseerd wordt zonder dat je zelf elke stap handmatig hoeft te herordenen.

## Partitionering in de Praktijk: Repartition en Coalesce

Het aantal partities beïnvloedt direct hoe goed werk parallelliseert, en Spark biedt twee manieren om dat aantal aan te passen, met een belangrijk verschil in kosten.

**`repartition(n)`** shuffelt data volledig opnieuw over `n` partities — kostbaar (het is zelf een wide transformation), maar noodzakelijk als je het aantal partities wilt *vergroten*, of data gelijkmatiger wilt herverdelen na een operatie die scheve partities achterliet.

**`coalesce(n)`** combineert bestaande partities zonder een volledige shuffle, en werkt alleen om het aantal partities te *verkleinen* — aanmerkelijk goedkoper dan `repartition` voor dat specifieke doel, bijvoorbeeld vlak vóór het wegschrijven van een resultaat om te veel kleine outputbestanden te vermijden.

```
df_repartitioned = df.repartition(200, "customer_id")  # herverdeel expliciet op sleutel
df_output = df_summary.coalesce(10)                     # verminder outputbestanden, goedkoop
df_output.write.format("delta").save("gold/summary")
```

**Data skew** — wanneer één partitiesleutel onevenredig veel rijen bevat (bijvoorbeeld één extreem grote klant tussen duizenden kleine) — is een van de meest voorkomende oorzaken van een Spark-job die "bijna klaar" blijft hangen: negenennegentig procent van de tasks

is snel klaar, terwijl één executor blijft worstelen met een oneven grote partitie. Herpartitioneren op een andere, gelijkmatiger verdeelde sleutel, of het toevoegen van een "salt"-waarde aan de skewed key, zijn de gangbare oplossingen.

## Broadcast Joins: een Shuffle Vermijden

Wanneer je een grote tabel joint met een kleine tabel (een fact table met een kleine dimension table, zie Star Schema & Dimensional Modeling), is een volledige shuffle-join onnodig kostbaar. Een **broadcast join** kopieert de kleine tabel volledig naar elke executor, zodat de join lokaal, zonder shuffle, kan plaatsvinden:

```
from pyspark.sql.functions import broadcast

df_result = large_orders_df.join(
    broadcast(small_dim_customers_df), on="customer_id"
)
```

Spark past dit vaak automatisch toe voor tabellen onder een configureerbare groottedrempel (`spark.sql.autoBroadcastJoinThreshold`), maar expliciet `broadcast()` gebruiken is verstandig zodra je zeker weet dat een tabel klein genoeg is en Spark's automatische schatting (gebaseerd op statistieken die niet altijd actueel zijn) dat zelf niet met zekerheid detecteert.

## Cachen: Herhaalde Berekening Vermijden

Omdat lazy evaluation geen tussenresultaten onthoudt, wordt een DataFrame die meerdere keren wordt bevraagd (bijvoorbeeld eenmaal voor een `count()` en eenmaal voor een `write()`) standaard **twee keer volledig herberekend**. `.cache()` (of `.persist()` voor meer controle over opslaglocatie: geheugen, schijf, of beide) bewaart het resultaat na de eerste berekening expliciet:

```
df_expensive = df.join(other_df, "key").groupBy("category").agg(F.sum("amount"))
df_expensive.cache()

print(df_expensive.count())        # berekent en cachet nu
df_expensive.write.format("delta").save("gold/summary")   # hergebruikt de cache, geen herberekening
```

De vuistregel: cache een DataFrame zodra je hem meer dan één keer nodig hebt binnen dezelfde run, en ontcache (`.unpersist()`) expliciet zodra je klaar bent — cache die onnodig lang blijft hangen, verbruikt executor-geheugen dat andere taken nodig hebben.

## Adaptive Query Execution: Optimaliseren Tijdens de Run

Catalyst optimaliseert vóór uitvoering, gebaseerd op statistieken die niet altijd accuraat zijn — vooral na complexe transformatieketens weet Spark vooraf vaak niet precies hoeveel rijen een tussenresultaat zal bevatten. **Adaptive Query Execution (AQE)**, standaard ingeschakeld in moderne Spark-versies, past het uitvoeringsplan tijdens de run zelf aan, op basis van daadwerkelijk waargenomen tussenresultaten in plaats van vooraf geschatte statistieken.

Drie concrete dingen die AQE automatisch doet: **het samenvoegen van te kleine shuffle-partities** (die anders veel overhead per taak zouden veroorzaken relatief tot de hoeveelheid data), **het splitsen of anders afhandelen van skewed partities** (het probleem uit de vorige sectie, deels automatisch verzacht), en **het dynamisch omzetten van een reguliere join naar een broadcast join** zodra tijdens de run blijkt dat een van de twee tabellen kleiner is dan vooraf geschat. Dit betekent niet dat handmatige partitionering en expliciete broadcast joins overbodig zijn geworden — AQE werkt met wat het tijdens de run waarneemt, en een bewust ontworpen pipeline blijft voorspelbaarder dan volledig vertrouwen op runtime-aanpassingen — maar het verklaart waarom veel Spark-jobs tegenwoordig beter presteren dan de handmatige tuning-adviezen uit oudere documentatie zouden doen vermoeden.

```
spark.conf.set("spark.sql.adaptive.enabled", "true")              # standaard al aan
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")      # skew-afhandeling specifiek
```

## Best Practices

- **Ketens narrow transformations vrijelijk** — ze zijn vrijwel gratis; wees terughoudender en bewuster met elke wide transformation (groupBy, join, distinct).

- **Gebruik `.explain()` bij elke onverwacht trage query** vóórdat je gokt naar een oplossing — het toont exact welke optimalisaties wel en niet zijn toegepast.
- **Cache expliciet zodra een DataFrame meer dan eenmaal wordt gebruikt**, en ontcache zodra je klaar bent.
- **Gebruik broadcast joins bewust voor fact/dimension-achtige joins** waar de kleine tabel ruim onder de automatische drempel valt.
- **Gebruik `coalesce` in plaats van `repartition` wanneer je alleen het aantal partities wilt verkleinen** — de kostenbesparing is direct en aanzienlijk.
- **Laat AQE ingeschakeld staan** (het is standaard) en behandel het als een vangnet bovenop, niet als vervanging van, bewuste partitionering en expliciete broadcast joins.

## Veelgemaakte Fouten

- **Meerdere acties op dezelfde onbecachte DataFrame aanroepen**, waardoor dezelfde dure berekening onnodig meerdere keren wordt uitgevoerd.
- **`repartition` gebruiken waar `coalesce` had volstaan**, wat een onnodige volledige shuffle veroorzaakt voor iets dat goedkoper kon.
- **Data skew negeren** totdat een job structureel traag blijft hangen, in plaats van vroeg te herkennen dat één partitiesleutel de rest domineert.
- **Een expliciete broadcast join forceren op een tabel die eigenlijk te groot is**, wat elke executor met een onnodig zware kopie belast en geheugenproblemen kan veroorzaken.
- **`.cache()` overal toepassen "voor de zekerheid"**, wat executor-geheugen verspilt aan tussenresultaten die maar één keer worden gebruikt en dus geen enkel voordeel van cachen hebben.

## Performance Tips

- **Filter en selecteer kolommen zo vroeg mogelijk** in de keten — narrow transformations vóór een shuffle verkleinen de hoeveelheid data die geshuffled moet worden.
- **Controleer partitieaantal na een zware filter** — een sterk gefilterde DataFrame behoudt vaak het oorspronkelijke, nu te hoge partitieaantal, wat overhead per taak veroorzaakt op steeds kleinere hoeveelheden data per partitie.
- **Gebruik kolomgeoriënteerde formaten (Parquet/Delta) als bron**, nooit CSV voor zware Spark-workloads — Catalyst kan dan kolomsnoei en predicate pushdown toepassen, wat bij CSV niet mogelijk is.
- **Monitor de Spark UI actief** bij prestatieproblemen — het toont per-stage en per-task-duur, en identificeert direct welke specifieke taak (vaak door skew) veel langer duurt dan de rest.

## Interviewvragen

**"Wat is het verschil tussen een transformation en een action in Spark?"** Let op: transformations zijn lui en bouwen alleen een uitvoeringsplan; actions triggeren de daadwerkelijke berekening — met een concreet voorbeeld van waarom een fout pas bij de action zichtbaar wordt.

**"Wat is een shuffle, en waarom is het duur?"** Let op: het herverdelen van data over partities heen zodat rijen met dezelfde sleutel samenkomen, wat serialisatie, netwerkverkeer en schijf-I/O vereist — wezenlijk duurder dan binnen-partitie-verwerking.

**"Wat is het verschil tussen wide en narrow transformations?"** Let op: narrow (filter, select) heeft geen shuffle nodig omdat elke output-partitie van precies één input-partitie afhangt; wide (groupBy, join) vereist wel een shuffle.

**"Wanneer zou je een broadcast join gebruiken, en waarom is het sneller dan een reguliere join?"** Let op: bij het joinen van een grote met een kleine tabel — de kleine tabel wordt naar elke executor gekopieerd, waardoor de join lokaal plaatsvindt zonder shuffle.

**"Wat is het verschil tussen `repartition` en `coalesce`?"** Let op: `repartition` shuffelt volledig en kan het aantal partities vergroten of verkleinen; `coalesce` combineert partities zonder volledige shuffle, maar alleen om te verkleinen.

**"Wat is data skew, en hoe herken je het?"** Let op: een ongelijke verdeling van data over partitiesleutels, herkenbaar doordat vrijwel alle tasks snel klaar zijn terwijl één of enkele executors veel langer blijven werken — zichtbaar in de Spark UI.

**"Wat doet Adaptive Query Execution, en waarom is het nodig naast Catalyst's vooraf-optimalisatie?"** Let op: Catalyst optimaliseert vóór uitvoering op basis van (soms onnauwkeurige) statistieken; AQE past het plan tijdens de run aan op basis van daadwerkelijk waargenomen tussenresultaten — bijvoorbeeld door een join dynamisch naar broadcast om te zetten of te kleine shuffle-partities samen te voegen.

## Relevante Documentatie

- Apache Spark: RDD Programming Guide — Transformations and Actions — de officiële uitleg van lazy evaluation.
- Apache Spark: Performance Tuning — shuffles, broadcast joins en Catalyst in detail.
- Databricks: Understanding the Spark UI — hoe je stages, tasks en skew in de praktijk identificeert.

## Samenvatting

PySpark's gedrag wordt volledig verklaard door een klein aantal onderliggende mechanismen: lazy evaluation (transformations bouwen een plan, actions voeren het uit), partitionering over executors als de fysieke basis van parallellisatie, en shuffles als de duurste operatie — het onderscheid tussen wide en narrow transformations formaliseert precies wanneer die shuffle optreedt. De Catalyst optimizer herschrijft plannen automatisch voor efficiëntie, maar `.explain()` blijft onmisbaar om te verifiëren dat die optimalisatie daadwerkelijk plaatsvond. Repartition/coalesce, broadcast joins en expliciet cachen zijn de directe hendels om controle te nemen over kosten die anders impliciet en onzichtbaar blijven. Met deze mechaniek begrepen, is het volgende hoofdstuk de logische vervolgstap: Delta Lake, de tabellaag die ACID-garanties toevoegt aan precies de gedistribueerde bestanden die Spark hier verwerkt.

HOOFDSTUK 16

# Delta Lake

## Introductie

Delta Lake is in eerdere hoofdstukken telkens genoemd als "de tabellaag die ACID-garanties toevoegt aan een lakehouse" — in Data Warehousing Fundamentals, Databricks, Microsoft Fabric — zonder ooit uit te leggen hóe dat precies werkt. Dat is het onderwerp van dit hoofdstuk: het mechanisme dat transacties, schema-garanties en versiehistorie mogelijk maakt bovenop iets dat in de kern nog steeds een verzameling gewone, onveranderlijke Parquet-bestanden is.

## De Transaction Log: het Hele Geheim in Eén Concept

Parquet-bestanden zelf zijn onveranderlijk en weten niets van elkaar — een map vol Parquet-bestanden is puur dat: bestanden. Delta Lake's volledige functionaliteit komt voort uit één aanvullend mechanisme: de **transaction log**, een map genaamd `_delta_log` naast de data, gevuld met genummerde JSON-bestanden (`000000.json`, `000001.json`, ...) die exact vastleggen welke Parquet-bestanden op elk moment "geldig" zijn.

```
orders_delta_table/
├── part-00000-xyz.parquet          ← ruwe databestanden, zelf onveranderlijk
├── part-00001-abc.parquet
└── _delta_log/
    ├── 00000000000000000000.json    ← commit 0: tabel aangemaakt, 2 bestanden toegevoegd
    ├── 00000000000000000001.json    ← commit 1: 1 bestand toegevoegd, 1 "verwijderd" (logisch)
    └── 00000000000000000002.json
```

Elk JSON-bestand beschrijft een **commit**: welke bestanden zijn toegevoegd (`add`), welke zijn logisch verwijderd (`remove` — het fysieke bestand blijft nog even bestaan, zie VACUUM verderop), en metadata zoals het schema op dat moment. Een "update" van een rij in Delta Lake schrijft dus **nooit** een bestaand Parquet-bestand aan — het schrijft een nieuw Parquet-bestand met de bijgewerkte data, en voegt een commit toe die het oude bestand als verwijderd markeert en het nieuwe als toegevoegd. Dit is de sleutel tot bijna alles wat volgt in dit hoofdstuk: ACID, time travel, en concurrency zijn allemaal directe gevolgen van dit ene ontwerp.

## Optimistic Concurrency Control

Omdat meerdere schrijfprocessen tegelijk tegen dezelfde Delta-tabel kunnen draaien, moet er een mechanisme zijn om conflicten te voorkomen zonder de tabel voor iedereen te vergrendelen (wat traditionele databases vaak wel doen, ten koste van doorvoersnelheid). Delta Lake gebruikt **optimistic concurrency control**: elk schrijfproces leest de huidige staat van de log, bereidt zijn wijziging voor, en probeert dan de *volgende* genummerde commit toe te voegen. Als een ander proces intussen al die volgende commit heeft geclaimd, faalt de poging, en Delta Lake probeert automatisch opnieuw — het herevalueert of de wijziging nog steeds veilig is gegeven de nieuwe staat, en committed dan als een latere versie.

Dit werkt goed zolang conflicten zeldzaam zijn (het gebruikelijke geval bij pipelines die verschillende delen van een tabel raken), maar twee schrijfprocessen die *dezelfde* rijen proberen te wijzigen, resulteren in een retry-poging die alsnog kan mislukken — relevant om te weten bij het ontwerpen van sterk parallelle laadpatronen, waar je bewust conflicterende writes wilt vermijden in plaats van op automatische retries te vertrouwen.

## Time Travel: de Log als Geschiedenisboek

Omdat elke commit een volledig, geordend overzicht is van welke bestanden op dat moment geldig waren, kan Delta Lake een query tegen elke eerdere versie beantwoorden door simpelweg de log terug te lezen tot dat punt — functioneel hetzelfde resultaat als Snowflake's Time Travel uit Snowflake, maar hier expliciet gebaseerd op de commit-geschiedenis in plaats van een intern, verborgen mechanisme:

```
-- Bevragen op versienummer
SELECT * FROM orders_delta TIMESTAMP AS OF '2026-08-10';
SELECT * FROM orders_delta VERSION AS OF 42;
df_old = spark.read.format("delta").option("versionAsOf", 42).load("/path/orders_delta")
```

## Schema Enforcement versus Schema Evolution

Standaard **handhaaft** Delta Lake het schema strikt: een write die kolommen bevat die niet in het bestaande schema passen (verkeerd type, onverwachte kolom), wordt geweigerd — precies het schema-contract-probleem uit Modern Data Platform Architecture, hier afgedwongen op storage-niveau in plaats van alleen via een extern data contract-document.

```
# Faalt standaard als het binnenkomende schema niet overeenkomt met het tabelschema
df.write.format("delta").mode("append").save("/path/orders_delta")

# Expliciete, bewuste schema-evolutie toestaan (nieuwe kolom toevoegen)
df.write.format("delta").mode("append").option("mergeSchema", "true").save("/path/orders_delta")
```

Het verschil met de schema-on-read-flexibiliteit van een pure data lake (zie Data Warehousing Fundamentals) is precies het punt: Delta Lake kiest bewust voor warehouse-achtige striktheid als standaardgedrag, met `mergeSchema` als expliciete, bewuste uitzondering — niet andersom. Dit voorkomt dat een stille schemawijziging in een bronsysteem een pipeline corrumpeert in plaats van hem direct, zichtbaar te laten falen.

## MERGE INTO: Upserts op wat Ooit Alleen-Lezen Was

Plain Parquet-bestanden ondersteunen geen `UPDATE` of `MERGE` — je kunt alleen hele bestanden lezen of nieuwe wegschrijven. Delta Lake's transaction log maakt `MERGE INTO` mogelijk door onder de motorkap precies te bepalen welke Parquet-bestanden rijen bevatten die moeten wijzigen, alleen díe bestanden te herschrijven, en de log dienovereenkomstig bij te werken — functioneel hetzelfde resultaat als de `MERGE`-patronen uit SQL for Data Engineers, nu mogelijk op bestandsopslag die dat van nature niet ondersteunt:

```
MERGE INTO silver.orders AS target
USING staging.orders_updates AS source
ON target.order_id = source.order_id
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
```

Dit is precies het mechanisme waarmee de SCD Type 2-patronen uit Star Schema & Dimensional Modeling en de idempotente laadpatronen uit Introduction to Data Engineering op een lakehouse worden geïmplementeerd, in plaats van alleen op een traditioneel warehouse.

## OPTIMIZE en Z-Ordering: het Kleine-Bestanden-Probleem

Frequente, kleine writes (elke paar minuten een nieuwe batch) resulteren in veel kleine Parquet-bestanden — inefficiënt om te lezen, omdat elk bestand een eigen open/close-overhead heeft, ongeacht hoe weinig data het bevat. `OPTIMIZE` compacteert kleine bestanden tot grotere, efficiëntere bestanden:

```
OPTIMIZE silver.orders;
OPTIMIZE silver.orders ZORDER BY (customer_id);
```

**Z-ordering** gaat een stap verder dan simpele compactie: het herordent data binnen de nieuwe, grotere bestanden zodat rijen met gerelateerde waarden voor de opgegeven kolom(men) fysiek dicht bij elkaar liggen — vergelijkbaar in doel met een clustering key in Snowflake uit Snowflake, maar met een andere onderliggende techniek (een Z-order-curve die meerdere kolommen tegelijk kan optimaliseren, in plaats van Snowflake's enkelvoudige clustering-sleutel-aanpak). Het resultaat is effectievere **file skipping**: een query die filtert op `customer_id` kan hele bestanden overslaan omdat de min/max-metadata van elk bestand aangeeft dat de gezochte waarde daar niet in voorkomt.

## VACUUM: de Spanning tussen Time Travel en Opslagkosten

Elke `MERGE`, `UPDATE` of `OPTIMIZE` laat de oude, nu logisch verwijderde Parquet-bestanden fysiek staan — nodig om time travel naar eerdere versies mogelijk te maken, maar het betekent dat opslag onbeperkt blijft groeien zonder opruiming. `VACUUM` verwijdert fysiek bestanden die ouder zijn dan de retentiedrempel (standaard 7 dagen) én niet meer worden gerefereerd door een recente commit:

```
VACUUM silver.orders RETAIN 168 HOURS;   -- 7 dagen, de standaard-ondergrens
```

De afweging is expliciet: een langere retentie behoudt meer time-travel-geschiedenis maar kost meer opslag; een kortere retentie bespaart opslag maar beperkt hoe ver je terug kunt kijken. Verlaag de retentie nooit lichtvaardig onder de standaard zonder te beseffen dat dit ook lopende, langlopende leesqueries kan breken als het bestand waar ze middenin lezen, wordt weggehaald.

## Change Data Feed: Rijwijzigingen Zelf Consumeren

Standaard toont een Delta-tabel alleen de huidige staat; **Change Data Feed (CDF)**, expliciet ingeschakeld per tabel, legt daarnaast vast wélke rijen zijn toegevoegd, gewijzigd of verwijderd per commit, inclusief een `_change_type`-kolom (`insert`, `update_preimage`, `update_postimage`, `delete`):

```
ALTER TABLE silver.orders SET TBLPROPERTIES (delta.enableChangeDataFeed = true);

SELECT * FROM table_changes('silver.orders', 10, 15);  -- wijzigingen tussen versie 10 en 15
```

Dit is direct bruikbaar om downstream-consumenten (een andere pipeline, een reverse ETL-proces zoals besproken in ETL vs ELT) alleen te laten reageren op wat daadwerkelijk is veranderd, in plaats van steeds de hele tabel opnieuw te verwerken — een incrementeel patroon dat een stuk preciezer is dan alleen op een `updated_at`-watermark vertrouwen.

## Deletion Vectors: Snel Verwijderen zonder Herschrijven

Een `DELETE` of `UPDATE` op een traditionele Delta-tabel herschrijft historisch het volledige Parquet-bestand dat de betreffende rijen bevat, ook als er maar één rij van de duizenden in dat bestand verandert — kostbaar bij grote bestanden en frequente, kleine wijzigingen. **Deletion vectors**, inmiddels standaard ingeschakeld op moderne Delta-versies, lossen dit op door verwijderde rijen te markeren in een apart, klein bestand naast het originele Parquet-bestand, zonder dat bestand zelf te herschrijven:

```
-- Met deletion vectors: dit markeert rijen als verwijderd i.p.v. bestanden te herschrijven
DELETE FROM silver.orders WHERE order_id = '12345';
```

Een latere `OPTIMIZE` compacteert deze markeringen alsnog fysiek weg door de betrokken bestanden op dat moment pas echt te herschrijven — deletion vectors verschuiven dus de kostbare herschrijfoperatie van "onmiddellijk bij elke DELETE" naar "periodiek, gebundeld, op een moment dat jij kiest", vergelijkbaar met hoe write-ahead logs in traditionele databases directe schrijfkosten uitstellen naar een latere, efficiëntere checkpoint-operatie.

## Liquid Clustering: Herclusteren zonder Herontwerp

Z-ordening hierboven vereist dat je vooraf een vaste set clusteringkolommen kiest, en het wijzigen daarvan betekent een volledige herschrijving van de tabel. **Liquid clustering** is Delta Lake's nieuwere antwoord hierop: clusteringkolommen kunnen worden aangepast zonder de bestaande data te herschrijven, en het systeem past incrementeel toe op nieuw geschreven data in plaats van alles in één keer opnieuw te ordenen:

```
CREATE TABLE silver.orders CLUSTER BY (customer_id, order_date);

-- Later, zonder de tabel te herbouwen, de clusteringkolommen aanpassen
ALTER TABLE silver.orders CLUSTER BY (order_date, region);
```

Voor tabellen waar querypatronen na verloop van tijd veranderen — precies het soort evolutie dat in de Flowmetric-case study uit Modern Data Platform Architecture van fase 1 naar fase 3 plaatsvindt — is dit een aanzienlijke operationele verbetering ten opzichte van Z-ordering's vaste-kolommen-aanpak: de clusteringstrategie kan meegroeien met de organisatie zonder een dure, disruptieve herbouw.

## Best Practices

- **Laat schema enforcement standaard aan staan**, en gebruik `mergeSchema` alleen bewust, per write, niet als permanente instelling die stille schemadrift toestaat.
- **Draai `OPTIMIZE` periodiek op tabellen met frequente, kleine writes** — het kleine-bestanden-probleem lost zichzelf niet op.
- **Kies Z-order-kolommen op basis van je meest voorkomende filterpatronen**, net als bij clustering keys — niet blindelings op elke kolom.
- **Stel een bewust retentiebeleid in voor `VACUUM`**, afgestemd op hoe ver terug je daadwerkelijk time travel nodig hebt, niet op de standaardwaarde zonder nadenken.
- **Gebruik Change Data Feed voor downstream-consumenten die specifiek op wijzigingen moeten reageren**, in plaats van een volledige tabel-rescan te forceren bij elke run.
- **Overweeg liquid clustering boven traditionele Z-ordering voor tabellen met evoluerende querypatronen**, zodat clusteringkolommen kunnen meegroeien zonder een dure, disruptieve herbouw.

## Veelgemaakte Fouten

- `mergeSchema` **standaard aanzetten** op elke write, waardoor stille, ongewenste schemawijzigingen ongemerkt een tabel binnensluipen.
- `OPTIMIZE` **nooit draaien** op een tabel met frequente kleine writes, waardoor leesprestaties geleidelijk verslechteren zonder duidelijke oorzaak.
- `VACUUM` **met een te korte retentie draaien** terwijl er nog langlopende queries of time-travel-afhankelijkheden actief zijn, wat die queries kan laten falen.
- **Denken dat plain Parquet en Delta Lake uitwisselbaar zijn** — een Delta-tabel is een map met Parquet-bestanden plus een `_delta_log`; die log direct negeren en de Parquet-bestanden los behandelen, breekt alle ACID-garanties.
- **Change Data Feed inschakelen op elke tabel "voor de zekerheid"**, wat onnodige opslag- en verwerkingskosten toevoegt voor tabellen waar niemand ooit de wijzigingsstroom consumeert.

## Performance Tips

- **Combineer `OPTIMIZE ZORDER BY` met je partitioneringskolom-strategie** — Z-ordering werkt binnen partities, dus een goed gekozen partitiekolom (vaak datum) blijft ook op een Delta-tabel relevant.
- **Plan `OPTIMIZE` buiten piekuren** — het is zelf een zware operatie die tijdelijk extra compute vraagt, net als reclustering bij Snowflake uit Snowflake.
- **Gebruik `VACUUM` regelmatig, niet incidenteel**, zodra retentie is verstreken — opgehoopte, nooit opgeruimde oude bestanden vertragen ook metadata-operaties zoals het lezen van de transaction log zelf.
- **Beperk het aantal kolommen in een Z-order-clausule tot twee of drie** — te veel kolommen tegelijk verdunt het skipping-voordeel per kolom, in plaats van het te versterken.

## Interviewvragen

**"Hoe voegt Delta Lake ACID-garanties toe aan iets dat in de kern gewone, onveranderlijke Parquet-bestanden zijn?"** Let op: de `_delta_log`-transaction-log, die per commit vastlegt welke bestanden geldig zijn — updates schrijven nooit een bestaand bestand aan, ze voegen nieuwe bestanden toe en markeren oude als verwijderd.

**"Wat is optimistic concurrency control, en hoe voorkomt het conflicten tussen gelijktijdige schrijfprocessen?"** Let op: elk proces probeert de volgende commit toe te voegen; bij een conflict (een ander proces was sneller) wordt automatisch herprobeerd, in plaats van de tabel voor iedereen te vergrendelen.

**"Wat is het verschil tussen schema enforcement en schema evolution in Delta Lake?"** Let op: enforcement is het standaardgedrag (afwijkende schema's worden geweigerd); evolution (`mergeSchema`) is een expliciete, bewuste uitzondering — nooit andersom.

**"Waarom is `VACUUM` nodig, en welke afweging maak je bij het instellen van de retentieperiode?"** Let op: oude, logisch verwijderde bestanden blijven fysiek bestaan voor time travel; een langere retentie behoudt meer geschiedenis maar kost meer opslag, een kortere kan lopende queries breken.

**"Wat is Z-ordering, en hoe verhoudt het zich tot een clustering key in Snowflake?"** Let op: beide optimaliseren fysieke data-layout voor filterprestaties via file/partition skipping, met Z-ordering's Z-curve-techniek als het onderliggende, iets andere mechanisme dan Snowflake's clustering keys.

**"Wat is Change Data Feed, en wanneer zou je het gebruiken?"** Let op: een expliciet ingeschakeld mechanisme dat rijwijzigingen (insert/update/delete) per commit blootlegt, nuttig voor downstream-consumenten die incrementeel willen reageren op wijzigingen in plaats van steeds de hele tabel te herverwerken.

**"Wat zijn deletion vectors, en welk probleem lossen ze op?"** Let op: ze markeren verwijderde rijen in een klein, apart bestand in plaats van direct het volledige onderliggende Parquet-bestand te herschrijven — de kostbare herschrijving wordt uitgesteld naar een latere, gebundelde `OPTIMIZE`-operatie.

**"Wat is het verschil tussen Z-ordering en liquid clustering?"** Let op: Z-ordering vereist vooraf vastgelegde kolommen en een volledige herschrijving om te wijzigen; liquid clustering laat clusteringkolommen aanpassen zonder bestaande data te herschrijven, wat beter meegroeit met veranderende querypatronen.

## Relevante Documentatie

- Delta Lake: Transaction Log Protocol — de technische specificatie van `_delta_log`.
- Databricks: Delta Lake — Optimize and Z-Order — compactie en Z-ordering in detail.
- Delta Lake: Table Utility Commands (VACUUM) — retentie en opruiming.
- Delta Lake: Change Data Feed — configuratie en het gebruik van `table_changes()`.

## Samenvatting

Delta Lake's volledige functionaliteit — ACID-transacties, time travel, schema-garanties — komt voort uit één mechanisme: de `_delta_log`-transaction-log die bijhoudt welke onveranderlijke Parquet-bestanden op elk moment geldig zijn, terwijl schrijfoperaties nooit bestaande bestanden aanpassen maar altijd nieuwe toevoegen. Optimistic concurrency control laat meerdere schrijvers veilig samenwerken zonder vergrendeling; `MERGE INTO` maakt upserts mogelijk op opslag die dat van nature niet ondersteunt. `OPTIMIZE`/Z-ordering en `VACUUM` zijn de operationele tegenhangers van respectievelijk het kleine-bestanden-probleem en de opslagkosten van behouden geschiedenis, en Change Data Feed ontsluit rijwijzigingen voor incrementele downstream-consumptie. Met deze mechaniek begrepen — van SQL, via medallion architecture, platformen, Python en Spark, tot de tabellaag zelf — verschuift het volgende hoofdstuk de aandacht naar hoe je dit alles betrouwbaar naar productie krijgt: CI/CD for Data Engineering.

HOOFDSTUK 17

# CI/CD voor Data Engineering

## Introductie

CI/CD-principes zijn in eerdere hoofdstukken al toegepast op specifieke situaties — het `adf_publish`-patroon in Azure Data Factory, Databricks Repos in Databricks, workspace-Git-integratie in Microsoft Fabric. Dit hoofdstuk behandelt de generieke principes erachter, en vooral waarom CI/CD voor data pipelines wezenlijk anders is dan voor gewone applicatiecode — een verschil dat vaak wordt onderschat door teams die CI/CD-praktijken direct overnemen uit software engineering zonder de data-specifieke complicaties te herkennen.

## Waarom CI/CD voor Data Anders Is

Applicatiecode is doorgaans **stateless** in de zin die hier telt: een nieuwe versie deployen vervangt de oude volledig, en een rollback naar de vorige versie is meestal net zo simpel als het vorige artefact opnieuw deployen. Data pipelines zijn dat niet — ze werken op **stateful** tabellen die historie opbouwen. Een gewijzigd transformatiemodel dat een fout bevat, kan al foute data hebben weggeschreven vóórdat het probleem wordt opgemerkt; een "rollback" van de code lost het probleem in de tabel zelf niet op, zoals verderop in dit hoofdstuk wordt uitgewerkt.

Een tweede verschil: **testen tegen realistische datavolumes is duur en traag.** Een unit test voor een webapplicatie draait in milliseconden; een dbt-model tegen een productiekopie van een tabel met miljarden rijen testen, kost warehouse-compute en tijd die niet past in een snelle CI-feedbackloop. Dit dwingt tot compromissen — kleinere testsets, steekproeven, of slimmer selecteren wát je test — die bij applicatiecode zelden nodig zijn.

## De Stadia: Lint, Test, Build, Deploy

Een typische CI/CD-pijplijn voor een dbt-project doorloopt vier stadia, elk met een specifiek data-engineering-doel.

**Lint** — SQL-stijl en -correctheid controleren vóórdat er iets draait, met tools als `sqlfluff`, dat SQL-conventies (consistente hoofdlettergebruik, geen `SELECT *`, consistente inspringing) afdwingt op dezelfde manier als een Python-linter code-stijl afdwingt.

**Test** — de dbt-tests uit Medallion Architecture en dbt Fundamentals daadwerkelijk uitvoeren tegen een test- of staging-omgeving, niet tegen productie.

**Build** — de modellen daadwerkelijk materialiseren in een tijdelijke of staging-schema, om te verifiëren dat de SQL syntactisch en logisch correct compileert en uitvoert op het doelplatform.

**Deploy** — bij succes, de wijziging daadwerkelijk naar productie promoten — het `adf_publish`-mechanisme uit Azure Data Factory is hier een platformspecifiek voorbeeld van; voor dbt is dit doorgaans simpelweg `dbt run` tegen het productie-target.

```
# .github/workflows/dbt_ci.yml (vereenvoudigd)
name: dbt CI
on: [pull_request]
jobs:
  ci:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pip install dbt-snowflake
      - run: sqlfluff lint models/ --dialect snowflake
      - run: dbt build --target ci
```

## dbt Slim CI: Alleen Bouwen wat Veranderde

Een naïeve CI-pijplijn draait `dbt build` tegen het *volledige* project bij elke pull request — correct, maar traag zodra een project honderden modellen bevat, en onnodig: een wijziging aan één silver-model raakt zelden meer dan een handvol downstream gold-modellen. **Slim CI** lost dit op met **state comparison**: dbt vergelijkt de huidige staat van het project met een opgeslagen `manifest.json` van de laatst succesvolle productierun, en bouwt/test alleen de modellen die daadwerkelijk zijn gewijzigd, plus hun downstream-afhankelijkheden:

```
dbt build --select state:modified+ --state ./prod-manifest
```

`state:modified` selecteert modellen waarvan de gecompileerde SQL is gewijzigd ten opzichte van het productie-manifest; de `+` erachter breidt dat uit naar alles wat daarvan afhangt (dezelfde `ref()`-graph-logica uit dbt Fundamentals, hier gebruikt om selectief te bouwen in plaats van om uitvoeringsvolgorde te bepalen). Voor een project met honderden modellen kan dit het verschil betekenen tussen een CI-run van twintig minuten en een van twee — direct relevant voor hoe snel een team feedback krijgt op een pull request, en dus hoe vaak ze bereid zijn kleine, veilige wijzigingen te maken in plaats van grote, risicovolle batches.

## Omgevingspromotie: Dev, Staging, Productie

De dev/prod-scheiding via `profiles.yml`-targets uit dbt Fundamentals is de basis; een volledige CI/CD-flow voegt daar een tussenliggende **staging**-omgeving aan toe, waar wijzigingen automatisch worden gedeployed en getest vóórdat een mens ze goedkeurt voor productie:

```
Feature branch → PR geopend → CI draait Slim CI tegen 'ci'-target
                                        ↓ (bij succes)
                              Merge naar main → auto-deploy naar 'staging'
                                        ↓ (na handmatige goedkeuring/smoke test)
                              Release-tag → deploy naar 'prod'
```

Dit patroon — automatisch tot staging, bewuste, vaak handmatige stap naar productie — balanceert snelheid (de meeste feedback komt automatisch en snel) met voorzichtigheid (de laatste stap naar productie, waar een fout het duurst is om te herstellen, blijft een bewuste beslissing).

## Secrets Management: Nooit in Code

Elke CI/CD-pijplijn heeft toegang nodig tot warehouse-credentials, API-sleutels, en andere gevoelige configuratie — en die horen nooit in code of in platte configuratiebestanden die met het project worden meegecommit. GitHub Actions, Azure DevOps en vergelijkbare platformen bieden een ingebouwde, versleutelde **secrets store**, waarbij de CI-pijplijn secrets als omgevingsvariabelen injecteert tijdens de run, zonder ze ooit in logs of in de repository zelf bloot te leggen:

```
- run: dbt build --target prod
  env:
    DBT_SNOWFLAKE_PASSWORD: ${{ secrets.SNOWFLAKE_PROD_PASSWORD }}
```

Voor complexere organisaties is een dedicated secrets-manager (Azure Key Vault, HashiCorp Vault, AWS Secrets Manager) de volgende stap — met als voordeel centrale rotatie en auditlogging van wie welk secret wanneer heeft opgehaald, iets wat een simpele CI-platform-secrets-store meestal niet biedt.

## Rollbacks: Waarom Data Pipelines Fundamenteel Lastiger Zijn

Bij een applicatie is een rollback meestal eenvoudig: deploy de vorige versie van de code opnieuw. Bij een data pipeline is de code maar de helft van het probleem — als een gewijzigd model al enkele uren lang foute data naar `gold.revenue_by_category` heeft geschreven vóórdat het probleem werd opgemerkt, lost het terugzetten van de oude code die al geschreven foute data niet met terugwerkende kracht op.

Dit is precies waarom de medallion-architectuur uit Medallion Architecture zo waardevol is in een CI/CD-context: omdat bronze onveranderlijk blijft, kan een rollback bestaan uit (1) de code terugzetten naar de vorige, correcte versie, én (2) de betrokken

silver/gold-tabellen **herbouwen vanaf bronze** met die herstelde code — iets dat zonder een bewaarde, ongewijzigde bronze-laag simpelweg niet mogelijk zou zijn. Voor Delta Lake-tabellen specifiek biedt time travel uit Delta Lake een extra, snellere optie: terugkeren naar een eerdere tabelversie zonder een volledige herberekening, mits het probleem recent genoeg is en binnen de VACUUM-retentie valt.

## Zero-Downtime Deployments: Tabellen Wisselen zonder Onderbreking

Een gold-tabel die halverwege een `TRUNCATE + INSERT`-deploy staat, is voor elke query die op dat moment binnenkomt tijdelijk leeg of incompleet — zichtbaar voor downstream-dashboards en -consumenten als een storing, ook al is de deploy zelf "gelukt". De gangbare oplossing is een **swap-patroon**: bouw de nieuwe versie van de tabel volledig onder een tijdelijke naam, en wissel pas daarna, in één atomaire operatie, de naam om met de bestaande tabel:

```
-- Bouw de nieuwe versie volledig, onder een tijdelijke naam
CREATE OR REPLACE TABLE gold.revenue_by_category_new AS
SELECT ... FROM silver.orders ...;

-- Atomaire wissel: geen moment waarop de tabel leeg of half-gevuld zichtbaar is
ALTER TABLE gold.revenue_by_category_new SWAP WITH gold.revenue_by_category;
DROP TABLE gold.revenue_by_category_new;   -- bevat nu de oude versie
```

Dit garandeert dat elke query die de tabel bevraagt, óf de volledige oude versie óf de volledige nieuwe versie ziet — nooit een tussentijdse, incomplete staat. Voor dbt-projecten wordt dit patroon vaak automatisch toegepast via de standaard `table`-materialization, die intern al een vergelijkbare create-and-swap-aanpak hanteert in plaats van een bestaande tabel direct te legen en opnieuw te vullen — relevant om te weten omdat het verklaart waarom een dbt-`run` zelden zichtbare downtime veroorzaakt, zelfs bij volledige herberekening van een druk bevraagde tabel.

## Testdata: het Spanningsveld tussen Realisme en Snelheid

CI-runs die tegen een volledige productiekopie testen, zijn traag en kostbaar; runs tegen volledig verzonnen, kleine testsets missen vaak precies de edge cases (rommelige waarden, onverwachte NULL's, schaalgerelateerde problemen) die een productie-incident veroorzaken. Twee gangbare compromissen adresseren dit.

**Seeds en gecureerde sample-sets** — een klein, met opzet samengesteld bestand (dbt seeds, zie dbt Fundamentals) dat bekende edge cases bevat: een order met een NULL-klant-ID, een extreem grote bestelling, een record met een ongebruikelijk teken in een tekstveld. Dit test gerichte scenario's snel, zonder een volledige datakopie nodig te hebben.

**Gemaskeerde productiesteekproeven** — een representatieve, willekeurige steekproef van echte productiedata, met gevoelige velden gemaskeerd of gepseudonimiseerd (dezelfde technieken als bij de ETL/ELT-compliance-afweging uit ETL vs ELT) vóór het als CI-testdata wordt gebruikt. Dit vangt realistische datakwaliteitsproblemen op die een handmatig samengestelde seed-set makkelijk mist, zonder de compliance-risico's van ongemaskeerde persoonsgegevens in een minder streng beveiligde CI-omgeving.

De praktische vuistregel: gebruik seeds voor gerichte, bekende edge cases die je bewust wilt blijven testen, en een periodiek ververste, gemaskeerde steekproef voor bredere, realistischere dekking — niet het een óf het ander, maar beide voor een ander doel.

## Best Practices

- **Gebruik Slim CI zodra een project meer dan een handvol modellen bevat** — de tijdsbesparing is direct voelbaar en stimuleert kleinere, veiligere pull requests.
- **Automatiseer deploys tot staging, houd productie-deploys bewust** (handmatige goedkeuring of een expliciete release-stap) — snelheid waar het kan, voorzichtigheid waar het telt.
- **Gebruik nooit hardcoded credentials**, ook niet "tijdelijk voor het testen" — dat tijdelijke wordt vrijwel altijd vergeten en blijft staan.
- **Behandel bronze als je rollback-vangnet.** Een medallion-architectuur die herbouwbaarheid garandeert (zie Medallion Architecture) is een directe, praktische CI/CD-asset, geen abstract architectuurprincipe.

- **Test lint- en syntaxfouten vroeg en goedkoop**, vóór je overgaat tot dure build/test-stadia tegen echte compute.
- **Gebruik het swap-patroon (of vertrouw op dbt's ingebouwde equivalent) voor elke druk bevraagde gold-tabel**, zodat deploys nooit zichtbaar zijn als tijdelijke storing voor eindgebruikers.
- **Combineer gerichte seeds met een periodiek ververste, gemaskeerde productiesteekproef** voor CI-testdata, in plaats van te kiezen voor slechts één van de twee.

## Veelgemaakte Fouten

- **Het volledige project bij elke PR bouwen** in plaats van Slim CI te gebruiken, wat CI-runs onnodig traag maakt en teams ontmoedigt om vaak, in kleine stappen te committen.
- **Rechtstreeks naar productie deployen zonder een staging-tussenstap**, waardoor de eerste keer dat een wijziging tegen realistische data draait, ook meteen de eerste keer is dat het productie raakt.
- **Secrets in `profiles.yml` of vergelijkbare configuratiebestanden hardcoderen** en per ongeluk meecommitten naar een repository.
- **Rollback van alleen de code verwachten zonder de reeds geschreven foute data te herstellen** — een veelgemaakte misvatting die pas bij het eerste echte incident pijnlijk duidelijk wordt.
- **Geen lint-stap vóór de dure build/test-stadia**, waardoor triviale syntaxfouten pas laat in de pijplijn — en dus langzamer — worden ontdekt.
- **Direct `TRUNCATE + INSERT` gebruiken op een druk bevraagde tabel** in plaats van een swap-patroon, waardoor gebruikers tijdens elke deploy een leeg of incompleet resultaat kunnen zien.
- **Alleen handmatig samengestelde, kleine testsets gebruiken**, waardoor CI stelselmatig de rommelige edge cases mist die alleen in echte productiedata voorkomen.

## Performance Tips

- **Gebruik `state:modified+` in elke PR-CI-run**, en reserveer een volledige `dbt build` voor een periodieke (bijvoorbeeld nachtelijke) validatie tegen het complete project.
- **Cache dependency-installaties** (Python packages, dbt packages) tussen CI-runs waar het CI-platform dat ondersteunt, om herhaalde downloadtijd te vermijden.
- **Parallelliseer onafhankelijke CI-stadia** (lint en een deel van de tests kunnen vaak gelijktijdig draaien) in plaats van alles strikt sequentieel uit te voeren.
- **Gebruik een kleinere, representatieve testset voor CI-builds** in plaats van een volledige productiekopie, en bewaar zware, volledige-datavolumetests voor een aparte, minder frequente validatiecyclus.

## Interviewvragen

**"Waarom is CI/CD voor data pipelines fundamenteel anders dan voor applicatiecode?"** Let op: data pipelines zijn stateful (een rollback van code lost al geschreven foute data niet op) en testen tegen realistische volumes is duur/traag — twee complicaties die bij de meeste applicatiecode niet spelen.

**"Wat is dbt Slim CI, en hoe werkt state comparison?"** Let op: het vergelijken van de huidige projectstaat met een opgeslagen productiemanifest om alleen gewijzigde modellen (plus hun downstream-afhankelijkheden via `+`) te bouwen en testen, in plaats van het hele project.

**"Hoe zou je een rollback aanpakken als een gewijzigd dbt-model al enkele uren foute data heeft geschreven?"** Let op: de code terugzetten is niet genoeg — de betrokken tabellen moeten herbouwd worden vanaf een onveranderlijke bronze-laag (of hersteld via Delta Lake time travel), niet alleen de codewijziging ongedaan gemaakt.

**"Hoe zou je secrets beheren in een CI/CD-pijplijn voor een dataplatform?"** Let op: nooit hardcoded in code of configuratiebestanden; gebruik de ingebouwde secrets-store van het CI-platform of een dedicated secrets-manager, geïnjecteerd als

omgevingsvariabelen tijdens de run.

**"Beschrijf een typische omgevingspromotie-flow voor een dbt-project."** Let op: feature branch → PR met Slim CI → merge naar main met auto-deploy naar staging → bewuste, vaak handmatige stap naar productie — automatisering waar veilig, een menselijke controle waar de impact het grootst is.

**"Hoe zorg je ervoor dat een tabel-deploy geen tijdelijke downtime of incomplete data veroorzaakt?"** Let op: het swap-patroon — de nieuwe versie volledig onder een tijdelijke naam bouwen, dan atomair van naam wisselen met de bestaande tabel, zodat elke query óf de volledige oude óf de volledige nieuwe versie ziet, nooit een tussenstaat.

**"Hoe zou je CI-testdata samenstellen zonder de kosten en risico's van een volledige productiekopie?"** Let op: een combinatie van gerichte seeds voor bekende edge cases en een periodiek ververste, gemaskeerde productiesteekproef voor bredere, realistischere dekking — niet volledig verzonnen data en niet ongemaskeerde productiedata.

## Relevante Documentatie

- dbt Labs: About State and Deferral — de officiële uitleg van `state:modified` en gerelateerde selectors.
- dbt Labs: Slim CI — de aanbevolen CI-workflow met state comparison.
- GitHub Actions: Encrypted Secrets — secrets management in GitHub Actions specifiek.
- sqlfluff Documentation — SQL-linting-configuratie en -regels.

## Samenvatting

CI/CD voor data engineering bouwt voort op dezelfde principes als applicatie-CI/CD — lint, test, build, deploy, geautomatiseerde omgevingspromotie — maar met twee fundamentele complicaties die specifieke oplossingen vereisen: data pipelines zijn stateful, waardoor een codewijziging terugdraaien niet automatisch de al geschreven foute data herstelt, en testen tegen realistische volumes is duur, wat Slim CI's state-comparison-aanpak (alleen gewijzigde modellen bouwen) tot een noodzakelijke, niet optionele optimalisatie maakt zodra een project groeit. Medallion architecture's onveranderlijke bronze-laag is hier niet alleen een architectuurprincipe maar een direct bruikbaar rollback-mechanisme, en het swap-patroon zorgt ervoor dat elke deploy zelf al onzichtbaar blijft voor eindgebruikers, los van of er later alsnog een rollback nodig blijkt. Met CI/CD als proces behandeld, verschuift het volgende hoofdstuk naar het fundament eronder: Git zelf, en de branchingstrategieën die deze hele workflow mogelijk maken.

HOOFDSTUK 18

# Git & Branch-strategieën

## Introductie

Git en branches zijn in bijna elk voorgaand hoofdstuk terloops genoemd — Databricks Repos in Databricks, de `adf_publish`-branch in Azure Data Factory, de feature-branch-naar-staging-flow in CI/CD for Data Engineering. Dit hoofdstuk behandelt Git zelf en, belangrijker, de branchingstrategie-vraag specifiek voor data engineering: welke aanpak past bij hoe dataeams daadwerkelijk werken, en waarom dat vaak een ander antwoord oplevert dan bij traditionele softwareteams.

## Git-Fundamenten: het Objectmodel in het Kort

Voor wie voornamelijk uit een SQL-achtergrond komt, is het waard even stil te staan bij wat Git onder de motorkap daadwerkelijk vastlegt, omdat het de meeste verwarrende Git-situaties verklaart. Een **commit** is geen "diff" — het is een volledige, onveranderlijke snapshot van de complete projectstructuur op dat moment, met een verwijzing naar zijn ouder-commit(s). Een **branch** is niets meer dan een beweeglijk label dat naar een specifieke commit wijst — wanneer je een nieuwe commit maakt, verschuift het label van de huidige branch automatisch mee.

Dit verklaart waarom **merge** en **rebase** verschillende geschiedenissen opleveren voor hetzelfde eindresultaat: een merge voegt een nieuwe commit toe die twee ouder-commits samenbrengt (de geschiedenis vertakt zichtbaar en komt weer samen), terwijl een rebase de commits van jouw branch een voor een herschrijft bovenop de nieuwste stand van de doelbranch (de geschiedenis blijft lineair, alsof je vanaf het begin op de nieuwste code had gewerkt). Voor data engineering-projecten, waar een lineaire, makkelijk te doorzoeken geschiedenis vaak waardevoller is dan een exacte weergave van wanneer welke branch bestond, geven veel teams de voorkeur aan rebase voor feature branches en reserveren ze merge-commits voor het samenvoegen naar `main` zelf.

## Branchingstrategieën Vergeleken

Drie strategieën domineren de discussie, elk met een andere balans tussen structuur en snelheid.

**Git Flow** gebruikt langlevende `develop`- en `main`-branches, met aparte `feature/`, `release/` en `hotfix/`-branches die volgens een strikt patroon samenkomen. Het biedt veel structuur en expliciete release-momenten, maar de overhead is aanzienlijk — voor de meeste data engineering-teams, die zelden discrete "releases" uitbrengen zoals een versioned softwareproduct, is dit vaak meer proces dan nodig.

**GitHub Flow** is aanzienlijk eenvoudiger: één langlevende `main`-branch, korte feature branches die direct vanuit `main` worden getakt en er via een pull request weer in worden gemerged, met continue deployment na elke merge. Dit past goed bij teams die vaak, in kleine stappen, naar productie deployen — precies het patroon uit de CI/CD-flow in CI/CD for Data Engineering.

**Trunk-based development** gaat nog een stap verder: extreem korte of zelfs geen feature branches, met wijzigingen die binnen uren (niet dagen) terug in `main` landen, vaak achter feature flags voor grotere wijzigingen die nog niet volledig klaar zijn. Dit minimaliseert merge-conflicten door simpelweg de tijd te beperken waarin twee branches uit elkaar kunnen groeien.

## Waarom Dataeams Vaak voor Eenvoudigere Flows Kiezen

Data engineering-projecten hebben twee kenmerken die de balans richting eenvoudigere strategieën (GitHub Flow, trunk-based) doen doorslaan, vergeleken met bijvoorbeeld een groot softwareproduct met discrete versienummers.

Ten eerste: **dbt-modellen en SQL-bestanden conflicteren vervelender dan de meeste applicatiecode.** Een gegenereerd of sterk geherformatteerd SQL-bestand waar twee feature branches allebei aan hebben gewerkt, levert vaak merge-conflicten op die niet regel-

voor-regel op te lossen zijn zonder de onderliggende businesslogica van beide wijzigingen te begrijpen — anders dan bijvoorbeeld twee losse functies in verschillende delen van een applicatiebestand, die Git meestal probleemloos automatisch samenvoegt. Kortere branches met snellere merges beperken direct hoe lang zulke conflicten de kans krijgen om te ontstaan.

Ten tweede: **data pipelines hebben zelden discrete "releases" in de zin van een softwareproduct.** Er is geen versie 2.0 van een dataplatform die gebruikers bewust installeren — er is een continu stromende, incrementeel verbeterde reeks modellen. Git Flow's release-branches lossen een probleem op (het coördineren van een versioned, installeerbaar product) dat de meeste dataplatformen simpelweg niet hebben.

De praktische vuistregel: begin met GitHub Flow als standaard voor een dataplatform-repository, en overweeg Git Flow specifiek alleen voor herbruikbare, versioned artefacten zoals een gedeeld dbt-package dat door meerdere, onafhankelijke projecten wordt geïmporteerd — daar geldt wél een traditioneel versioning-argument.

## Branches en Omgevingen: het Algemene Patroon

CI/CD for Data Engineering toonde de feature-branch-naar-staging-naar-productie-flow al concreet; hier het onderliggende, platformonafhankelijke principe dat elk platformspecifiek voorbeeld in dit handbook — Databricks Repos, ADF's collaboration branch, Fabric's workspace-Git-koppeling — deelt: **een branch (of een specifieke stand daarvan) correspondeert met een omgeving**, en het promoveren van een wijziging naar een volgende omgeving is letterlijk het mergen van die branch naar de volgende:

```
feature/add-churn-model  →  main (= staging-omgeving)  →  release-tag (= productie)
```

Wat per platform verschilt, is puur de mechaniek van hoe die merge daadwerkelijk een omgeving bijwerkt — dbt-targets, ADF's `adf_publish`-ARM-templates, Databricks' workspace-synchronisatie — niet het onderliggende Git-principe zelf.

## Merge-Conflicten in SQL/dbt-Projecten Beperken

Naast kortere branches helpen een paar concrete gewoontes om de vervelende SQL-merge-conflicten hierboven te minimaliseren. **Kleinere, gerichte pull requests** — één model of één logisch samenhangende set wijzigingen per PR, in plaats van een grote PR die tien ongerelateerde modellen tegelijk aanraakt — verkleinen zowel de kans op conflicten als de moeite om ze op te lossen wanneer ze toch optreden. **Consistente SQL-formattering** (via `sqlfluff format` uit CI/CD for Data Engineering, automatisch toegepast vóór commit) voorkomt dat triviale opmaakverschillen — spaties, hoofdlettergebruik — zich vermengen met daadwerkelijke logica-conflicten en die onnodig lastiger maken om te doorzien.

## Commit Message Conventies

**Conventional Commits** (`feat:`, `fix:`, `docs:`, `refactor:`) is een lichte, veelgebruikte conventie die commit-berichten machineleesbaar maakt, wat automatische changelog-generatie en semantic versioning mogelijk maakt — vooral relevant voor gedeelde dbt-packages die andere projecten als dependency importeren, zoals genoemd bij de packages-sectie in dbt Fundamentals:

```
feat(orders): voeg incrementele laadstrategie toe aan fct_orders
fix(customers): corrigeer dubbele klant-ID's door hoofdletterongevoelige match
docs: leg grain-conventie uit in silver-modellen
```

Voor een intern dataplatform zonder gedeelde packages is dit minder kritiek dan bij open-source of gedeelde libraries, maar de discipline van een duidelijk, gestructureerd commit-bericht — wat is er veranderd, en waarom — blijft waardevol voor iedereen die maanden later probeert te achterhalen waarom een specifieke wijziging is gemaakt, zonder de oorspronkelijke auteur te hoeven vragen.

## Branch Protection: main Beschermen tegen Zichzelf

Een productie-`main`-branch zonder bescherming staat toe dat iemand per ongeluk direct pusht, CI overslaat, of een PR merget zonder review — precies de risico's die de zorgvuldig opgebouwde CI/CD-flow uit CI/CD for Data Engineering ondermijnen. **Branch protection rules** (op GitHub, GitLab, Azure DevOps vergelijkbaar beschikbaar) dwingen af dat:

- Directe pushes naar `main` niet zijn toegestaan — alles moet via een pull request.
- Een pull request pas gemerged kan worden nadat de CI-checks (lint, Slim CI, tests) succesvol zijn afgerond.
- Minstens één andere engineer de wijziging heeft goedgekeurd vóór een merge.

Dit is geen bureaucratie om zichzelf — het is de technische afdwinging van precies de "geen enkele wijziging bereikt productie zonder review en groen licht van CI"-garantie die de rest van de CI/CD-flow probeert te bieden.

## Een Merge-Conflict Ontrafelen: een Werkvoorbeeld

Concreet, zodat het geen abstract probleem blijft: twee engineers werken parallel aan `models/gold/fct_orders.sql`. Engineer A voegt een kolom toe voor belastingberekening; engineer B wijzigt onafhankelijk de filterlogica voor geannuleerde orders. Bij het mergen toont Git:

```
<<<<<<< HEAD
SELECT
    order_id,
    amount,
    amount * 0.21 AS tax_amount,
    status
FROM {{ ref('stg_orders') }}
WHERE status != 'cancelled'
=======
SELECT
    order_id,
    amount,
    status
FROM {{ ref('stg_orders') }}
WHERE status NOT IN ('cancelled', 'test')
>>>>>>> feature/exclude-test-orders
```

Dit is precies het scenario waar automatische samenvoeging faalt: beide wijzigingen zijn geldig en moeten allebei behouden blijven, maar Git kan niet weten dat `tax_amount` en de uitgebreide `WHERE`-clausule onafhankelijke, allebei-gewenste wijzigingen zijn. De juiste oplossing vereist begrip van beide bedoelingen, niet alleen tekstuele samenvoeging:

```
SELECT
    order_id,
    amount,
    amount * 0.21 AS tax_amount,
    status
FROM {{ ref('stg_orders') }}
WHERE status NOT IN ('cancelled', 'test')
```

Dit is exact waarom kleinere, snellere pull requests er in de praktijk toe doen: hoe korter engineer A en B's branches naast elkaar bestonden vóór het mergen, hoe kleiner de kans dat hun wijzigingen elkaar op deze manier kruisten in de eerste plaats.

## Monorepo versus Polyrepo voor Dataplatformen

Een vraag die vroeg in een project moet worden beslist: horen alle dbt-modellen, Python-scripts en infrastructuurcode van een dataplatform in **één repository** (monorepo), of verspreid over **meerdere, aparte repositories** per team of domein (polyrepo)?

Een **monorepo** maakt cross-domein wijzigingen eenvoudiger (één pull request kan tegelijk een silver-model en de gold-modellen die ervan afhangen aanpassen), geeft iedereen zicht op de volledige codebase, en vereenvoudigt Slim CI uit CI/CD for Data Engineering — één `manifest.json` dekt het hele project. De keerzijde: naarmate een organisatie groeit, kan een monorepo een knelpunt worden voor onafhankelijke team-releasecycli, en toegangscontrole op repository-niveau wordt grover (lastiger om het finance-team alleen toegang te geven tot finance-modellen als alles in dezelfde repo staat).

Een **polyrepo**-aanpak, met een aparte repository per domein of team (vergelijkbaar met hoe Unity Catalog-catalogi in Databricks governance per domein scheiden), geeft teams meer autonomie en fijnmaziger toegangsbeheer, ten koste van meer overhead bij cross-domein wijzigingen en een lastiger te onderhouden, gefragmenteerde Slim CI-configuratie over meerdere repositories heen.

De praktische vuistregel: begin met een monorepo voor een dataplatform tot de organisatie een omvang bereikt (meerdere onafhankelijke teams met eigen releasecycli en strikte onderlinge toegangsscheiding) die de overhead van polyrepo daadwerkelijk rechtvaardigt — vroegtijdig opsplitsen voegt vooral coördinatiekosten toe zonder een navenant voordeel.

## Best Practices

- **Kies GitHub Flow of trunk-based development als standaard** voor een dataplatform-repository, en reserveer Git Flow specifiek voor gedeelde, versioned dbt-packages.
- **Houd feature branches kort-levend** (dagen, niet weken) om SQL-merge-conflicten te minimaliseren door simpelweg de tijd te beperken waarin branches uit elkaar groeien.
- **Automatiseer SQL-formattering vóór commit**, zodat opmaakverschillen nooit vermengd raken met daadwerkelijke logicawijzigingen in een merge-conflict.
- **Stel branch protection in op `main`** vanaf het eerste moment dat een dataplatform-repository productie voedt, niet pas na het eerste incident.
- **Gebruik kleine, gerichte pull requests** — één logisch samenhangende wijziging per PR is makkelijker te reviewen én te mergen dan een grote, verzamelde batch.
- **Begin met een monorepo** voor een nieuw dataplatform, en overweeg polyrepo pas expliciet zodra meerdere onafhankelijke teams met eigen releasecycli en strikte toegangsscheiding daar daadwerkelijk om vragen.

## Veelgemaakte Fouten

- **Langlevende feature branches aanhouden** die weken uit `main` blijven groeien, wat de kans op pijnlijke, moeilijk oplosbare SQL-merge-conflicten aanzienlijk vergroot.
- **Git Flow overnemen zonder na te denken of het bij het project past** — veel van de release-branch-complexiteit lost een probleem op (versioned, installeerbare releases) dat de meeste dataplatformen niet hebben.
- **Geen branch protection op `main`**, waardoor een enkele, ongereviewde direct-push de zorgvuldig opgebouwde CI/CD-garanties omzeilt.
- **Enorme, verzamelde pull requests** die tien ongerelateerde modellen tegelijk wijzigen, wat reviews vertraagt en conflicten moeilijker te ontrafelen maakt.
- **Inconsistente SQL-formattering tussen teamleden**, waardoor elke merge onnodige, puur cosmetische conflicten met zich meebrengt bovenop de echte logicawijzigingen.
- **Te vroeg opsplitsen in polyrepo's** vanuit een aanname over toekomstige schaal, wat vooral coördinatie-overhead toevoegt zolang de organisatie die complexiteit nog niet daadwerkelijk nodig heeft.

## Performance Tips

- **Houd gegenereerde of grote binaire bestanden buiten Git** (via `.gitignore`) — compiled dbt-artefacten, lokale virtual environments, en grote testdatasets horen niet in versiebeheer en vertragen elke clone en fetch onnodig.
- **Gebruik shallow clones** (`git clone --depth 1`) in CI-omgevingen waar de volledige geschiedenis niet nodig is, om checkout-tijd te verkorten.
- **Squash kleine, opeenvolgende "fix typo"-commits** vóór het mergen van een feature branch, zodat de geschiedenis van `main` leesbaar blijft in plaats van vervuild te raken met ruis.
- **Prune verouderde, al-gemergede branches regelmatig**, zowel lokaal als remote, om de repository overzichtelijk te houden naarmate een project groeit.

## Interviewvragen

**"Wat is het verschil tussen merge en rebase, en wanneer zou je welke gebruiken?"** Let op: merge behoudt de vertakkende geschiedenis met een expliciete merge-commit; rebase herschrijft commits bovenop de nieuwste doelbranch voor een lineaire geschiedenis — rebase vaak binnen een feature branch, merge (via PR) om terug naar `main` te gaan.

**"Waarom kiezen veel data engineering-teams voor GitHub Flow of trunk-based development in plaats van Git Flow?"** Let op: dataplatformen hebben zelden discrete, versioned releases zoals een softwareproduct, en langlevende branches vergroten de kans op lastige SQL/dbt-merge-conflicten — eenvoudigere flows sluiten beter aan bij hoe dataeams daadwerkelijk werken.

**"Waarom zijn merge-conflicten in SQL/dbt-projecten vaak lastiger dan in applicatiecode?"** Let op: gegenereerde of sterk gestructureerde SQL laat zich niet altijd regel-voor-regel automatisch samenvoegen zonder de onderliggende businesslogica van beide wijzigingen te begrijpen — anders dan bijvoorbeeld twee losse functies in verschillende bestandsdelen.

**"Wat doet branch protection, en waarom is het nodig naast een CI/CD-pijplijn?"** Let op: het dwingt technisch af dat wijzigingen alleen via een gereviewde, CI-goedgekeurde pull request naar `main` kunnen — zonder deze afdwinging blijft de zorgvuldig opgebouwde CI/CD-garantie afhankelijk van discipline in plaats van techniek.

**"Hoe zou je merge-conflicten in een dbt-project structureel minimaliseren?"** Let op: kortere feature branches, kleinere/gerichte pull requests, en geautomatiseerde, consistente SQL-formattering vóór commit — een combinatie van procesmatige en technische maatregelen.

**"Wanneer zou je voor een monorepo kiezen bij een dataplatform, en wanneer voor polyrepo?"** Let op: monorepo vereenvoudigt cross-domein wijzigingen en Slim CI voor kleinere tot middelgrote teams; polyrepo geeft meer autonomie en fijnmaziger toegangscontrole naarmate een organisatie meerdere, onafhankelijke teams met eigen releasecycli krijgt — begin met monorepo tot de overhead van polyrepo daadwerkelijk gerechtvaardigd is.

## Relevante Documentatie

- Git: Pro Git Book — Branching — de officiële, diepgaande uitleg van Git's objectmodel en branches.
- GitHub: About Protected Branches — configuratie van branch protection rules.
- Conventional Commits — de specificatie voor gestructureerde commit-berichten.
- dbt Labs: Git and Version Control Best Practices — dbt-specifieke aanbevelingen voor branching en samenwerking.

## Samenvatting

Git's objectmodel — onveranderlijke commits, branches als beweeglijke labels — verklaart waarom merge en rebase verschillende geschiedenissen opleveren voor hetzelfde resultaat. Van de drie dominante branchingstrategieën passen GitHub Flow en trunk-based development doorgaans beter bij data engineering-teams dan Git Flow, omdat dataplatformen zelden discrete, versioned releases hebben en langlevende branches de kans op lastige SQL/dbt-merge-conflicten vergroten. Branches corresponderen met omgevingen — hetzelfde onderliggende principe dat Databricks Repos, ADF's `adf_publish` en Fabric's Git-integratie elk op hun eigen manier implementeren. Branch protection, kleine gerichte pull requests en consistente formattering zijn de concrete, technische maatregelen die deze principes afdwingen in plaats van aan discipline over te laten. Met Git als fundament behandeld, verschuift het volgende hoofdstuk naar een ander DevOps-bouwsteen: Docker, en hoe containerisatie past binnen data engineering-workflows.

HOOFDSTUK 19

# Docker voor Data Engineers

## Introductie

Python for Data Engineers loste het "werkt bij mij niet in productie"-probleem op voor Python-dependencies via virtual environments — maar een venv isoleert alleen Python-packages, niet de Python-versie zelf, niet systeembibliotheken, niet specifieke versies van command-line tools waar een pipeline mogelijk van afhangt. Docker lost dat bredere probleem op: een container legt de **volledige runtime-omgeving** vast — besturingssysteem-niveau dependencies, exacte taal- en toolversies, configuratiebestanden — zodat "het werkt bij mij" wordt vervangen door "het werkt overeal waar deze container draait", zonder uitzonderingen.

## Containers versus Virtual Machines

Het onderscheid is de moeite waard om precies te begrijpen, omdat het verklaart waarom containers zo lichtgewicht zijn. Een **virtuele machine** virtualiseert volledige hardware, inclusief een eigen kernel — zwaar, traag om op te starten (minuten), maar volledig geïsoleerd. Een **container** deelt de kernel van het hostbesturingssysteem en isoleert alleen het proces zelf (bestandssysteem, netwerk, processen) via kernel-features (namespaces, cgroups) — aanzienlijk lichter, start in seconden, maar met iets minder isolatie dan een volledige VM. Voor de meeste data engineering-doeleinden (een reproduceerbare, geïsoleerde runtime voor een pipeline-stap) is die iets mindere isolatie een acceptabele afweging tegen de aanzienlijke snelheids- en resourcewinst.

## Images en Layers: hoe een Container Wordt Opgebouwd

Een Docker **image** is een onveranderlijke blauwdruk, opgebouwd uit een reeks **layers** — elke instructie in een Dockerfile (`RUN`, `COPY`, `FROM`) creëert een nieuwe layer, die Docker cachet. Bij een volgende build wordt elke layer alleen opnieuw uitgevoerd als de instructie zelf óf de bestanden die hij gebruikt, zijn veranderd — een principe dat direct de bouwsnelheid van je CI/CD-pijplijn beïnvloedt, relevant voor de build-stap uit CI/CD for Data Engineering.

```
FROM python:3.12-slim

WORKDIR /app

# Dependencies EERST kopiëren en installeren — deze layer wordt gecached
# en alleen herbouwd als requirements.txt zelf verandert
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Pas DAARNA de broncode kopiëren — deze layer verandert het vaakst,
# maar de dure dependency-installatie hierboven blijft gecached
COPY . .

CMD ["python", "extract_orders.py"]
```

De volgorde hier is bewust, niet toevallig: `requirements.txt` kopiëren en installeren gebeurt vóór het kopiëren van de rest van de broncode, omdat broncode veel vaker wijzigt dan dependencies. Zou je de volgorde omdraaien, dan zou elke wijziging aan de broncode — ook een kleine, cosmetische — de volledige, tijdrovende `pip install`-layer opnieuw triggeren, omdat Docker's layer-cache elke wijziging vroeg in de keten alle daaropvolgende layers ongeldig maakt.

## Docker Compose voor Lokale Ontwikkeling

Een enkele pipeline heeft vaak een lokale database of andere afhankelijkheid nodig om tegen te ontwikkelen en testen — **Docker Compose** definieert meerdere samenwerkende containers in één configuratiebestand, direct relevant voor de testdata-strategieën uit CI/CD for Data Engineering:

```
# docker-compose.yml
services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_DB: orders_test
      POSTGRES_PASSWORD: localdev
    ports:
      - "5432:5432"

  pipeline:
    build: .
    depends_on:
      - postgres
    environment:
      DB_HOST: postgres   # containers verwijzen naar elkaar via de servicenaam
docker compose up   # start beide containers, met de juiste netwerkverbindingen ertussen
```

Dit geeft elke engineer een identieke, geïsoleerde lokale testomgeving — een lokale Postgres-instantie die niemand anders raakt, opgezet en afgebroken in seconden, zonder een gedeelde, mogelijk door anderen gebruikte database te hoeven benaderen.

## Containers per Taak in Orchestratoren

Airflow (zie Introduction to Data Engineering) en vergelijkbare orchestratoren kunnen elke individuele taak in een eigen, geïsoleerde container draaien in plaats van alles in dezelfde Python-omgeving als de orchestrator zelf — relevant zodra verschillende pipeline-stappen conflicterende dependency-versies nodig hebben (een taak die een oudere versie van een ML-library vereist naast een taak die de nieuwste versie van een andere library nodig heeft, wat in één gedeelde omgeving simpelweg niet oplosbaar is):

```
from airflow.providers.cncf.kubernetes.operators.pod import KubernetesPodOperator

extract_task = KubernetesPodOperator(
    task_id="extract_orders",
    image="myregistry/extract-orders:1.4.0",   # eigen, geïsoleerde image per taak
    cmds=["python", "extract_orders.py"],
    namespace="data-pipelines",
)
```

Elke taak draait dan in zijn eigen container, met exact de dependencies die die specifieke taak nodig heeft — volledige isolatie tussen taken binnen dezelfde pipeline, zonder dat de orchestrator zelf enige kennis hoeft te hebben van de inhoud van elke container.

## Volumes: Data Overleeft de Container

Containers zijn standaard **ephemeral**: alles wat binnen een container wordt weggeschreven, verdwijnt zodra die container stopt en wordt verwijderd — prima voor de stateloze verwerkingslogica van een pipeline zelf, problematisch voor de Postgres-container uit het Compose-voorbeeld hierboven, waar je juist wilt dat testdata blijft bestaan tussen sessies. Een **volume** koppelt een map buiten de container (op de hostmachine, of beheerd door Docker zelf) aan een pad binnen de container, zodat data die daar wordt geschreven, de levensduur van de container zelf overleeft:

```
services:
  postgres:
    image: postgres:16
    volumes:
      - pgdata:/var/lib/postgresql/data   # data overleeft 'docker compose down'
    environment:
      POSTGRES_PASSWORD: localdev

volumes:
  pgdata:
```

Zonder deze volume-declaratie zou elke `docker compose down` gevolgd door `docker compose up` een volledig lege database opleveren — met de volume blijft de data behouden, terwijl de container zelf nog steeds probleemloos vervangen, herstart of herbouwd kan worden. Dit onderscheid — code en configuratie in de image, data in een volume — is een van de meest fundamentele ontwerpprincipes van

containerized systemen, en het verklaart waarom "gewoon de container verwijderen en opnieuw starten" een veilige, routinematige operatie is zolang state correct in volumes leeft.

## Omgevingsvariabelen en Secrets in Containers

Net als bij de CI/CD-secrets uit CI/CD for Data Engineering horen wachtwoorden en API-sleutels nooit gebakken in een image zelf — ze worden **tijdens het starten** van de container meegegeven, niet tijdens het bouwen ervan:

```
# Fout: het geheim wordt onderdeel van de image-geschiedenis, permanent zichtbaar
RUN export DB_PASSWORD=supersecret

# Goed: wordt pas bij het starten van de container geïnjecteerd, nooit in de image zelf
docker run -e DB_PASSWORD="${DB_PASSWORD}" myregistry/extract-orders:1.4.0
```

Voor productieomgevingen is een expliciete `docker run -e`-vlag met een lokaal beschikbare waarde vaak niet praktisch of veilig genoeg — orchestratoren zoals Kubernetes bieden eigen **secret-objecten** die als omgevingsvariabelen of gemounte bestanden in een container worden geïnjecteerd, beheerd en versleuteld los van de container-definitie zelf, vergelijkbaar in geest met de secrets-store-aanpak uit CI/CD for Data Engineering maar dan toegepast op runtime-containers in plaats van CI-pijplijnen.

## Docker in de Platformen die je Al Kent

De platformen uit eerdere hoofdstukken abstraheren containers doorgaans grotendeels weg — je schrijft een dbt-model of een notebook, niet een Dockerfile — maar containers duiken op specifieke, herkenbare plekken toch weer op. **Snowpark Container Services** (Snowflake, zie Snowflake) laat je eigen, custom containerized workloads binnen Snowflake's beheerde infrastructuur draaien, voor logica die noch pure SQL noch Snowpark's Python-ondersteuning comfortabel dekt. **Databricks Container Services** laat je clusters starten vanaf een custom Docker-image in plaats van Databricks' standaard runtime-image, nuttig wanneer een team zeer specifieke, van de standaard afwijkende systeemdependencies nodig heeft die niet via een reguliere library-installatie op te lossen zijn. Het patroon is in beide gevallen hetzelfde als bij de `KubernetesPodOperator` hierboven: het platform beheert de orchestratie en infrastructuur, jij levert alleen het containerized deel dat specifieke, afwijkende requirements heeft.

## Multi-Stage Builds: Kleinere, Veiligere Images

Een naïeve Dockerfile installeert vaak build-tools (compilers, headerbestanden) die alleen nodig zijn tijdens het bouwen, maar niet tijdens het daadwerkelijk draaien van de applicatie — wat onnodig grote images oplevert met een groter aanvalsoppervlak. **Multi-stage builds** scheiden een build-fase van een runtime-fase, en nemen alleen het uiteindelijke resultaat mee naar het kleinere, uiteindelijke image:

```
# Stage 1: bouwen, met alle benodigde build-tools
FROM python:3.12 AS builder
COPY requirements.txt .
RUN pip install --user -r requirements.txt

# Stage 2: alleen het resultaat meenemen, geen build-tools in het uiteindelijke image
FROM python:3.12-slim
COPY --from=builder /root/.local /root/.local
COPY . .
CMD ["python", "extract_orders.py"]
```

Het resultaat is een aanzienlijk kleiner uiteindelijk image — sneller te downloaden bij elke deploy, minder onnodige software die potentiële kwetsbaarheden kan bevatten, zonder dat de build-fase zelf minder krachtig hoeft te zijn.

## Best Practices

- **Kopieer dependency-bestanden vóór de broncode** in elke Dockerfile, om Docker's layer-cache maximaal te benutten en onnodige rebuilds te vermijden.

- **Gebruik multi-stage builds standaard** voor elke image die build-tools nodig heeft die niet in de uiteindelijke runtime horen.
- **Gebruik specifieke, vastgepinde image-tags** (`python:3.12-slim`, niet `python:latest`) — dezelfde reden als vastgepinde dependency-versies in Python for Data Engineers: reproduceerbaarheid boven gemak.
- **Gebruik Docker Compose voor lokale ontwikkelomgevingen** die een database of andere afhankelijkheid nodig hebben, in plaats van een gedeelde, mogelijk instabiele testomgeving te benaderen.
- **Draai elke pipeline-taak met conflicterende dependencies in zijn eigen container**, in plaats van te proberen alle mogelijke versiecombinaties in één gedeelde omgeving te verzoenen.
- **Declareer volumes expliciet voor elke container die state moet behouden** (een lokale database, opgeslagen testresultaten) — behandel het ontbreken van een volume als een bewuste keuze voor ephemeral gedrag, niet als een vergeten detail.
- **Injecteer secrets altijd bij het starten van een container, nooit tijdens het bouwen** — een `RUN`-instructie die een geheim gebruikt, laat dat geheim permanent in de image-laag-geschiedenis staan, ook na een latere "opruim"-instructie.

## Veelgemaakte Fouten

- **Broncode vóór dependencies kopiëren in een Dockerfile**, waardoor elke kleine codewijziging een volledige, trage dependency-herinstallatie triggert.
- **`latest` als image-tag gebruiken** in plaats van een specifieke versie, waardoor een build op een willekeurig moment een andere, mogelijk incompatibele basisimage oppikt.
- **Build-tools meenemen in het uiteindelijke runtime-image** zonder multi-stage builds te gebruiken, wat onnodig grote, minder veilige images oplevert.
- **Eén enorme, gedeelde container proberen te bouwen** die alle mogelijke dependency-versies voor elke pipeline-taak tegelijk bevat, in plaats van taken met conflicterende behoeften te isoleren.
- **Geheimen (wachtwoorden, API-sleutels) direct in een Dockerfile hardcoderen** — ze blijven dan permanent in de image-geschiedenis staan, ook als een latere layer ze "verwijdert".
- **Geen volume declareren voor een lokale databasecontainer**, waardoor testdata bij elke `docker compose down` verdwijnt en teamleden onnodig steeds opnieuw hun testomgeving moeten opbouwen.
- **Aannemen dat containers een vervanging zijn voor alle platformspecifieke compute** — de meeste dagelijkse dbt/Snowflake/Databricks-werk vereist geen Dockerfile; containers zijn specifiek waardevol voor de afwijkende, isolatie-vereisende gevallen die dit hoofdstuk beschrijft.

## Performance Tips

- **Gebruik `.dockerignore`** om onnodige bestanden (lokale virtual environments, `.git`, testdata) buiten de build-context te houden — een kleinere context betekent een snellere build.
- **Combineer gerelateerde `RUN`-instructies** waar mogelijk tot één layer, om het totale aantal layers en de bijbehorende overhead te beperken.
- **Gebruik een `-slim`- of `-alpine`-basisimage** waar compatibel, in plaats van een volledige, zware basisimage die functionaliteit bevat die een pipeline nooit gebruikt.
- **Cache dependency-layers expliciet in CI** (via registry-caching of vergelijkbare mechanismen van je CI-platform) zodat niet elke CI-run vanaf nul dezelfde dependencies opnieuw hoeft te downloaden en installeren.
- **Vermijd onnodig veel, kleine volumes** voor data die eigenlijk prima ephemeral kan zijn — elke volume is een stukje state dat iemand op enig moment moet begrijpen, beheren en opruimen.

## Interviewvragen

**"Wat is het verschil tussen een container en een virtuele machine?"** Let op: een container deelt de kernel van het hostsysteem en isoleert alleen het proces (lichter, sneller); een VM virtualiseert volledige hardware inclusief een eigen kernel (zwaarder, meer

geïsoleerd).

**"Waarom is de volgorde van instructies in een Dockerfile belangrijk?"** Let op: Docker's layer-cache maakt elke wijziging vroeg in de keten alle daaropvolgende layers ongeldig — dependencies vóór broncode kopiëren voorkomt onnodige, trage rebuilds bij elke kleine codewijziging.

**"Waarom zou je Docker gebruiken als je al een Python virtual environment hebt?"** Let op: een venv isoleert alleen Python-packages, niet OS-niveau dependencies, systeembibliotheken, of de Python-versie zelf — Docker legt de volledige runtime-omgeving vast, niet alleen een deel ervan.

**"Wat is een multi-stage build, en welk probleem lost het op?"** Let op: het scheidt een build-fase (met build-tools) van een runtime-fase, zodat alleen het uiteindelijke resultaat — niet de build-tools zelf — in het uiteindelijke, kleinere image terechtkomt.

**"Waarom zou je in een orchestrator elke pipeline-taak in een eigen container laten draaien in plaats van in één gedeelde omgeving?"** Let op: verschillende taken kunnen conflicterende dependency-versies nodig hebben die in één gedeelde omgeving simpelweg niet naast elkaar kunnen bestaan — eigen containers per taak lossen dat isolatieprobleem op.

**"Wat is een Docker volume, en waarom heb je die nodig voor een lokale databasecontainer?"** Let op: containers zijn standaard ephemeral — alles binnen de container verdwijnt bij verwijdering; een volume koppelt een pad buiten de container aan een pad erbinnen, zodat data (zoals databestanden) de levensduur van de container overleeft.

**"Hoe zou je een wachtwoord veilig aan een container meegeven, zonder het in de image zelf te bakken?"** Let op: injecteren tijdens het starten van de container (via `-e` of een orchestrator-specifiek secret-mechanisme), nooit via een `RUN`-instructie in de Dockerfile die het geheim permanent in de image-geschiedenis vastlegt.

**"Waar duiken containers op binnen platformen als Snowflake en Databricks, ook al schrijf je meestal geen Dockerfile?"** Let op: Snowpark Container Services voor custom workloads binnen Snowflake, Databricks Container Services voor clusters met een aangepaste basisimage — het platform beheert de orchestratie, jij levert alleen het containerized deel met afwijkende requirements.

## Relevante Documentatie

- Docker: Dockerfile Best Practices — officiële richtlijnen voor layer-ordening en caching.
- Docker: Compose Overview — configuratie van meerdere samenwerkende containers.
- Apache Airflow: KubernetesPodOperator — taken uitvoeren in geïsoleerde containers binnen Airflow.
- Docker: Multi-Stage Builds — officiële uitleg en voorbeelden.

## Samenvatting

Docker lost een breder probleem op dan een Python virtual environment: het legt de volledige runtime-omgeving vast, niet alleen taalspecifieke dependencies, via lichte, kernel-gedeelde containers in plaats van zware, volledig gevirtualiseerde machines. Layer-caching maakt de volgorde van Dockerfile-instructies direct relevant voor buildsnelheid; Docker Compose geeft elke engineer een identieke, geïsoleerde lokale ontwikkelomgeving. In orchestratoren laat het draaien van elke taak in zijn eigen container conflicterende dependency-versies tussen pipeline-stappen naast elkaar bestaan, en multi-stage builds houden uiteindelijke images klein door build-tools buiten de runtime te houden. Volumes bewaren state waar dat nodig is, secrets horen bij het starten geïnjecteerd te worden, nooit gebakken in een image, en zelfs platformen als Snowflake en Databricks vallen voor hun meest afwijkende workloads terug op ditzelfde containerprincipe. Met DevOps-bouwstenen — CI/CD, Git, Docker — nu behandeld, verschuift het volgende hoofdstuk terug naar de data zelf: Data Quality & Testing.

HOOFDSTUK 20

# Datakwaliteit & Testen

## Introductie

Testing is in dit handbook al concreet behandeld — de teststrategie per medallion-laag in Medallion Architecture, generic versus singular tests in dbt Fundamentals, testdata-strategie in CI/CD in CI/CD for Data Engineering. Dit hoofdstuk herhaalt die dbt-mechaniek niet, maar behandelt datakwaliteit als vakgebied op zich: het denkkader waarmee je "kwaliteit" concreet maakt, testvormen die verder gaan dan vaste regels als `not_null`, en wat er operationeel gebeurt wanneer een test daadwerkelijk faalt.

## De Zes Datakwaliteitsdimensies

Voordat je kunt testen, moet je kunnen benoemen wát je precies test. De industrie heeft zich grotendeels rond zes dimensies geschaard, en elke concrete test die je schrijft — of het nu een dbt-test is of iets geavanceerders — valt in de kern onder een van deze:

**Accuraatheid** — komt de waarde overeen met de werkelijkheid? Een adres dat bestaat maar niet klopt met waar de klant daadwerkelijk woont, is inaccuraat maar niet per se incompleet of ongeldig.

**Volledigheid** — ontbreken er waarden die er wel zouden moeten zijn? Dit is wat `not_null`-tests uit dbt Fundamentals meten, maar volledigheid gaat breder: ontbreken er hele rijen (een dag zonder orders terwijl de winkel open was)?

**Consistentie** — komt dezelfde entiteit overeen tussen verschillende systemen of tabellen? Een klant met verschillende geboortedatums in het CRM en het factuursysteem schendt consistentie, ook al is elke waarde op zich mogelijk geldig.

**Tijdigheid** — is de data vers genoeg voor het gebruiksdoel? Dit is precies waar freshness-checks uit Medallion Architecture op inspelen, en waar Monitoring & Observability verder op ingaat.

**Validiteit** — voldoet een waarde aan het verwachte formaat of bereik? Een e-mailadres zonder `@`, een leeftijd van -5 — dit is het domein van `accepted_values`- en range-tests.

**Uniciteit** — komt een entiteit precies één keer voor waar dat verwacht wordt? Dit is wat `unique`-tests bewaken, en de reden dat deduplicatie uit SQL for Data Engineers zo'n centrale plek inneemt.

Het nut van dit denkkader: als een stakeholder zegt "de data klopt niet", dwingt deze indeling je om te vragen *welke* dimensie precies faalt — want de oplossing voor een volledigheidsprobleem (ontbrekende rijen onderzoeken bij de bron) is compleet anders dan voor een consistentieprobleem (twee systemen synchroniseren).

## Vaste Regels versus Anomalie-gebaseerd Testen

dbt's generic tests (`not_null`, `unique`, `accepted_values`) zijn **deterministisch**: een vaste, vooraf bekende regel die een rij wel of niet doorstaat. Dit werkt uitstekend voor validiteit en uniciteit, maar schiet tekort bij problemen die zich pas tonen als **afwijking van een patroon**, niet als schending van een vaste regel.

Stel: het aantal orders per dag ligt normaliter tussen 800 en 1200. Een dag met 3 orders schendt geen enkele `not_null`- of `unique`-regel — elke individuele rij is perfect geldig — maar er is overduidelijk iets mis met de ingestion. Dit is waar **anomaliedetectie** in beeld komt: tests die niet een vaste waarde controleren, maar of een metriek significant afwijkt van zijn historische verdeling.

```
-- Eenvoudige, zelfgebouwde anomaliecheck: vergelijk vandaag met een historisch gemiddelde
WITH daily_counts AS (
    SELECT order_date, COUNT(*) AS order_count
    FROM silver.orders
    WHERE order_date >= CURRENT_DATE - 30
    GROUP BY order_date
```

```
),
stats AS (
    SELECT AVG(order_count) AS avg_count, STDDEV(order_count) AS stddev_count
    FROM daily_counts
    WHERE order_date < CURRENT_DATE
)
SELECT d.order_date, d.order_count, s.avg_count
FROM daily_counts d, stats s
WHERE d.order_date = CURRENT_DATE
  AND ABS(d.order_count - s.avg_count) > 3 * s.stddev_count;   -- >3 standaarddeviaties afwijking
```

Packages als `dbt_expectations` (genoemd in dbt Fundamentals) bieden kant-en-klare varianten hiervan, maar het onderliggende principe blijft hetzelfde: een grens gebaseerd op historisch gedrag, niet op een hardgecodeerde waarde die iemand ooit heeft bedacht.

## De Testpiramide Toegepast op Data

Softwareontwikkeling kent de testpiramide: veel snelle, goedkope unit tests aan de basis, minder integratietests, een klein aantal trage end-to-end-tests aan de top. Hetzelfde principe vertaalt zich naar dataplatformen, met een net iets andere invulling per laag.

**Unit tests op transformatielogica** — test een SQL-functie of macro geïsoleerd, met kunstmatige, kleine input, zonder een heel warehouse te raken. dbt ondersteunt dit via unit tests op modelniveau: je geeft vaste input-rijen op en controleert de exacte output, vergelijkbaar met een unit test in reguliere software.

**Integratietests op pipelines** — controleer of een volledige bronze-naar-silver-naar-gold-keten correct doorloopt met een kleine, representatieve testset (de seeds/steekproef-afweging uit CI/CD for Data Engineering), zonder tegen volledige productievolumes te draaien.

**Data tests op productiedata zelf** — dit zijn de generic/singular dbt-tests uit dbt Fundamentals en de anomaliechecks hierboven, uitgevoerd tegen de daadwerkelijke, actuele data. Dit is de enige laag die niet vervangen kan worden door een kleinere testset, omdat het punt juist is de echte data te controleren.

De veelgemaakte fout is alle drie de lagen door elkaar te gebruiken: dure, volledige data-tests bij elke code-commit draaien (traag, kostbaar) in plaats van snelle unit tests voor logicafouten en data-tests specifiek voor wat alleen op echte data zichtbaar wordt.

## Wat Gebeurt er als een Test Faalt? Quarantaine versus Blokkeren

Een test die faalt, roept een operationele vraag op die dbt's testframework zelf niet beantwoordt: moet de hele pipeline stoppen, of moet alleen de foute data apart gezet worden terwijl de rest doorgaat? Twee patronen, met een directe parallel in de `expect`/`expect_or_drop`/`expect_or_fail`-niveaus van Delta Live Tables uit Databricks.

**Hard falen (blokkeren)** — de pipeline stopt volledig zodra een kritieke test faalt, en niets stroomafwaarts wordt bijgewerkt met mogelijk onbetrouwbare data. Geschikt voor tests op fundamentele garanties (een primary key die niet uniek is, een verplicht veld dat massaal ontbreekt) waar doorgaan erger is dan stilstaan.

**Quarantaine (zachte afhandeling)** — rijen die een test niet doorstaan, worden apart gezet in een quarantainetabel voor onderzoek, terwijl geldige rijen gewoon doorstromen naar de volgende laag. Geschikt voor problemen die een klein deel van de data raken en waar de rest van de pipeline niet hoeft te wachten op een handmatig onderzoek van een randgeval.

```
-- Quarantainepatroon: scheid geldige van ongeldige rijen, blokkeer niets
CREATE OR REPLACE TABLE silver.orders AS
SELECT * FROM staging.orders_clean
WHERE order_id IS NOT NULL AND amount >= 0;

CREATE OR REPLACE TABLE silver.orders_quarantine AS
SELECT *, CURRENT_TIMESTAMP() AS quarantined_at
FROM staging.orders_clean
WHERE order_id IS NULL OR amount < 0;
```

De keuze tussen deze twee is zelden platformbreed — net als bij de ETL/ELT-keuze uit ETL vs ELT is dit een beslissing per test, afhankelijk van hoe kritiek en hoe wijdverspreid het risico van doorgaan is.

## Voorbij dbt: Great Expectations en Soda

dbt's ingebouwde tests dekken veel, maar gespecialiseerde datakwaliteitstools bieden functionaliteit die dbt zelf niet heeft. **Great Expectations** werkt met "expectation suites" — declaratieve, herbruikbare verzamelingen verwachtingen die zowel binnen als buiten een dbt-project kunnen draaien, met uitgebreide, automatisch gegenereerde datakwaliteitsrapporten (data docs) als bijproduct:

```
import great_expectations as gx

context = gx.get_context()
validator = context.sources.pandas_default.read_csv("orders.csv")

validator.expect_column_values_to_not_be_null("order_id")
validator.expect_column_values_to_be_between("amount", min_value=0, max_value=100000)
validator.expect_table_row_count_to_be_between(min_value=500, max_value=5000)
```

**Soda Core** volgt een vergelijkbare filosofie maar met een lichtgewicht, YAML-gebaseerde syntax (SodaCL), specifiek ontworpen om eenvoudig in bestaande pipelines te integreren zonder een aparte Python-omgeving op te tuigen:

```
# checks.yml
checks for silver.orders:
  - row_count between 500 and 5000
  - missing_count(order_id) = 0
  - duplicate_count(order_id) = 0
```

De praktische vuistregel: gebruik dbt's ingebouwde tests als standaard voor alles wat al binnen je dbt-project leeft — het is al aanwezig, geen extra tool nodig. Grijp naar Great Expectations of Soda specifiek wanneer je datakwaliteit wilt bewaken op plekken die dbt niet bereikt (ruwe bestanden vóór ze het warehouse bereiken, of een gedeeld kwaliteitsraamwerk dat meerdere teams met verschillende transformatietools moeten kunnen gebruiken).

## Contract Tests: Testen op de Grens Tussen Teams

Modern Data Platform Architecture introduceerde het data contract als afspraak tussen een producerend en consumerend team, vastgelegd als YAML. Het contract wordt pas waardevol zodra het ook daadwerkelijk **getest** wordt — niet als documentatie die niemand meer leest, maar als een CI-check die faalt zodra de werkelijkheid van het contract afwijkt:

```
# test_contract_orders.py — draait in CI van het producerende team
import yaml

def test_orders_schema_matches_contract():
    contract = yaml.safe_load(open("contracts/orders.yml"))
    actual_columns = get_live_schema("production.orders")  # query tegen de bron

    contract_columns = {c["name"]: c["type"] for c in contract["contract"]["schema"]}
    for name, expected_type in contract_columns.items():
        assert name in actual_columns, f"Contractkolom '{name}' ontbreekt in de bron"
        assert actual_columns[name] == expected_type, f"'{name}' heeft type {actual_columns[name]}, contract verwacht {expected_type}"
```

Dit draait in de CI-pijplijn van het **producerende** team (het productteam uit het Flowmetric-voorbeeld), zodat een schemawijziging die het contract breekt al wordt gevangen vóórdat die code naar productie gaat — niet pas wanneer het datateam een week later een mysterieus falende dbt-run ontdekt. Dit is het verschil tussen een data contract als *intentie* en een data contract als *afgedwongen garantie*, en het is precies de reden dat contract-tests in bredere organisaties steeds vaker als verplichte CI-stap voor bronsystemen worden ingericht.

## Testdekking Meten en Bewaken

Een vraag die zelden gesteld wordt maar wel zou moeten: hoeveel van je kritieke modellen hebben eigenlijk tests? Zonder dit te meten groeit een dbt-project vaak organisch met tests op de eerste paar modellen, gevolgd door tientallen nieuwe modellen zonder enige

test omdat niemand het structureel bewaakt. dbt's eigen manifest (dezelfde `manifest.json` die Slim CI uit CI/CD for Data Engineering gebruikt) bevat genoeg informatie om dit te berekenen:

```
import json

manifest = json.load(open("target/manifest.json"))
models = {k: v for k, v in manifest["nodes"].items() if v["resource_type"] == "model"}
tested_models = {m for t in manifest["nodes"].values() if t["resource_type"] == "test"
                  for m in t.get("depends_on", {}).get("nodes", []) if m in models}

coverage = len(tested_models) / len(models) * 100
print(f"Testdekking: {coverage:.1f}% ({len(tested_models)}/{len(models)} modellen)")
```

Net als codedekking in reguliere software is 100% zelden het doel — sommige modellen zijn triviaal genoeg dat een test weinig toevoegt — maar het *zichtbaar maken* van dekking, bijvoorbeeld als vast onderdeel van een sprint-review of kwartaalrapportage, voorkomt dat kritieke gold-modellen jarenlang ongetest blijven simpelweg omdat niemand het ooit expliciet controleerde.

## Best Practices

- **Koppel elke test expliciet aan een datakwaliteitsdimensie** in je documentatie — het maakt duidelijk wélk probleem een falende test signaleert, niet alleen dát er iets faalt.
- **Gebruik anomaliedetectie naast vaste regels**, niet in plaats daarvan — vaste regels vangen bekende schendingen, anomaliedetectie vangt onbekende patroonverschuivingen.
- **Kies quarantaine als standaard, hard falen als uitzondering** — reserveer blokkerende tests voor garanties waarvan schending de hele downstream-keten onbetrouwbaar maakt.
- **Gebruik unit tests voor logicafouten, data-tests voor datafouten** — laat data-tests niet de taak van unit tests overnemen, dat maakt CI onnodig traag.
- **Reserveer Great Expectations/Soda voor wat dbt niet bereikt**, niet als vervanging van dbt's ingebouwde tests waar die al volstaan.

## Veelgemaakte Fouten

- **Alleen op vaste regels vertrouwen**, waardoor een geleidelijke, patroonmatige verslechtering (een langzaam dalend ordervolume) onopgemerkt blijft omdat geen enkele individuele rij een regel schendt.
- **Elke falende test hard laten blokkeren**, ook voor kleine, randgevallen — wat pipelines fragiel en operationeel vermoeiend maakt.
- **Quarantainetabellen aanmaken zonder ooit iemand te laten kijken wat erin staat** — quarantaine zonder opvolging is alleen uitstel van hetzelfde probleem.
- **Dure data-tests bij elke commit draaien** in plaats van ze te reserveren voor waar ze daadwerkelijk nodig zijn, wat CI onnodig traag maakt.
- **Een aparte tool (Great Expectations/Soda) invoeren zonder concrete leemte** die dbt's eigen tests niet al dekken — onnodige toolingcomplexiteit.

## Performance Tips

- **Laat anomaliechecks draaien op vooraf geaggregeerde metrieken** (dagelijkse tellingen, niet ruwe rijen), zodat de check zelf snel blijft ongeacht hoe groot de onderliggende tabel is.
- **Beperk quarantaine-checks tot de silver-laag**, niet elke laag — bronze hoort sowieso ruw te blijven, gold bouwt al voort op reeds gevalideerde silver-data.
- **Cache historische statistieken voor anomaliedetectie** (gemiddelde, standaarddeviatie) in plaats van ze bij elke run opnieuw over de volledige historie te berekenen.
- **Parallelliseer onafhankelijke tests** in CI waar mogelijk, dezelfde overweging als bij Slim CI uit CI/CD for Data Engineering.

## Interviewvragen

**"Noem de zes datakwaliteitsdimensies en geef van elk een voorbeeld."** Let op: accuraatheid, volledigheid, consistentie, tijdigheid, validiteit, uniciteit — met een concreet, onderscheidend voorbeeld per dimensie, niet alleen de namen opsommen.

**"Wat is het verschil tussen een vaste-regel-test en anomaliedetectie?"** Let op: vaste regels controleren een vooraf bekende grens (not_null, range); anomaliedetectie vergelijkt met een historisch patroon en signaleert onverwachte afwijkingen die geen enkele vaste regel zou vangen.

**"Wanneer zou je een falende test laten blokkeren versus de foute rijen laten quarantaine?"** Let op: blokkeren voor fundamentele garanties waar doorgaan de hele downstream-keten onbetrouwbaar maakt; quarantaine voor lokale, beperkte problemen waar de rest van de pipeline niet hoeft te wachten.

**"Leg de testpiramide uit zoals toegepast op een dataplatform."** Let op: unit tests op transformatielogica (snel, geïsoleerd), integratietests op pipelines met representatieve testdata, data-tests op de daadwerkelijke productiedata — elk met een andere snelheid/kostenafweging.

**"Wanneer zou je Great Expectations of Soda gebruiken naast dbt's eigen tests?"** Let op: specifiek voor datakwaliteitscontrole buiten het bereik van dbt (ruwe bestanden vóór het warehouse, gedeelde kwaliteitsraamwerken over meerdere tools heen) — niet als standaardvervanging van dbt-tests.

**"Hoe zou je een data contract daadwerkelijk afdwingen, niet alleen documenteren?"** Let op: een contract-test die in de CI-pijplijn van het producerende team draait en faalt zodra het werkelijke schema afwijkt van het contract — vóórdat de wijziging productie bereikt, niet pas wanneer een downstream-consument iets ziet breken.

**"Hoe zou je meten of een dbt-project voldoende testdekking heeft?"** Let op: het aantal modellen met minstens één gekoppelde test afzetten tegen het totale aantal modellen, afleidbaar uit `manifest.json` — en het besef dat 100% zelden het doel is, maar zichtbaarheid wel voorkomt dat kritieke modellen structureel ongetest blijven.

## Relevante Documentatie

- dbt Labs: Unit Tests — geïsoleerde tests op transformatielogica met kunstmatige input.
- Great Expectations Documentation — expectation suites en data docs in detail.
- Soda Core Documentation — SodaCL-syntax en integratiepatronen.
- dbt Labs: dbt_expectations Package — statistische en distributie-tests bovenop dbt.

## Samenvatting

De zes datakwaliteitsdimensies — accuraatheid, volledigheid, consistentie, tijdigheid, validiteit, uniciteit — geven taal aan wat "kwaliteit" concreet betekent, en elke test die je schrijft valt in de kern onder een ervan. Vaste regels (dbt's generic tests) vangen bekende schendingen; anomaliedetectie vangt patroonverschuivingen die geen enkele vaste regel zou signaleren. De testpiramide (unit, integratie, data) bepaalt welke laag welke snelheid en dekking verdient, en de keuze tussen hard blokkeren en quarantaine bepaalt wat er operationeel gebeurt zodra een test daadwerkelijk faalt. Great Expectations en Soda vullen aan waar dbt's eigen tests niet reiken. Met datakwaliteit als denkkader op zak, behandelt het volgende hoofdstuk het bredere organisatorische raamwerk eromheen: Data Governance.

HOOFDSTUK 21

# Datagovernance

## Introductie

Governance-mechanica is al concreet aan bod gekomen — rolgebaseerde toegang en lineage via Unity Catalog in Databricks, de governance-sectie in Modern Data Platform Architecture, data contracts als afgedwongen CI-check in Data Quality & Testing. Dit hoofdstuk herhaalt die techniek niet, maar behandelt governance als organisatorische discipline: wie is verantwoordelijk voor wat, hoe classificeer je data zodat beleid uitvoerbaar wordt, en hoe organiseer je governance zodat het een dataplatform niet vertraagt maar juist mogelijk maakt.

## Wat Data Governance Werkelijk Is

Een veelgemaakte vergissing: governance gelijkstellen aan toegangscontrole. RBAC (wie mag welke tabel lezen) is een **afdwingingsmechanisme**, niet governance zelf. Governance is het geheel van beslissingen over wie data mag *definiëren*, wie verantwoordelijk is voor de *kwaliteit* ervan, welke *regels* gelden voor gevoelige data, en hoe die regels *consistent* worden toegepast over een organisatie — RBAC is slechts één van de technische middelen om die beslissingen af te dwingen, naast classificatietags, retentiebeleid en catalogisering.

Zonder dit onderscheid eindigen organisaties met technisch uitstekend beveiligde platformen (strikte RBAC, versleuteling, audit logs) die alsnog governance-problemen hebben: niemand weet wie een tabel "eigenaar" is, dezelfde klant heet in drie systemen anders, en een analist gebruikt een verouderde definitie van "actieve klant" zonder dat iemand het merkt. Governance lost dát op — het is een organisatorisch, niet uitsluitend technisch vraagstuk.

## Classificatieniveaus: de Basis van Elk Beleid

Voordat je beleid kunt afdwingen ("gevoelige data mag niet naar analisten"), moet je kunnen benoemen *wat* gevoelig is. De meeste organisaties werken met een classificatiehiërarchie van vier niveaus:

**Publiek** — informatie die vrijelijk gedeeld kan worden, zoals gepubliceerde productprijzen of marketingcontent. Geen beperkingen nodig.

**Intern** — bedrijfsinformatie die niet voor externen bedoeld is, maar geen direct risico vormt bij interne verspreiding — operationele metrics, interne rapportages.

**Vertrouwelijk** — data die schade kan veroorzaken bij verkeerde verspreiding: financiële resultaten vóór publicatie, strategische plannen, salarisgegevens.

**Restricted (persoonsgegevens/PII)** — data die onder wettelijke bescherming valt (AVG/GDPR): namen, e-mailadressen, BSN's, medische gegevens. Dit niveau vereist niet alleen toegangscontrole maar ook maskering, encryptie en een expliciete rechtsgrond voor verwerking.

Deze classificatie hoort **op kolomniveau** te leven, niet alleen op tabelniveau — een `customers`-tabel bevat vaak een mix: `customer_id` is intern, `email` is restricted, `signup_date` is publiek-achtig. Platformen als Unity Catalog en Purview ondersteunen dit via **policy tags**: metadata die je aan een kolom hangt, waarna toegangsregels automatisch op basis van die tag worden afgedwongen, in plaats van dat elke tabel individueel handmatig beveiligd moet worden.

```
-- Classificatie als kolomtag, niet als los document
ALTER TABLE silver.customers ALTER COLUMN email SET TAGS ('classification' = 'restricted');
ALTER TABLE silver.customers ALTER COLUMN signup_date SET TAGS ('classification' = 'public');

-- Beleid gekoppeld aan de tag, niet aan de specifieke tabel
```

```
CREATE MASKING POLICY mask_restricted AS (val STRING) RETURNS STRING ->
  CASE WHEN CURRENT_ROLE() IN ('data_engineer', 'compliance_officer') THEN val
       ELSE '***MASKED***' END;
ALTER TAG classification SET MASKING POLICY mask_restricted WHEN classification = 'restricted';
```

Dit tag-gebaseerde patroon schaalt fundamenteel beter dan tabel-voor-tabel beleid: een nieuwe kolom met een `email`-achtig karakter krijgt bij aanmaak de juiste tag, en het maskeringsbeleid past zich automatisch toe — niemand hoeft zich te herinneren om een nieuwe tabel handmatig te beveiligen.

## Eigenaarschap en Stewardship: Wie is Verantwoordelijk?

Een tabel zonder eigenaar is een tabel die niemand onderhoudt zodra er iets misgaat. Volwassen governance-programma's onderscheiden drie rollen, vaak verward maar functioneel verschillend.

**Data owner** — meestal een business-rol (een hoofd Sales voor klantdata, een hoofd Finance voor financiële data), verantwoordelijk voor *beslissingen*: wie mag deze data zien, wat betekent een bepaald veld precies, wanneer is data verouderd genoeg om te verwijderen. De owner is zelden de persoon die de tabel technisch bouwt.

**Data steward** — vaak een analist of domeinexpert, verantwoordelijk voor de *dagelijkse kwaliteit*: het bewaken van definities, het signaleren van datakwaliteitsproblemen (zie Data Quality & Testing), het onderhouden van een businessglossarium.

**Data custodian** — meestal de data engineer, verantwoordelijk voor de *technische implementatie*: de tabel daadwerkelijk bouwen, beveiligen, en operationeel houden volgens de beslissingen van de owner en de kwaliteitsnormen van de steward.

Het probleem dat deze scheiding oplost: zonder expliciete rollen valt eigenaarschap impliciet toe aan wie de tabel toevallig heeft gebouwd (de custodian), die zelden de bevoegdheid of businesskennis heeft om beslissingen te nemen over wie toegang moet krijgen of wat een veld precies betekent — met als gevolg dat niemand die beslissingen ooit expliciet neemt.

## Master Data Management: de Ene Waarheid over een Entiteit

Wanneer dezelfde klant in het CRM, het factuursysteem en het supportsysteem onder net iets andere gegevens bestaat, ontstaat een vraag die governance moet beantwoorden: wat is de **golden record** — de ene, gezaghebbende waarheid over die klant? Dit is het domein van **master data management (MDM)**.

MDM lost dit op met een combinatie van matching-logica (welke records in verschillende bronnen representeren dezelfde echte entiteit — vaak via fuzzy matching op naam/adres/e-mail wanneer een gedeelde sleutel ontbreekt) en overervingsregels (welke bron "wint" bij conflicterende waarden — vaak de meest recent bijgewerkte bron, of een vaste prioriteitsvolgorde per veld). Dit raakt direct de Data Vault-benadering uit Data Vault 2.0: een hub met business keys plus satellites per bron is de ruwe basis, en de golden record wordt gebouwd als afgeleide informatielaag erbovenop — MDM is in de praktijk vaak precies het businessvault dat in dat hoofdstuk werd genoemd, met expliciete matching- en overervingsregels als kernlogica.

```
-- Vereenvoudigd golden-record-patroon: nieuwste, meest complete waarde per bron wint
CREATE OR REPLACE TABLE gold.customer_golden AS
SELECT
    customer_key,
    COALESCE(
        MAX(CASE WHEN record_source = 'crm_sales' THEN email END),
        MAX(CASE WHEN record_source = 'crm_acquired' THEN email END)
    ) AS email,
    MAX(load_date) AS last_updated
FROM raw_vault.sat_customer_all_sources
GROUP BY customer_key;
```

## Data Catalogs: Vindbaarheid als Governance-functie

Lineage (behandeld in Databricks) toont *waar data vandaan komt*; een **data catalog** lost een ander probleem op: *welke data bestaat er eigenlijk, en wat betekent het?* Zonder catalog herontdekken analisten steeds opnieuw dat een tabel al bestaat, of erger, bouwen ze een

dubbele versie omdat ze niet wisten dat er al een `gold.revenue_by_category` was. Een catalog combineert technische metadata (schema, laatste update) met businessmetadata (een beschrijving in gewone taal, de eigenaar, gerelateerde termen in een businessglossarium) in één doorzoekbare index — de brug tussen de technische wereld van tabellen en kolommen en de businesswereld van "wat betekent actieve klant eigenlijk".

## Gefedereerd versus Centraal: de Organisatorische Vraag

Een laatste, structurele beslissing: wie is verantwoordelijk voor governance-beleid — één centraal team, of elk domeinteam voor zijn eigen data?

**Centrale governance** — één team stelt beleid vast en handhaaft het over de hele organisatie. Consistent, maar wordt bij grote organisaties al snel een knelpunt: elk nieuw dataset moet langs hetzelfde kleine team, wat vertraging oplevert die precies ingaat tegen de snelheid die een modern dataplatform zou moeten bieden.

**Gefedereerde governance (data mesh)** — elk domeinteam (sales, finance, product) is zelf verantwoordelijk voor de governance van zijn eigen data, binnen een gedeeld, organisatiebreed kader van standaarden (dezelfde classificatieniveaus, dezelfde catalogtool, dezelfde minimale kwaliteitsnormen). Dit is de kerngedachte achter **data mesh**: domeineigenaarschap in plaats van centrale controle, met governance als gedeeld contract in plaats van een centraal goedkeuringsproces.

De praktische vuistregel: centrale governance werkt prima tot een organisatie een omvang bereikt waarop één team simpelweg niet meer alle domeinen kan bijbenen — vergelijkbaar met de monorepo-versus-polyrepo-afweging uit Git & Branch-strategieën, is dit een schaalvraag, geen principiële keuze die je dag één al moet maken.

## Retentiebeleid: Hoe Lang Bewaar je Wat?

Classificatie beantwoordt "wie mag dit zien"; **retentiebeleid** beantwoordt een net zo belangrijke vraag die governance vaak vergeet: "hoe lang mag dit blijven bestaan?" Dit is geen puur technische kwestie (VACUUM-instellingen uit Delta Lake regelen alleen hoelang je *terug kunt kijken*, niet hoelang data *hoort te bestaan*) maar een beleidsbeslissing die per classificatieniveau en vaak per rechtsgrond verschilt.

Restricted-data (persoonsgegevens) valt onder AVG-vereisten die bewaring beperken tot "niet langer dan noodzakelijk voor het doel" — een klant die zijn account drie jaar geleden opzegde, hoort niet voor altijd in je silver-laag te blijven staan zonder concrete reden. Financiële data heeft vaak juist een *minimale* bewaartermijn (wettelijke bewaarplicht, vaak 7 jaar in Nederland), waardoor retentiebeleid soms twee kanten opwerkt: sommige data moet je actief verwijderen, andere data mag je niet te vroeg kwijtraken.

```
-- Retentiebeleid als uitvoerbare regel, niet alleen een document
CREATE OR REPLACE TABLE gold.retention_candidates AS
SELECT customer_id, last_activity_date
FROM silver.customers
WHERE classification = 'restricted'
  AND last_activity_date < CURRENT_DATE - INTERVAL '3 years'
  AND NOT EXISTS (SELECT 1 FROM legal_holds WHERE legal_holds.customer_id = silver.customers.customer_id);
```

De `NOT EXISTS`-check tegen een `legal_holds`-tabel is bewust: een **legal hold** (bijvoorbeeld een lopend juridisch geschil) kan een verwijderverplichting tijdelijk opschorten, ook voor data die anders al verwijderd had moeten zijn — retentiebeleid moet dit soort uitzonderingen expliciet kunnen respecteren, niet blind een vaste regel toepassen.

## Audit Trails: Wie Deed Wat, Wanneer?

Voor gereguleerde sectoren (financieel, zorg) is het niet genoeg om toegang correct te beperken — je moet ook kunnen *aantonen* wie welke data heeft geraadpleegd, gewijzigd of geëxporteerd, vaak met jarenlange bewaarplicht op die logs zelf. Dit is een andere laag dan de lineage uit Databricks (die toont *hoe data getransformeerd is*), en een andere laag dan de teststrategie uit Data Quality & Testing (die toont *of data correct is*) — een audit trail toont specifiek **wie er bij was**.

De meeste moderne platformen loggen dit automatisch op query-niveau (wie voerde welke query uit, tegen welke tabellen, wanneer), maar governance moet expliciet bepalen: welke acties zijn audit-plichtig (elke SELECT, of alleen op restricted-data?), hoe lang blijven audit-logs zelf bewaard, en wie mag de logs zelf inzien — audit-logs zijn zelf vaak weer gevoelige data, omdat ze onthullen wie waarin geïnteresseerd was.

## Best Practices

- **Classificeer op kolomniveau via tags**, niet op tabelniveau via losse documenten — dit is wat automatische, schaalbare beleidsafdwinging mogelijk maakt.
- **Wijs expliciete owners, stewards en custodians toe** per belangrijk domein, met duidelijk onderscheid tussen wie beslist, wie kwaliteit bewaakt, en wie bouwt.
- **Bouw een golden record via expliciete matching- en overervingsregels**, gedocumenteerd en reproduceerbaar — niet via een ad-hoc `COALESCE` die niemand meer kan uitleggen na een jaar.
- **Investeer in een doorzoekbare catalog met businessbeschrijvingen**, niet alleen technische metadata — de meeste "bestaat deze data al"-vragen worden hierdoor voorkomen.
- **Kies gefedereerde governance zodra centrale goedkeuring een merkbaar knelpunt wordt**, niet eerder — vroegtijdige federatie voegt coördinatiekosten toe zonder voordeel.

## Veelgemaakte Fouten

- **Governance gelijkstellen aan RBAC**, waardoor organisatorische vragen (wie is eigenaar, wat betekent dit veld) onbeantwoord blijven ondanks technisch prima beveiligde tabellen.
- **Classificatie op tabelniveau in plaats van kolomniveau**, waardoor een tabel met één gevoelig veld in zijn geheel overbeveiligd wordt, of erger, ondergeclassificeerd blijft omdat de tabel als geheel "niet zo gevoelig" aanvoelde.
- **Geen expliciete owner toewijzen**, waardoor eigenaarschap impliciet bij de bouwer van een tabel terechtkomt, zonder de bevoegdheid om echte governance-beslissingen te nemen.
- **Golden-record-logica ad-hoc en ongedocumenteerd bouwen**, waardoor niemand meer kan reconstrueren waarom bron A voor veld X wint en bron B voor veld Y.
- **Centrale governance aanhouden bij een organisatie die het allang ontgroeid is**, wat elk nieuw dataset onnodig vertraagt en teams motiveert om governance te omzeilen in plaats van te volgen.

## Performance Tips

- **Automatiseer classificatietagging waar mogelijk** (patroonherkenning op kolomnamen als `email`, `ssn`, `phone`) in plaats van elke kolom handmatig te taggen — handmatige tagging wordt bij honderden tabellen onhaalbaar.
- **Cache catalogzoekopdrachten en metadata-aggregaties** — een catalog die bij elke zoekopdracht live alle tabellen bevraagt, schaalt slecht bij duizenden objecten.
- **Beperk MDM-matching tot de velden die het daadwerkelijk nodig hebben** — fuzzy matching over te veel kolommen tegelijk is zowel traag als foutgevoelig.
- **Meet de doorlooptijd van governance-goedkeuringen expliciet** — dit is de concrete metriek die aangeeft wanneer centrale governance een knelpunt begint te worden.

## Interviewvragen

**"Wat is het verschil tussen governance en toegangscontrole (RBAC)?"** Let op: RBAC is een technisch afdwingingsmechanisme; governance omvat de bredere organisatorische beslissingen (eigenaarschap, definities, classificatie) waarvan RBAC er slechts één van de technische uitvoeringen van is.

**"Leg het verschil uit tussen een data owner, een data steward en een data custodian."** Let op: owner beslist (business-rol), steward bewaakt dagelijkse kwaliteit en definities, custodian bouwt en beheert technisch (vaak de data engineer) — drie functioneel verschillende verantwoordelijkheden.

**"Waarom zou je classificatie op kolomniveau doen in plaats van tabelniveau?"** Let op: de meeste tabellen bevatten een mix van gevoeligheidsniveaus; kolomniveau-classificatie via tags maakt geautomatiseerde, schaalbare beleidsafdwinging mogelijk zonder elke tabel individueel te hoeven beveiligen.

**"Wat is een golden record, en hoe bouw je er een?"** Let op: de ene gezaghebbende waarheid over een entiteit die in meerdere bronnen met afwijkende gegevens voorkomt, gebouwd via expliciete matching-logica (welke records dezelfde entiteit zijn) en overervingsregels (welke bron wint per veld).

**"Wanneer zou je kiezen voor gefedereerde in plaats van centrale governance?"** Let op: zodra een centraal governance-team niet meer alle domeinen kan bijbenen en een merkbaar knelpunt wordt — een schaalvraag, geen principiële keuze die vanaf dag één gemaakt moet worden.

**"Wat is het verschil tussen retentiebeleid en de VACUUM-instellingen van Delta Lake?"** Let op: VACUUM bepaalt hoelang je technisch terug kunt kijken (time travel/opslagbeheer); retentiebeleid is een beleidsbeslissing over hoelang data hoort te bestaan, vaak met tegenstrijdige eisen per classificatie (AVG dwingt tot verwijderen, wettelijke bewaarplicht dwingt juist tot bewaren).

**"Wat is een audit trail, en waarom is het iets anders dan lineage?"** Let op: lineage toont hoe data getransformeerd is; een audit trail toont wie welke data heeft geraadpleegd of gewijzigd, wanneer — relevant voor compliance-aantoonbaarheid, niet voor het begrijpen van transformatielogica.

## Relevante Documentatie

- Databricks: Unity Catalog Data Classification — policy tags en geautomatiseerde beleidsafdwinging in de praktijk.
- Microsoft Purview: Data Governance — catalogisering en classificatie binnen het Microsoft-ecosysteem.
- Data Mesh Principles — de kernprincipes achter gefedereerde, domeingedreven governance.

## Samenvatting

Data governance is een organisatorische discipline die verder gaat dan technische toegangscontrole: het omvat classificatie op kolomniveau, expliciet eigenaarschap verdeeld over owners/stewards/custodians, master data management voor een consistente golden record over bronnen heen, en een doorzoekbare catalog die vindbaarheid net zo belangrijk maakt als lineage. De keuze tussen centrale en gefedereerde governance is een schaalvraag: centraal werkt tot het een knelpunt wordt, waarna een data-mesh-achtige, domeingedreven aanpak beter past. Met governance als organisatorisch raamwerk behandeld, verschuift het volgende hoofdstuk naar de technische tegenhanger: Data Security.

HOOFDSTUK 22

# Databeveiliging

## Introductie

Datagovernance behandelde classificatie en kolommaskering al concreet, inclusief een werkende `MASKING POLICY`. Dit hoofdstuk bouwt daarop voort met wat governance bewust openliet: encryptie, rijniveau-beveiliging als aanvulling op kolommaskering, netwerkisolatie, en het onderscheid tussen authenticatie en autorisatie dat de basis vormt onder elk RBAC-systeem dat in dit handbook is langsgekomen.

## Encryptie: At Rest versus In Transit

Twee verschillende beschermingslagen, elk met een ander dreigingsmodel. **Encryptie in transit** (TLS) beschermt data die over het netwerk beweegt — tussen je Python-script en het warehouse, tussen twee microservices — tegen afluisteren onderweg. Vrijwel elk modern platform dwingt dit standaard af; het is zelden iets waar je zelf iets voor hoeft te configureren.

**Encryptie at rest** beschermt data zoals die fysiek op schijf staat, tegen een scenario waarin iemand toegang krijgt tot de onderliggende opslag zonder via het platform zelf te gaan — een gestolen back-up, een verkeerd geconfigureerde cloud-storage-bucket. Hier is een keuze die er echt toe doet: **platform-managed keys** (het platform genereert en beheert de encryptiesleutel zelf, transparant voor jou) versus **customer-managed keys / bring-your-own-key (CMK/BYOK)** (jij beheert de sleutel, vaak in een aparte key vault, met volledige controle over rotatie en intrekking).

```
-- Snowflake: customer-managed key via Tri-Secret Secure
-- (vereenvoudigd — vereist een key vault-integratie op accountniveau)
ALTER ACCOUNT SET ENCRYPTION_KEY_ROTATION = 'CUSTOMER_MANAGED';
```

Het praktische verschil: met platform-managed keys is het platform verantwoordelijk voor beveiliging van de sleutel zelf; met CMK ligt die verantwoordelijkheid — en de mogelijkheid om toegang volledig in te trekken door de sleutel te vernietigen — bij jouw organisatie. Dit is exact de crypto-shredding-techniek uit Medallion Architecture toegepast op accountniveau in plaats van per record: CMK geeft je een "kill switch" die platform-managed encryptie niet biedt, ten koste van de operationele last om sleutelbeheer zelf te doen.

## Row-Level Security: een Andere Dimensie dan Kolommaskering

Datagovernance toonde kolommaskering — welke *kolommen* een gebruiker mag zien. **Row-level security (RLS)** beantwoordt een andere vraag: welke *rijen* mag een gebruiker zien, met dezelfde kolommen zichtbaar voor iedereen die toegang heeft. Het klassieke voorbeeld: een salesmanager mag alle kolommen van de `orders`-tabel zien, maar alleen de rijen van zijn eigen regio.

```
-- Row access policy: elke rol ziet alleen zijn eigen regio
CREATE ROW ACCESS POLICY region_policy AS (region STRING) RETURNS BOOLEAN ->
  CURRENT_ROLE() = 'GLOBAL_ADMIN'
  OR region = (SELECT sales_region FROM user_regions WHERE username = CURRENT_USER());

ALTER TABLE gold.orders ADD ROW ACCESS POLICY region_policy ON (region);
```

Het cruciale verschil met een aparte view per regio (een oudere, minder schaalbare aanpak): met RLS bestaat er **één** tabel en **één** set gold-modellen die erop voortbouwen; het platform past de rijfilter automatisch toe op basis van wie de query uitvoert. Dit voorkomt de wildgroei aan bijna-identieke views die ontstaat wanneer teams RLS proberen te simuleren met losse, per-rol gefilterde views — elk met hun eigen onderhoudslast en risico om uit sync te raken.

## Dynamische Maskering versus Tokenisatie

De maskeringspolicy uit Datagovernance is een vorm van **dynamische maskering**: de onderliggende waarde blijft ongewijzigd opgeslagen, en maskering gebeurt live, op het moment van bevragen, afhankelijk van wie de query uitvoert. Het alternatief,

**tokenisatie**, vervangt de gevoelige waarde al bij het schrijven door een onomkeerbaar of apart-versleuteld token, zodat de werkelijke waarde nergens in de analytische lagen zelf bestaat — alleen een aparte, streng beveiligde tokenizatiedienst kan het token terugvertalen naar de originele waarde, voor de zeldzame gevallen waarin dat nodig is.

Dynamische maskering is flexibeler (dezelfde tabel bedient zowel gemaskeerde als ongemaskeerde gebruikers, afhankelijk van rol) maar vereist dat de ongemaskeerde waarde ergens in het platform blijft bestaan. Tokenisatie is strenger — de waarde bestaat letterlijk nergens meer in leesbare vorm binnen het analytische platform — maar vereist een aparte, apart te onderhouden dienst en maakt het onmogelijk om ooit met de originele waarde te werken zonder terug te gaan naar die dienst. De keuze is vergelijkbaar met de ETL/ELT-compliance-afweging uit ETL vs ELT: hoe strenger de garantie die je nodig hebt, hoe meer architecturale complexiteit je ervoor terug moet accepteren.

## Netwerkisolatie: de Laag Vóór Toegangscontrole

RBAC, RLS en maskering bepalen wat een geauthenticeerde gebruiker mag zien; **netwerkisolatie** bepaalt of een verbinding er überhaupt komt, ongeacht wie erachter zit. Een warehouse dat vanaf elk IP-adres ter wereld bereikbaar is met alleen een wachtwoord, heeft een aanzienlijk groter aanvalsoppervlak dan een warehouse dat alleen bereikbaar is vanuit een specifiek, vertrouwd netwerk.

**IP-allowlisting** beperkt verbindingen tot vooraf goedgekeurde IP-reeksen (kantoornetwerk, VPN-uitgang) — eenvoudig te configureren, maar broos zodra IP-adressen wijzigen. **Private connectivity** (Azure Private Link, AWS PrivateLink) gaat een stap verder: het platform is helemaal niet bereikbaar via het publieke internet, alleen via een privéverbinding binnen je eigen cloudnetwerk — het verschil tussen "een deur met een goed slot" en "geen deur aan de buitenkant, alleen een interne gang".

```
Publiek internet ──X── [geen directe route]
                              |
Jouw VNet/VPC ─── Private Endpoint ─── Warehouse (geen publiek IP)
```

Voor de meeste organisaties is de vuistregel: gebruik private connectivity voor productieomgevingen met gevoelige data, en reserveer publieke toegang (zelfs met IP-allowlisting) voor minder kritieke, ontwikkelomgevingen waar het gemak zwaarder weegt dan de extra beveiligingslaag.

## Authenticatie versus Autorisatie: het Onderscheid Onder Elk RBAC-systeem

Elk toegangscontrolesysteem dat in dit handbook is langsgekomen — Unity Catalog in Databricks, rolgebaseerde toegang in Modern Data Platform Architecture — rust op een onderscheid dat zelden expliciet wordt benoemd. **Authenticatie** beantwoordt "ben je wie je zegt dat je bent" (een wachtwoord, een SSO-login via Azure AD/Okta, een multi-factor-check). **Autorisatie** beantwoordt "mag je, gegeven dat je bent wie je zegt, dit specifieke ding doen" (RBAC, RLS, maskeringspolicies).

Dit onderscheid is niet alleen theoretisch: **single sign-on (SSO)** via een centrale identity provider lost alleen het authenticatie-vraagstuk op — het bepaalt of iemand met een geldig bedrijfsaccount kan inloggen, niet wat die persoon vervolgens mag zien. Een organisatie die SSO invoert zonder de onderliggende autorisatie te herzien, heeft nu simpelweg een makkelijkere manier om iedereen met een bedrijfsaccount bij alles te laten die daarvoor toevallig al brede toegang had — SSO alleen maakt een slecht ingericht autorisatiemodel niet beter, het maakt inloggen alleen makkelijker.

## Service Principals: Nooit een Persoonlijk Account voor een Pipeline

Een veelgemaakte, riskante gewoonte: een pipeline (Airflow-DAG, dbt-CI-run) laten inloggen met de persoonlijke credentials van de engineer die hem heeft opgezet. Dit koppelt de pipeline aan een individu — als die persoon het bedrijf verlaat en zijn account wordt gedeactiveerd, faalt de pipeline zonder duidelijke reden, en de audit trail uit Datagovernance toont "Jan deed dit" voor een actie die eigenlijk een geautomatiseerd systeem uitvoerde.

Een **service principal** (of service account) is een non-human identity, specifiek aangemaakt voor een pipeline of applicatie, met exact de rechten die die specifieke workload nodig heeft — niet meer. Credentials voor een service principal horen in een secrets-store (zie

CI/CD for Data Engineering), nooit hardcoded, en de rechten ervan volgen hetzelfde least-privilege-principe als elke menselijke rol: een ingestion-pipeline die alleen naar bronze schrijft, heeft geen enkele reden om schrijftoegang tot gold te hebben.

## Data Loss Prevention: Ongemarkeerde Gevoelige Data Opsporen

Het classificatiesysteem uit Datagovernance werkt alleen zo goed als de discipline waarmee kolommen daadwerkelijk getagd worden — en in de praktijk ontsnapt er altijd wel eens iets: een engineer voegt een `notes`-veld toe aan een supporttabel, en zes maanden later blijkt daar sporadisch een klant-e-mailadres in te zijn geplakt, in een kolom die nooit als restricted was gemarkeerd. **Data Loss Prevention (DLP)**-tooling scant automatisch, periodiek, de inhoud van kolommen op patronen die op gevoelige data wijzen — e-mailformaten, creditcardnummers, BSN-patronen — ongeacht of die kolom vooraf als zodanig was getagd.

```
# Vereenvoudigd DLP-scanpatroon: periodieke content-scan, niet alleen kolomnaam-heuristiek
import re

EMAIL_PATTERN = re.compile(r"[\w.+-]+@[\w-]+\.[\w.-]+")

def scan_column_for_pii(rows: list[str]) -> float:
    """Geeft het percentage rijen terug dat een e-mailachtig patroon bevat."""
    matches = sum(1 for r in rows if r and EMAIL_PATTERN.search(r))
    return matches / len(rows) if rows else 0.0

# Een 'notes'-kolom die nooit als restricted was getagd, maar wel 12% e-mailachtige waarden bevat
if scan_column_for_pii(sample_rows) > 0.05:
    flag_for_review("support.notes", reason="mogelijk ongemarkeerd PII-patroon gedetecteerd")
```

Het verschil met classificatie uit het vorige hoofdstuk: classificatie is **proactief** (bepaal vooraf wat gevoelig is en tag het), DLP is **reactief/detecterend** (vind wat er per ongeluk toch gevoelig in bleek te zitten, ondanks classificatie). Beide zijn nodig — classificatie dekt het merendeel van de gevallen efficiënt, DLP vangt de onvermijdelijke uitzonderingen die menselijke discipline alleen niet had gevonden.

## Threat Detection: Afwijkende Toegangspatronen

Naast de audit trail uit Datagovernance (die *registreert* wie wat deed) is er een actievere laag: **threat detection**, die afwijkend gedrag in bijna-realtime signaleert. Een analist die normaliter enkele honderden rijen per dag bevraagt en plotseling een volledige tabel met miljoenen rijen exporteert, een login vanuit een ongebruikelijk land, of toegang buiten normale werktijden vanaf een nooit eerder geziene locatie — dit zijn patronen die een audit log wel *vastlegt*, maar niet uit zichzelf *signaleert*.

De meeste cloudplatformen bieden hiervoor ingebouwde of aan te sluiten detectiediensten (Azure Sentinel, AWS GuardDuty, Snowflake's Trust Center) die statistische baselines per gebruiker opbouwen en afwijkingen daarvan markeren — hetzelfde anomaliedetectie-principe als bij datakwaliteit in Data Quality & Testing, hier toegepast op *toegangsgedrag* in plaats van op *datawaarden*. Dit is precies het raakvlak met het volgende hoofdstuk: dezelfde monitoring-infrastructuur die operationele problemen signaleert, is vaak ook waar beveiligingsafwijkingen het eerst zichtbaar worden.

## Best Practices

- **Gebruik customer-managed keys voor restricted-classificatiedata** waar het platform dat ondersteunt, zodat je organisatie zelf de "kill switch" op encryptie behoudt.
- **Kies RLS boven losse, per-rol views** zodra meerdere rollen dezelfde kolommen maar verschillende rijen moeten zien — één tabel, één policy, geen wildgroei aan bijna-identieke views.
- **Gebruik private connectivity voor productieomgevingen met gevoelige data**, en reserveer publieke toegang met IP-allowlisting voor minder kritieke omgevingen.
- **Geef elke pipeline zijn eigen service principal** met minimale, specifiek benodigde rechten — nooit een persoonlijk account, nooit gedeelde "god mode"-credentials.
- **Behandel SSO en autorisatie als twee aparte projecten** — SSO invoeren lost geen autorisatieprobleem op dat er al was.

## Veelgemaakte Fouten

- **Platform-managed keys gebruiken voor alle data zonder onderscheid**, ook voor de meest gevoelige classificatieniveaus waar een organisatie eigenlijk zelf controle over de sleutel zou willen hebben.
- **RLS simuleren met tientallen losse, per-regio/per-rol views** in plaats van één tabel met een row access policy, wat onderhoudslast en het risico op onderling inconsistente views vergroot.
- **Warehouses publiek bereikbaar laten met alleen IP-allowlisting** voor productieomgevingen met restricted-data, waar private connectivity het aanvalsoppervlak fundamenteel kleiner had gemaakt.
- **Persoonlijke credentials gebruiken voor pipelines**, waardoor een pipeline breekt zodra een medewerker vertrekt en audit trails de verkeerde verantwoordelijke tonen.
- **SSO invoeren en autorisatie als "vanzelf opgelost" beschouwen**, terwijl het onderliggende, mogelijk te brede autorisatiemodel ongewijzigd blijft.

## Performance Tips

- **Houd row access policies eenvoudig en indexeerbaar** (een directe kolomvergelijking, geen zware subquery per rij) — een complexe policy kan elke query op de beveiligde tabel merkbaar vertragen.
- **Cache gebruiker-naar-regio-mappings** (zoals de `user_regions`-tabel hierboven) apart en klein, zodat de policy-lookup zelf niet de bottleneck wordt.
- **Beperk het aantal actieve maskeringspolicies per kolom** — gestapelde, elkaar overlappende policies zijn lastig te doorgronden en te debuggen bij onverwacht gedrag.
- **Monitor de latency-impact van private connectivity** bij het opzetten — een verkeerd gerouteerde private endpoint kan onverwacht extra netwerklatency toevoegen ten opzichte van directe publieke toegang.

## Interviewvragen

**"Wat is het verschil tussen encryptie at rest en in transit?"** Let op: in transit beschermt data die over het netwerk beweegt (TLS); at rest beschermt data zoals die fysiek is opgeslagen — met platform-managed versus customer-managed keys als aanvullende keuze voor at-rest-encryptie.

**"Wat is row-level security, en hoe verschilt het van kolommaskering?"** Let op: RLS beperkt welke rijen een gebruiker ziet (dezelfde kolommen voor iedereen); kolommaskering beperkt welke kolomwaarden zichtbaar zijn (dezelfde rijen voor iedereen) — twee aanvullende, niet-overlappende mechanismen.

**"Wat is het verschil tussen dynamische maskering en tokenisatie?"** Let op: dynamische maskering verbergt de waarde live bij het bevragen terwijl de originele waarde blijft bestaan; tokenisatie vervangt de waarde al bij het schrijven, zodat de originele waarde nergens in de analytische laag zelf bestaat.

**"Leg het verschil uit tussen authenticatie en autorisatie."** Let op: authenticatie bevestigt identiteit (ben je wie je zegt); autorisatie bepaalt wat die geverifieerde identiteit mag doen — SSO lost alleen het eerste op, niet het tweede.

**"Waarom zou een pipeline nooit met een persoonlijk account moeten inloggen?"** Let op: koppelt de pipeline aan een individu (faalt als die persoon vertrekt), vertroebelt audit trails, en schendt least privilege — een service principal met minimale, specifieke rechten is de juiste aanpak.

**"Wat is het verschil tussen classificatie en Data Loss Prevention?"** Let op: classificatie is proactief (vooraf taggen wat gevoelig is); DLP is reactief/detecterend (periodiek scannen op content-patronen die op gevoelige data wijzen, ook in ongetagde kolommen) — beide nodig, ze dekken elkaars blinde vlekken.

**"Hoe zou je afwijkend toegangsgedrag detecteren dat een audit log wel vastlegt maar niet uit zichzelf signaleert?"** Let op: statistische baselines per gebruiker opbouwen (normaal volume, normale tijden/locaties) en afwijkingen daarvan actief markeren —

hetzelfde anomaliedetectie-principe als bij datakwaliteit, hier toegepast op toegangsgedrag.

## Relevante Documentatie

- Snowflake: Row Access Policies — RLS-configuratie en -voorbeelden in detail.
- Snowflake: Tri-Secret Secure (Customer-Managed Keys) — CMK-configuratie op accountniveau.
- Azure: Private Link Overview — private connectivity-architectuur.
- Databricks: Service Principals — non-human identities voor pipelines en automatisering.

## Samenvatting

Databeveiliging bouwt voort op governance's classificatie en kolommaskering met vier aanvullende lagen: encryptie (met customer-managed keys als "kill switch" voor de meest gevoelige data), row-level security als aanvulling op kolommaskering voor wanneer verschillende rollen dezelfde kolommen maar andere rijen moeten zien, netwerkisolatie die bepaalt of een verbinding er überhaupt komt, en het onderscheid tussen authenticatie (wie ben je) en autorisatie (wat mag je) dat onder elk RBAC-systeem in dit handbook ligt. Service principals zorgen dat pipelines nooit afhankelijk zijn van een individueel account. Met security als technische tegenhanger van governance behandeld, verschuift het volgende hoofdstuk naar hoe je dit alles — en de rest van het platform — operationeel bewaakt: Monitoring & Observability.

HOOFDSTUK 23

# Modern Dataplatform Architectuur

## Introductie

"Modern dataplatform" is een van die termen die overal wordt gebruikt en zelden wordt gedefinieerd. In dit hoofdstuk bedoelen we er iets specifieks mee: een architectuur waarin storage en compute onafhankelijk van elkaar schalen, transformatie gebeurt via versiebeheerde, testbare code in plaats van point-and-click ETL-tools, en de platformkeuze wordt gedreven door concrete workload-eisen in plaats van door wat net op een conferentie werd gepitcht.

De vorige hoofdstukken behandelden losse bouwstenen — de stacklagen in Introduction to Data Engineering, de SQL-patronen in SQL for Data Engineers. Dit hoofdstuk zet een stap terug en behandelt de architectuurbeslissingen die bepalen hoe die bouwstenen samen een platform vormen dat na twee jaar nog steeds prettig is om op te werken — in plaats van een systeem waar niemand meer aan durft te komen.

Om dit concreet te houden, volgen we doorheen het hoofdstuk één doorlopend voorbeeld: een fictief maar realistisch SaaS-bedrijf, **"Flowmetric"**, dat facturatiesoftware verkoopt aan MKB-klanten. We volgen hoe hun dataplatform evolueert van een eerste dashboard voor de founder tot een platform dat honderden interne gebruikers en een extern klantenportaal bedient. De architectuurbeslissingen die daarbij spelen, zijn geen hypothetische theorie — het zijn dezelfde beslissingen die ik bij vrijwel elke opdracht opnieuw tegenkom.

## De Drie Lagen Die Elk Platform Nodig Heeft

Ongeacht welke specifieke tools je kiest, elk volwassen dataplatform heeft drie architectonisch gescheiden lagen nodig, en de meeste problemen die ik in bestaande platformen tegenkom, zijn terug te voeren op het vermengen van deze lagen.

**Ingestion, losgekoppeld van transformatie.** De code die data binnenhaalt uit bronsystemen mag geen businesslogica bevatten. Zijn enige taak is data zo betrouwbaar mogelijk, in een zo ruw mogelijke vorm, naar storage krijgen. Zodra ingestion-code begint te filteren, aggregeren of "opschonen", verlies je de mogelijkheid om transformatielogica later te herzien zonder opnieuw te hoeven extraheren — en dat is precies de situatie waarin platformen vastlopen.

**Storage en compute, onafhankelijk schaalbaar.** Dit is de architectonische doorbraak die Snowflake, Databricks en Fabric allemaal delen (in tegenstelling tot traditionele on-premises warehouses): je betaalt voor opslag naar wat je daadwerkelijk bewaart, en voor compute naar wat je daadwerkelijk verwerkt, en die twee kosten schalen onafhankelijk van elkaar. Praktisch betekent dit dat je een goedkoop, altijd-aan opslaglaag kunt hebben met daarnaast meerdere, los van elkaar schaalbare compute-clusters voor verschillende workloads — een zwaar nachtelijk backfill-proces hoeft de compute voor interactieve BI-queries overdag niet te beïnvloeden.

**Transformatie, als code.** Business logica hoort in versiebeheerde, testbare, reviewbare code — SQL-bestanden in een dbt-project, of Python/Scala in Spark-notebooks die via CI/CD worden gedeployed. Niet in een GUI-gebaseerde ETL-tool waar wijzigingen niet te diffen zijn en waar "wie heeft dit veranderd en waarom" een archeologische opgave wordt. Dit is het onderwerp van dbt en CI/CD for Data Engineering.

## Case Study: Flowmetric van Startup naar Schaal

**Fase 1 — één founder, één spreadsheet, dan één warehouse (0-10 medewerkers).** Flowmetric begint zoals bijna elk bedrijf: de founder wil weten hoeveel omzet er binnenkomt. In deze fase is de "architectuur" één Snowflake-warehouse (of een gratis BigQuery-sandbox), data komt handmatig of via een enkel Fivetran-connectortje binnen uit Stripe, en er is precies één dbt-project met een man of twee, drie modellen. Dit is **prima**. De fout die ik hier het vaakst zie is niet te weinig architectuur, maar te veel: een team van twee mensen dat een Kafka-cluster opzet voor 500 events per dag omdat "we willen schaalbaar zijn." Bouw in fase 1 het simpelste dat werkt, met de medallion-basisstructuur (bronze/silver/gold) er al wél in, omdat die structuur later vrijwel niets kost om aan te houden en achteraf invoeren pijnlijk is.

**Fase 2 — meerdere teams, meerdere bronsystemen (10-100 medewerkers).** Flowmetric heeft nu een salesteam met een eigen CRM, een supportteam met een ticketsysteem, en een productteam dat event-tracking wil. Dit is het punt waarop ingestion een systeem wordt in plaats van een los scriptje: Fivetran of Airbyte voor de SaaS-bronnen, een eigen CDC-pipeline voor de productiedatabase omdat een dagelijkse volledige export te traag wordt. De silver-laag krijgt voor het eerst echte datakwaliteitsregels, omdat er nu mensen zijn die de cijfers gebruiken die de engineer die de pipeline bouwde nooit heeft gesproken. Dit is ook het moment waarop **data contracts** (verderop in dit hoofdstuk) van "aardig om te hebben" naar "noodzakelijk" gaan, omdat het productteam een kolom hernoemt zonder dat iemand in het dataeam het weet — en het omzetdashboard van de CFO een week lang stil staat op nul.

**Fase 3 — platform als product, honderden gebruikers (100+ medewerkers).** Flowmetric heeft nu een intern datateam van vijf mensen, een self-service BI-cultuur met tientallen analisten die zelf dbt-modellen bouwen, en een extern klantenportaal dat live metrics toont — wat betekent dat het dataplatform niet langer alleen intern is, maar onderdeel van het product zelf, met bijbehorende eisen aan uptime en latency. Hier verschijnen governance en toegangsbeheer als eerste-klas architectuurbeslissing (zie verderop), een formeel orchestratieplatform vervangt losse cron-jobs, en de vraag "welk warehouse-platform gebruiken we" krijgt een heel ander gewicht dan in fase 1 — een migratie op dit niveau kost al snel maanden.

Het punt van deze case study is niet dat elk bedrijf deze exacte fasen doorloopt, maar dat **architectuur die in fase 1 correct aanvoelt (simpel, weinig lagen, één platform) in fase 3 problematisch wordt, en andersom** — een fase 3-architectuur toepassen in fase 1 is de meest voorkomende vorm van over-engineering die ik zie bij startups en early-stage teams.

## Medallion Architecture als de Facto Standaard

Binnen de transformatielaag heeft één patroon zich de afgelopen jaren ontwikkeld tot de facto standaard: **medallion architecture**, met bronze-, silver- en gold-lagen. Dit is niet zomaar een naamgevingsconventie — het codificeert een specifieke verantwoordelijkheidsverdeling die rechtstreeks voortkomt uit de scheiding tussen ingestion en transformatie hierboven.

**Bronze** bevat ruwe, ongewijzigde data, precies zoals die uit de bron kwam — inclusief duplicaten, foute types en ontbrekende waarden. Voor Flowmetric in fase 2 betekent dit: de ruwe Stripe-facturen, de ruwe CRM-deals, de ruwe supporttickets, elk in hun eigen bronze-tabel, ongefilterd. Dit is je vangnet: als transformatielogica een bug bevat die pas maanden later wordt ontdekt, kun je silver en gold herbouwen vanuit bronze zonder opnieuw te hoeven extraheren uit een bronsysteem dat inmiddels misschien niet meer dezelfde data teruggeeft — relevant, want Stripe bewaart lang niet alle historische API-responses in dezelfde vorm.

**Silver** bevat schone, gededupliceerde, correct getypeerde data — één betrouwbare rij per entiteit, maar nog niet noodzakelijk geaggregeerd of verrijkt met businesslogica. Voor Flowmetric: één rij per factuur, gededupliceerd, met een consistent valutaformaat ongeacht of de brondata in centen of euro's stond. Dit is de laag waar de meeste datakwaliteitsvalidatie plaatsvindt.

**Gold** bevat businessklare, vaak geaggregeerde tabellen, direct bruikbaar door BI-tools of downstream-applicaties — precies gemodelleerd naar de vraag die ze moeten beantwoorden, niet naar de vorm van de bron. Voor Flowmetric: `gold.mrr_per_maand`, `gold.churn_per_segment` — tabellen die letterlijk de vraag beantwoorden die iemand in een dashboard stelt, zonder dat die persoon een join hoeft te schrijven.

De kracht van dit patroon zit in de foutisolatie: elke laag is onafhankelijk herbouwbaar, en een fout in één laag corrumpeert nooit stilletjes de lagen ervoor. Zie Medallion Architecture voor een volledige uitwerking, inclusief hoe dit patroon zich verhoudt tot Data Vault als alternatieve modelleeraanpak in Data Vault 2.0.

## Platform Kiezen: Snowflake versus Databricks versus Fabric

Dit is de vraag die ik het vaakst krijg, en het eerlijke antwoord is dat de drie grote platformen dichter bij elkaar zijn komen te liggen dan marketingmateriaal doet vermoeden — Snowflake heeft met Snowpark en Dynamic Tables Spark-achtige en declaratieve mogelijkheden toegevoegd, Databricks heeft met Databricks SQL en Unity Catalog een warehouse-achtige ervaring bovenop het lakehouse gebouwd, en Microsoft Fabric probeert beide werelden vanaf het begin te combineren. Toch zijn er praktische verschillen die de keuze bepalen.

**Snowflake** blinkt uit wanneer je primaire workload SQL-gebaseerde analytics is met veel gelijktijdige gebruikers. De scheiding tussen storage en meerdere onafhankelijk schaalbare virtual warehouses is uitzonderlijk volwassen, en het beheer is eenvoudiger dan bij Spark-gebaseerde platformen omdat je zelden onder de motorkap hoeft te kijken. De keerzijde: voor zware, custom Python/ML-workloads voelt het minder natuurlijk dan Databricks, en de kosten kunnen oplopen als warehouses niet actief worden beheerd op auto-suspend-instellingen.

**Databricks** is de sterkere keuze wanneer machine learning, complexe Python/Scala-transformaties, of streaming-workloads een substantieel deel van je platform vormen. Het notebook-gedreven ontwikkelmodel en de directe toegang tot Spark maken het flexibeler voor niet-SQL-workloads, en Delta Lake (behandeld in Delta Lake) geeft ACID-garanties op een lakehouse die vroeger alleen warehouses boden. De keerzijde: het beheren van clustergrootte en -configuratie vraagt meer operationele kennis dan Snowflake's warehouse-model.

**Microsoft Fabric** is het meest overtuigend voor organisaties die al diep in het Microsoft-ecosysteem zitten — Power BI, Azure Active Directory, Office 365 — en die profiteren van een unified platform waar ingestion (Data Factory), transformatie (notebooks, dataflows) en rapportage (Power BI) native op elkaar aansluiten zonder aparte licenties en connectoren te hoeven beheren. De keerzijde: als losstaand analytics-platform, buiten het Microsoft-ecosysteem, is het minder volwassen dan de andere twee.

| Dimensie | Snowflake | Databricks | Microsoft Fabric |
|---|---|---|---|
| Sterkste workload | SQL-analytics, veel gelijktijdige gebruikers | ML/Python, complexe Spark-transformaties | Microsoft-ecosysteem-integratie |
| Compute-model | Virtual warehouses, per-seconde billing | Clusters (job- of all-purpose), per-DBU billing | Capacity units, gedeeld over workloads |
| Declaratieve laag | Dynamic Tables (`TARGET_LAG`) | Delta Live Tables | Dataflows Gen2 |
| Beheerlast | Laag — weinig infrastructuurkeuzes | Gemiddeld — clustergrootte/-configuratie | Laag — grotendeels managed |
| Sterkste integratie | Data-marktplaats, third-party BI-tools | MLflow, notebooks, Unity Catalog | Power BI, Azure AD, Office 365 |

### Een Concreet Kostenvoorbeeld

Featurevergelijkingen zijn abstract; kosten zijn dat niet. Stel Flowmetric in fase 2 heeft een nachtelijke batchjob die 45 minuten draait op een medium-compute-node, plus continu een klein aantal interactieve BI-queries overdag. Ruwweg vertaalt zich dat naar twee zeer verschillende kostenprofielen, afhankelijk van hoe bewust je het inricht:

- **Zonder workload-scheiding** (één warehouse/cluster voor alles, altijd op medium-formaat): je betaalt het medium-tarief 24/7, ook tijdens de 22 uur per dag dat er niemand een query draait — dit is verreweg de meest voorkomende manier waarop teams onnodig geld verbranden.
- **Met workload-scheiding en auto-suspend** (een XS-warehouse voor interactieve BI-queries dat na 60 seconden inactiviteit stopt, en een apart medium-warehouse dat alleen 45 minuten per nacht draait voor de batchjob): je betaalt alleen voor de daadwerkelijke 45 minuten batch-compute plus de paar cumulatieve minuten dat er echt een BI-query loopt — in de praktijk vaak een besparing van 70-90% ten opzichte van het eerste scenario, zonder dat de gebruikerservaring merkbaar verandert.

Dit exacte patroon — auto-suspend, workload-scheiding, rechtsizing — is platformonafhankelijk: Snowflake noemt het warehouse-sizing, Databricks noemt het cluster-autoscaling en job-vs-all-purpose-clusters, Fabric regelt het via capacity-toewijzing. De architectuurles is hetzelfde ongeacht het platform: **compute die niet actief gebruikt wordt, hoort niet te draaien**, en dat is een instelling, geen toeval.

De praktische vuistregel voor platformkeuze: kies niet op basis van welk platform het meest indrukwekkend klinkt op een conferentie, maar op basis van waar je team al ervaring heeft, welk ecosysteem je organisatie al gebruikt, en of je workload primair SQL-analytics, ML/Python, of een mix is. Een verkeerde platformkeuze is duur om terug te draaien; een middelmatige implementatie op het juiste platform is altijd beter dan een perfecte implementatie op het verkeerde platform.

## Governance en Security als Architectuurbeslissing

Een fout die ik regelmatig zie: governance en security worden behandeld als iets dat je "later nog even toevoegt" zodra het platform draait. Dat werkt niet — wie toegang heeft tot welke laag, hoe gevoelige kolommen (BSN's, e-mailadressen, betaalgegevens) worden gemaskeerd, en hoe lineage wordt vastgelegd, zijn beslissingen die de architectuur zelf raken, niet een laagje dat je er achteraf overheen legt. Bij Flowmetric wordt dit concreet zodra het externe klantenportaal live metrics toont: op dat moment is een verkeerd ingestelde rol niet langer een intern ongemak, maar een datalek naar een klant van een andere klant.

Concreet betekent dit: rolgebaseerde toegang instellen per laag (bronze doorgaans alleen toegankelijk voor het data-engineeringteam, gold breder beschikbaar voor analisten), kolomniveau-maskering voor PII direct in de silver-laag definiëren zodat gevoelige data nooit onbeschermd in gold terechtkomt, en lineage-tools (Unity Catalog bij Databricks, Purview bij Microsoft Fabric/Azure) vanaf dag één aankoppelen in plaats van pas wanneer een auditor ernaar vraagt. Dit wordt in detail behandeld in Data Governance en Data Security.

## Data Contracts: de Grens Tussen Teams Vastleggen

Terug naar het moment in fase 2 waarop het productteam bij Flowmetric een kolom hernoemt en het omzetdashboard breekt. Dit is geen ongeluk dat je met meer alertheid voorkomt — het is een architectuurprobleem: er was geen expliciete, afdwingbare afspraak tussen het team dat de brondata produceert en het team dat die consumeert. Een **data contract** is precies die afspraak, vastgelegd als code in plaats van als document dat niemand meer leest:

```
# contracts/orders.yml — afspraak tussen het productteam (bron) en het datateam (consument)
contract:
  table: production.orders
  owner: product-team
  consumers: [data-team, finance-team]
  schema:
    - name: order_id
      type: string
      nullable: false
    - name: customer_id
      type: string
      nullable: false
    - name: amount_cents
      type: integer
      nullable: false
      description: "Bedrag in centen, NOOIT euro's — zie ADR-014"
    - name: status
      type: string
      allowed_values: [pending, paid, refunded, cancelled]
  breaking_change_policy: "Kolommen verwijderen of hernoemen vereist 30 dagen aankondiging in #data-contracts"
```

Dit bestand is geen documentatie in de zin van "leuk om te hebben" — het is idealiter gekoppeld aan een CI-check die faalt als het productteam een pull request opent die het schema van `production.orders` breekt zonder dit bestand bij te werken, vergelijkbaar met hoe dbt-tests (behandeld in Data Quality & Testing) silver- en gold-modellen bewaken. Het verschil is de richting: dbt-tests bewaken wat *jouw team* bouwt; data contracts bewaken wat *een ander team* aan jou levert — en dat is precies de grens waar de meeste platform-brede incidenten ontstaan.

## Orchestratie & de Opkomst van Declaratieve Pipelines

Een merkbare verschuiving in platformarchitectuur de afgelopen jaren is de beweging van imperatieve orchestratie ("draai stap A, dan stap B, dan stap C, in deze volgorde, op dit tijdstip") naar declaratieve pipelines ("dit is de gewenste eindtoestand van deze tabel; het platform bepaalt zelf hoe en wanneer die actueel gehouden wordt").

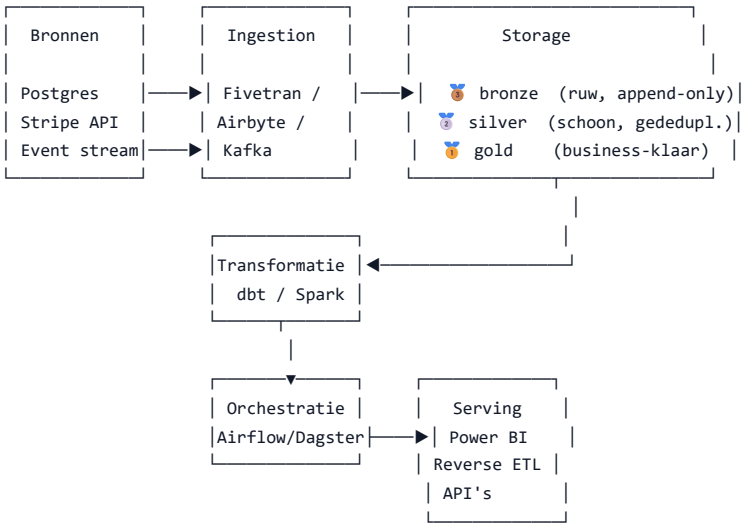
dbt was hier vroeg bij: je definieert modellen als SQL-`SELECT`-statements en dbt leidt de afhankelijkheidsgrafiek automatisch af uit `ref()`-aanroepen, in plaats van dat jij handmatig een DAG met volgordes onderhoudt. Snowflake's Dynamic Tables en Databricks' Delta Live Tables (DLT) trekken dit patroon nog verder door: je declareert een query en een verversingsdoel (`TARGET_LAG` bij Snowflake), en het platform bepaalt zelf de uitvoeringsfrequentie en -volgorde over een hele keten van afhankelijke tabellen. Het verschil tussen deze

declaratieve objectbeheer-aanpak en dbt's rol als transformatietool wordt in detail behandeld in twee gerelateerde artikelen op onze blog: dbt vs Snowflake DCM en Snowflake DCM vs Databricks Asset Bundles.

Deze verschuiving vermindert niet de noodzaak van orchestratietools als Airflow of Dagster — die blijven essentieel voor het coördineren van *cross-platform* workflows (bijvoorbeeld: wacht tot een externe API-extractie klaar is, trigger dan een dbt-run, en start pas daarna een ML-trainingsjob) — maar het verplaatst een groot deel van de *binnen-platform* verversingslogica van handmatig geplande DAG's naar declaratieve, door het platform beheerde afhankelijkheidsgrafieken. Voor Flowmetric in fase 3, met tientallen onderling afhankelijke modellen, is dit het verschil tussen een Airflow-DAG die niemand meer durft aan te passen en een dependency graph die dbt of DLT automatisch afleidt en visualiseert.

## Een Referentiearchitectuur

Een representatieve architectuur voor een platform op het niveau van Flowmetric in fase 2-3, samengevoegd uit de patronen hierboven:




De pijl van transformatie terug naar storage (bronze → silver → gold, allemaal binnen dezelfde storage-laag) is bewust: transformatie *verplaatst* data niet naar een apart systeem, het *herschrijft* data binnen dezelfde storage-laag naar een verder verfijnde vorm — precies het punt van de storage/compute-scheiding die dit hele model economisch maakt. Merk op dat het externe klantenportaal uit fase 3 hier niet rechtstreeks op gold inhaakt, maar via een aparte "Serving"-laag (API's, reverse ETL) — een extern-gerichte applicatie mag nooit direct tegen je warehouse bevragen, zowel om performance-isolatie als om governance-redenen.

## Codevoorbeelden

**Een dbt-model** dat silver naar gold transformeert, met expliciete afhankelijkheid via `ref()`:

```
-- models/gold/revenue_by_category.sql
{{ config(materialized='table') }}

WITH orders AS (
    SELECT * FROM {{ ref('silver_orders') }}
),
products AS (
    SELECT * FROM {{ ref('silver_products') }}
)

SELECT
    p.category,
    DATE_TRUNC('day', o.order_date) AS order_day,
    SUM(o.amount) AS revenue,
    COUNT(DISTINCT o.order_id) AS order_count
FROM orders o
JOIN products p ON p.product_id = o.product_id
GROUP BY p.category, DATE_TRUNC('day', o.order_date)
```

**Een Databricks-notebookfragment** dat dezelfde gold-laag bouwt met PySpark, voor een platform waar de transformatielaag Spark is in plaats van dbt-SQL:

```
# gold_revenue_by_category.py
from pyspark.sql import functions as F

orders = spark.read.table("silver.orders")
products = spark.read.table("silver.products")

gold_revenue = (
    orders
    .join(products, "product_id")
    .groupBy("category", F.date_trunc("day", "order_date").alias("order_day"))
    .agg(
        F.sum("amount").alias("revenue"),
        F.countDistinct("order_id").alias("order_count"),
    )
)

gold_revenue.write.format("delta").mode("overwrite").saveAsTable("gold.revenue_by_category")
```

**Een Fabric-pipelineschets** die laat zien hoe dezelfde bronze→silver→gold-keten wordt uitgedrukt als pipeline-activiteiten in Microsoft Fabric:

```
{
  "name": "revenue_pipeline",
  "activities": [
    { "name": "Copy_Orders_to_Bronze", "type": "Copy", "dependsOn": [] },
    { "name": "Notebook_Bronze_to_Silver", "type": "Notebook", "dependsOn": ["Copy_Orders_to_Bronze"] },
    { "name": "Notebook_Silver_to_Gold", "type": "Notebook", "dependsOn": ["Notebook_Bronze_to_Silver"] }
  ]
}
```

Deze drie voorbeelden zijn bewust functioneel identiek — hetzelfde referentiepatroon, drie verschillende platformen. Dat is precies het punt: de architectuur staat los van het gekozen platform, alleen de syntax verandert.

**Warehouse-sizing als infrastructuur-als-code**, zodat het kostenvoorbeeld van hierboven (workload-scheiding, auto-suspend) reproduceerbaar en reviewbaar is in plaats van een eenmalige handmatige klik in een dashboard:

```
# warehouses.tf — Terraform, workload-scheiding zoals hierboven beschreven
resource "snowflake_warehouse" "bi_interactive" {
  name           = "BI_INTERACTIVE_WH"
  warehouse_size = "X-Small"
  auto_suspend   = 60      # seconden — stop bijna direct na de laatste query
  auto_resume    = true
}

resource "snowflake_warehouse" "nightly_batch" {
  name           = "NIGHTLY_BATCH_WH"
  warehouse_size = "Medium"
  auto_suspend   = 300
  auto_resume    = true
}
```

## Best Practices

- **Ontkoppel ingestion volledig van transformatie.** Ingestion-code mag geen businesslogica bevatten; zijn enige taak is data zo ruw mogelijk laten landen.
- **Behandel elke laag als onafhankelijk herbouwbaar.** Als je silver of gold niet vanaf nul kunt herbouwen vanuit bronze, heb je geen medallion-architectuur — je hebt alleen een naamgevingsconventie.
- **Kies je platform op basis van workload, niet van hype.** SQL-zware analytics wijst naar Snowflake, Python/ML-zware workloads wijzen naar Databricks, diepe Microsoft-integratie wijst naar Fabric.
- **Leg transformatielogica vast als code**, gereviewd en getest via CI/CD — nooit in een GUI-tool waar wijzigingen niet te diffen zijn.

- **Documenteer de afhankelijkheidsgrafiek expliciet**, of laat een tool als dbt die automatisch afleiden uit `ref()`-aanroepen, zodat niemand een tabel per ongeluk breekt door een upstream-wijziging.
- **Formaliseer data contracts tussen teams**, niet alleen tests binnen je eigen team — de meeste platform-brede incidenten ontstaan op de grens tussen teams, niet binnen één team.
- **Match de architectuur aan de huidige fase van je organisatie**, niet aan waar je over drie jaar hoopt te zijn — zie de Flowmetric-case study hierboven.
- **Laat externe/klantgerichte applicaties nooit rechtstreeks tegen het warehouse bevragen** — altijd via een aparte serving-laag (API, reverse ETL) voor performance-isolatie en governance.

## Veelgemaakte Fouten

- **Over-engineering vanaf dag één.** Een volledige medallion-architectuur met vijf lagen en real-time streaming bouwen voor een dataset van tien miljoen rijen die één keer per dag wordt bijgewerkt. Begin eenvoudig; voeg complexiteit toe wanneer een concrete eis dat rechtvaardigt, niet vooraf.
- **Platformkeuze op basis van hype in plaats van fit.** Overstappen naar een nieuw platform omdat een concurrent het gebruikt, zonder de workload-eisen te analyseren die daadwerkelijk bepalen welk platform past.
- **Kostengovernance negeren.** Geen budgetalerts, geen auto-suspend op warehouses, geen review van clustergroottes — waardoor compute-kosten ongemerkt groeien tot iemand de rekening ziet, precies het scenario uit het kostenvoorbeeld hierboven.
- **Geen data contracts tussen teams.** Downstream-consumenten die stilletjes breken omdat een bronteam een tabel wijzigt zonder enige vorm van afspraak of versiebeheer op het schema.
- **Eén enorme monolithische transformatiejob.** In plaats van kleine, onafhankelijk testbare modellen (het dbt-patroon), één gigantische SQL- of Spark-job die alles in één keer doet — onmogelijk te debuggen als er iets misgaat, en alles moet opnieuw draaien voor elke kleine wijziging.
- **Externe applicaties direct op het warehouse aansluiten.** Een klantenportaal dat rechtstreeks tegen `gold`-tabellen queryt lijkt in fase 1 een snelle oplossing, maar koppelt de latency en beschikbaarheid van je analytics-platform aan die van je product — één zware interne backfill kan dan een klant-facing feature vertragen.
- **Eén architectuur voor alle fases proberen te bouwen.** Proberen in fase 1 al de governance- en schaal-eisen van fase 3 te implementeren, wat vooral vertraging oplevert zonder dat er nog gebruikers zijn die er baat bij hebben.

## Performance Tips

- **Scheid compute per workload.** Een apart, klein warehouse of cluster voor interactieve BI-queries en een apart, groter warehouse voor nachtelijke batchjobs voorkomt dat de twee elkaar in de weg zitten en maakt kosten per workload zichtbaar — zie het Terraform-voorbeeld hierboven.
- **Gebruik incrementele modellen agressief in de silver- en gold-laag.** Volledige herberekening is acceptabel in bronze (waar data toch al ruw binnenkomt), maar wordt snel onbetaalbaar in silver/gold naarmate historie opbouwt.
- **Monitor query-patronen, niet alleen jobduur.** Een pipeline die binnen zijn tijdvenster blijft, kan alsnog inefficiënt zijn — kijk naar bytes gescand, niet alleen naar kloktijd.
- **Cache of materialiseer dure, veelgebruikte joins** in een aparte tussenlaag in plaats van dezelfde zware join in meerdere downstream-modellen te herhalen.
- **Zet auto-suspend standaard laag** (60-120 seconden) op elk interactief warehouse/cluster, en behandel een hogere waarde als een bewuste uitzondering, niet als standaardinstelling.
- **Meet vóór je optimaliseert.** Een architectuurwijziging die "logisch" klinkt maar niet gemeten is tegen een concreet knelpunt (kosten, latency, doorlooptijd) is giswerk — begin bij de metriek, niet bij de oplossing.

## Interviewvragen

**"Hoe zou je een dataplatform ontwerpen voor een bedrijf dat net begint met data engineering?"** Let op: een pleidooi voor eenvoud eerst — een basale bronze/silver/gold-structuur op één platform, geen premature streaming of multi-platform-complexiteit, met ruimte om later uit te breiden.

**"Wanneer zou je Databricks kiezen boven Snowflake, en andersom?"** Let op: een concreet, workload-gebaseerd antwoord (SQL-analytics vs. ML/Python-zwaar), niet een oppervlakkige featurevergelijking.

**"Leg uit wat medallion architecture is en waarom elke laag bestaat."** Let op: begrip van foutisolatie en onafhankelijke herbouwbaarheid als de kernreden, niet alleen "bronze is ruw, gold is schoon".

**"Wat is het verschil tussen orchestratie en declaratieve pipelines zoals dbt of Delta Live Tables?"** Let op: orchestratie bepaalt wanneer en in welke volgorde stappen draaien (imperatief); declaratieve tools leiden dat automatisch af uit gedefinieerde afhankelijkheden en een gewenste eindtoestand.

**"Hoe zou je kostenbeheersing inbouwen in een nieuw dataplatform vanaf het begin?"** Let op: auto-suspend/auto-scaling instellingen, aparte compute per workload, budgetalerts, en periodieke review van query-patronen — niet pas kostenbeheersing toevoegen nadat de rekening al hoog is.

**"Wat is een data contract, en welk probleem lost het op?"** Let op: een expliciete, afdwingbare afspraak tussen een producerend en consumerend team over een schema, met een concreet voorbeeld van hoe dat een productieincident had kunnen voorkomen — niet alleen "het is documentatie".

**"Hoe zou je een externe, klant-facing feature op je dataplatform aansluiten zonder je interne analytics te vertragen?"** Let op: het besef dat directe queries tegen het warehouse vanuit een externe applicatie een architectuurfout is, en een voorstel voor een aparte serving-laag (API, reverse ETL, of een gerepliceerde read-store).

**"Hoe pas je je architectuur aan naarmate een organisatie groeit van 10 naar 200 medewerkers?"** Let op: herkenning dat architectuur moet meegroeien met organisatorische complexiteit (meerdere teams, meerdere bronnen, externe gebruikers) in plaats van in één keer "compleet" gebouwd te worden — de kern van de Flowmetric-case study.

## Relevante Documentatie

- Snowflake Architecture Overview — officiële uitleg van de storage/compute-scheiding.
- Snowflake Warehouse Considerations — sizing en auto-suspend in de praktijk.
- Databricks Lakehouse Architecture — hoe Delta Lake ACID-garanties toevoegt aan een data lake.
- Microsoft Fabric Architecture — hoe de unified Fabric-lagen samenhangen.
- dbt Labs: How We Structure Our dbt Projects — de industriestandaard voor bronze/silver/gold-structurering binnen dbt.
- dbt Labs: Data Contracts — hoe dbt schema-afspraken native ondersteunt via model contracts.

## Samenvatting

Een modern dataplatform draait niet om welk specifiek product je kiest, maar om architectonische principes die platformonafhankelijk zijn: ingestion losgekoppeld van transformatie, storage en compute die onafhankelijk schalen, transformatielogica vastgelegd als versiebeheerde code, en expliciete data contracts op de grenzen tussen teams. Medallion architecture (bronze/silver/gold) is het patroon dat deze principes in de praktijk brengt, met foutisolatie als kernvoordeel. De Flowmetric-case study laat zien dat de juiste architectuur niet vaststaat, maar meegroeit met de fase van de organisatie — wat in fase 1 verstandige eenvoud is, is in fase 3 een risico, en andersom. De keuze tussen Snowflake, Databricks en Fabric is een workload-vraag met een reëel, meetbaar kostenaspect (zoals het warehouse-sizingvoorbeeld liet zien), geen religieuze discussie. Met deze architectuurprincipes op zak ben je klaar voor de platformspecifieke hoofdstukken die hierna volgen.