DATA ENGINEERING HANDBOOK

Data Engineering Handbook

Learn how modern data platforms are designed, built and operated. From SQL and data modeling to Snowflake, Databricks, dbt and Microsoft Fabric.

Door Abdullah Özisik — DataPartner365 · 09-08-2026

Inhoudsopgave

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.

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.

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.

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.

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. 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.

Silver bevat schone, gededupliceerde, correct getypeerde data — één betrouwbare rij per entiteit, maar nog niet noodzakelijk geaggregeerd of verrijkt met businesslogica. 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.

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.

De praktische vuistregel: 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.

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

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.

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.

Orchestratie & de Opkomst van Declaratieve Pipelines

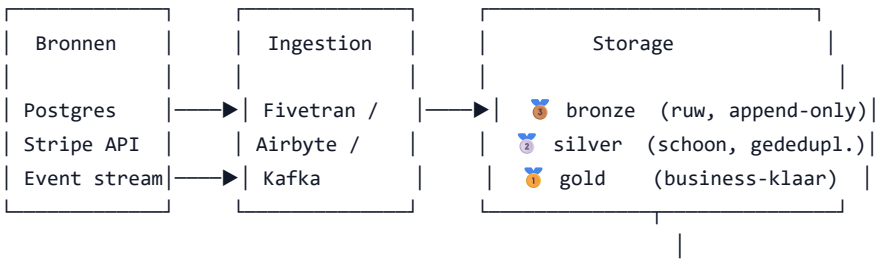
Een merkbare verschuiving in platformarchitectuur de afgelopen jaren is de beweging van imperatieve orchestratie ("draai stap A, dan stap B, dan stap C, in deze volgorde, op dit tijdstip") naar declaratieve pipelines ("dit is de gewenste eindtoestand van deze tabel; het platform bepaalt zelf hoe en wanneer die actueel gehouden wordt").

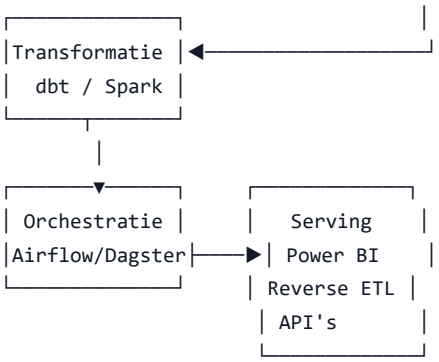
dbt was hier vroeg bij: je definieert modellen als SQL-`SELECT`-statements en dbt leidt de afhankelijkheidsgrafiek automatisch af uit `ref()`-aanroepen, in plaats van dat jij handmatig een DAG met volgordes onderhoudt. Snowflake's Dynamic Tables en Databricks' Delta Live Tables (DLT) trekken dit patroon 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.

Een Referentiearchitectuur

Een representatieve architectuur voor een middelgroot dataplatform, 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.

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.

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.

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.
- **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.

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.
- **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.

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.

Relevante Documentatie

- Snowflake Architecture Overview — officiële uitleg van de storage/compute-scheiding.
- 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.

Samenvatting

Een modern dataplatform draait niet om welk specifiek product je kiest, maar om drie architectonische principes die platformonafhankelijk zijn: ingestion losgekoppeld van transformatie, storage en compute die onafhankelijk schalen, en transformatielogica vastgelegd als versiebeheerde, testbare code. Medallion architecture (bronze/silver/gold) is het patroon dat deze principes in de praktijk brengt, met foutisolatie als kernvoordeel. De keuze tussen Snowflake, Databricks en Fabric is een workload-vraag, geen religieuze — SQL-zware analytics, Python/ML-zware verwerking, en Microsoft-ecosysteemintegratie wijzen elk een andere kant op. De verschuiving richting declaratieve pipelines (dbt, Dynamic Tables, Delta Live Tables) vermindert de handmatige orchestratielast binnen een platform, maar vervangt orchestratietools als Airflow niet voor cross-platform-coördinatie. Met deze architectuurprincipes op zak ben je klaar voor de platformspecifieke hoofdstukken die hierna volgen.