Een Datawarehouse Opzetten van A tot Z

Praktische handleiding voor beginnende data engineers

Datapartner365

by Datapartner365

Inhoudsopgave

1. Inleiding tot Datawarehousing
2. Architectuurprincipes en -patronen
3. Requirements en Ontwerp
4. Data Extract, Transform, Load (ETL)
5. Data Modelling Technieken
6. Data Kwaliteit en Governance
7. Cloud Datawarehouses
8. Performance Tuning en Optimalisatie
9. Monitoring en Onderhoud
10. Case Study: Volledige Implementatie

Inleiding tot Datawarehousing

Wat is een datawarehouse?

15 min leestijd Beginner

Een **datawarehouse** (DWH) is een centrale opslagplaats van geïntegreerde data uit één of meer bronsystemen, geoptimaliseerd voor analyse en rapportage. De klassieke definitie komt van Bill Inmon (1992): "*Een subject-oriented, integrated, time-variant, en non-volatile collectie van data, ter ondersteuning van management beslissingen.*"

Die vier eigenschappen vormen nog steeds de kern:

- **Subject-oriented** — georganiseerd rond bedrijfsthema's (klant, order, product) in plaats van rond applicaties.
- **Integrated** — data uit meerdere bronnen wordt geconsolideerd, geconverteerd en geharmoniseerd.
- **Time-variant** — historische data blijft bewaard, vaak met expliciete tijdstempels en SCD-mechanismen.
- **Non-volatile** — data wordt niet zomaar overschreven; nieuwe data wordt toegevoegd.

> **Kernidee**
>
> Een datawarehouse beantwoordt vragen als "*hoeveel verkochten we vorig kwartaal per regio?*" snel, terwijl een operationele database geoptimaliseerd is voor "*hoeveel staat er nu in voorraad voor product X?*". Dat verschil — analytische workload vs. transactionele workload — is de fundamentele reden waarom datawarehouses bestaan.

Korte geschiedenis

Datawarehousing als discipline ontstond in de jaren '80 toen bedrijven beseften dat hun operationele systemen ongeschikt waren voor analyses. De grote namen:

- **Bill Inmon** — "vader van het datawarehouse", introduceerde de top-down enterprise-warehouse aanpak (Corporate Information Factory).
- **Ralph Kimball** — pleitte voor bottom-up dimensionele modellen (data marts via een bus architecture).
- **Dan Linstedt** — bedenker van Data Vault (begin jaren 2000), gericht op auditbaarheid en agility.

De technologie evolueerde mee: van Teradata-appliances en Oracle Exadata in de jaren '90, naar MPP-databases (Netezza, Vertica) in de jaren 2000, naar de cloud-revolutie sinds ongeveer 2013 met Redshift, gevolgd door BigQuery, Snowflake, Synapse en Microsoft Fabric.

OLTP vs. OLAP

De belangrijkste tegenstelling om te begrijpen is die tussen **OLTP** (Online Transaction Processing) en **OLAP** (Online Analytical Processing). Beide zijn databases, maar met fundamenteel andere workloads.

Eigenschap	OLTP (operationeel)	OLAP (analytisch)
Doel	Transacties verwerken	Vragen beantwoorden
Querytype	Veel kleine, kortdurend	Weinig grote, lang lopend
Data volume	GB tot TB	TB tot PB
Schemavorm	Genormaliseerd (3NF)	Gedenormaliseerd (star)
Indexen	B-tree op primaire keys	Columnstore, partitions
Updates	Realtime, frequent	Batch, periodiek
Voorbeeld	PostgreSQL, SQL Server	Snowflake, BigQuery

Voorbeeld: dezelfde vraag, andere database

Stel je verkoopt online en wil weten: "Wat is de totale omzet per productcategorie in Q1 2026?"

In een OLTP-systeem zou die query er ruwweg zo uitzien:

```
 -- OLTP: zware join over genormaliseerde tabellen
SELECT c.category_name, SUM(oi.quantity * oi.unit_price) AS revenue
FROM order_items oi
JOIN orders o          ON o.order_id = oi.order_id
JOIN products p        ON p.product_id = oi.product_id
JOIN categories c      ON c.category_id = p.category_id
WHERE o.order_date BETWEEN '2026-01-01' AND '2026-03-31'
  AND o.status = 'completed'
GROUP BY c.category_name
ORDER BY revenue DESC;
```

Op een datawarehouse met een star schema zou diezelfde vraag eenvoudiger en vele malen sneller zijn:

```
 -- OLAP: gedenormaliseerd star schema, één join
SELECT p.category_name, SUM(f.revenue) AS revenue
FROM fact_sales f
JOIN dim_product p ON p.product_key = f.product_key
WHERE f.date_key BETWEEN 20260101 AND 20260331
GROUP BY p.category_name
ORDER BY revenue DESC;
```

Het verschil zit niet alleen in de query — het zit vooral in hoe de data is opgeslagen, geïndexeerd en gepartitioneerd. Een columnstore-engine leest alleen de drie kolommen die de query nodig heeft, slaat

partitions met andere datums over, en aggregeert miljoenen rijen in seconden.

Datawarehouse vs. data lake vs. lakehouse

De moderne data-architectuur kent drie verwante concepten:

- **Data warehouse** — gestructureerde data, schema-on-write, geoptimaliseerd voor SQL en BI.
- **Data lake** — alle datatypes, schema-on-read, goedkoop op object storage (S3, ADLS, GCS).
- **Lakehouse** — combineert beide: ACID-transacties op object storage via Delta Lake, Iceberg of Hudi. Snowflake, Databricks en Microsoft Fabric bewegen allemaal in deze richting.

> **Praktische regel**
>
> Heb je voornamelijk gestructureerde data en BI-vragen? Een traditioneel datawarehouse is het meest pragmatisch. Werk je met grote hoeveelheden semi-gestructureerde data (JSON, logs) en machine learning? Een lakehouse-architectuur loont. Pure data lakes zonder governance laag (Delta/Iceberg) zijn inmiddels achterhaald — die werden vroeger "data swamps".

Waarom een datawarehouse bouwen?

Bedrijven investeren om vier hoofdredenen in een datawarehouse:

1. **Single source of truth** — finance, marketing en operations gebruiken vaak verschillende cijfers voor "omzet". Een DWH dwingt één definitie af.
2. **Performance** — analyses op operationele databases zetten productie onder druk en zijn traag. Een DWH offload de werkdruk.
3. **Historie** — operationele systemen bewaren vaak alleen de huidige staat. Een DWH bewaart hoe de wereld er gisteren, vorige maand en vorig jaar uitzag.
4. **Integratie** — data uit ERP, CRM, e-commerce, logistiek en marketing tools wordt gekoppeld op klant, product en tijd.

De drie lagen van een datawarehouse

Vrijwel elk datawarehouse — ongeacht het patroon — heeft een logische opbouw in drie lagen:

- **Staging / bronzelaag** — ruwe kopie van de bron, zo ongewijzigd mogelijk. Doel: bron loskoppelen van transformaties.
- **Integratie / silverlaag** — opgeschoond, geharmoniseerd, deduplicaten verwijderd, business keys gestandaardiseerd. Hier woont de business logic.
- **Presentatie / goldlaag** — dimensionele modellen (star schemas) of data marts, geoptimaliseerd voor consumptie door BI-tools.

In de Microsoft Fabric en Databricks-wereld worden deze lagen ook wel **Bronze / Silver / Gold** genoemd (de "medallion architecture"). De namen verschillen, het concept is hetzelfde.

Wie werkt aan een datawarehouse?

Een datawarehouse-traject is geen one-man show. De typische rollen die je tegenkomt — al dan niet gecombineerd in één persoon bij kleinere organisaties:

- **Data engineer** — bouwt en onderhoudt pipelines, modelleert silver-laag, optimaliseert performance.
- **Analytics engineer** — bouwt de gold-laag in dbt, definieert metrics en zorgt dat BI-tools clean datasets krijgen.
- **Data analyst / BI-developer** — bouwt rapportages en dashboards op de goldlaag, vertaalt business-vragen naar metrieken.
- **Data architect** — kiest patronen, schrijft conventies, bewaakt consistentie over teams en projecten.
- **Data product owner** — prioriteert wat er gebouwd wordt en stemt af met stakeholders uit business-domeinen.
- **Data steward** — eigenaar van datakwaliteit en definities binnen een specifiek domein (klant, product, finance).

De grenzen tussen deze rollen zijn fluide. In een mkb is "de data engineer" vaak alle bovenstaande rollen tegelijk. In een enterprise zijn het tien teams. Wat blijft: zonder duidelijke verantwoordelijkheid voor kwaliteit en definities — de data steward-rol — eindigt elk DWH op termijn als black box.

Eenvoudig Python-voorbeeld: tijdsregistratie

Een tijdsdimensie is vrijwel altijd de eerste tabel die je bouwt. Hier een minimaal Python-script dat een dimensie genereert van 2020 t/m 2030:

```
 import pandas as pd

dates = pd.date_range("2020-01-01", "2030-12-31", freq="D")
dim_date = pd.DataFrame({
    "date_key":   dates.strftime("%Y%m%d").astype(int),
    "full_date":  dates,
    "year":       dates.year,
    "quarter":    dates.quarter,
    "month":      dates.month,
    "month_name": dates.month_name(locale="nl_NL"),
    "day":        dates.day,
    "weekday":    dates.day_name(locale="nl_NL"),
    "is_weekend": dates.weekday >= 5,
})

dim_date.to_csv("dim_date.csv", index=False)
print(f"{len(dim_date):,} datumrijen aangemaakt.")
```

De `date_key` als integer (YYYYMMDD) is een klassieke truc: het is sorteerbaar, joinbaar én leesbaar. Je ziet meteen of een record uit 2024 of 2026 komt.

Typische use cases — waar levert een DWH écht waarde?

Een datawarehouse is geen doel op zich. Concrete use cases die de investering rechtvaardigen:

- **360°-klantbeeld** — sales-data uit het CRM, support-tickets uit Zendesk, bestellingen uit het e-commerce platform en marketing-touchpoints uit GA4 worden geïntegreerd op klant-niveau. Zonder DWH blijft elke afdeling een eigen versie van de klant zien.
- **Financiële consolidatie** — meerdere ERP-instances (per land, per dochter) worden samengevoegd op een geharmoniseerde rekeningstructuur. Maandafsluitingen die voorheen drie weken kostten gaan terug naar drie dagen.
- **Operations dashboards** — live KPI's op voorraad, doorlooptijden, kwaliteit. Het DWH ontlast de productie-database en levert tegelijk historische context (deze week vs. vorig jaar).
- **Compliance- en audit-rapportages** — herleidbaarheid van transacties, GDPR-rechten op verzoek, regulator-rapportages. Een goed gemodelleerd DWH met audit-trail maakt dit beheersbaar.
- **Machine learning feature stores** — voorbereidde features voor ML-modellen (klantsegmentatie, fraud-detectie, churn-predictie). Het DWH levert de gehistoriseerde, opgeschoonde feiten waarop modellen getraind en geïnferentieerd worden.

Wanneer heb je *geen* datawarehouse nodig?

Niet elk bedrijf heeft een DWH nodig. Goede signalen dat je het (nog) niet hoeft te bouwen:

- Eén bronsysteem, kleine dataset (< 100 GB), weinig analytische gebruikers.
- Rapportages die makkelijk uit Excel of een ingebouwde rapportagemodule komen.
- Geen historische analyses nodig — alleen actuele cijfers.

In dat geval volstaat vaak een read replica van je productie-database, of een direct-query rapportagetool. Een DWH bouwen "voor het geval dat" is duur en levert weinig op.

> **Key takeaways**
>
> - Een datawarehouse is geoptimaliseerd voor analytische queries, niet voor transactionele.
> - OLTP en OLAP hebben fundamenteel verschillende ontwerpdoelen — verwar ze niet.
> - De drie lagen (staging / integratie / presentatie) komen in elk patroon terug.
> - Een lakehouse is de moderne convergentie van DWH en data lake.
> - Bouw alleen een DWH als historie, integratie of performance een echt probleem zijn.

Veelgestelde vragen

Wat is een datawarehouse?

Een centrale opslagplaats van geïntegreerde data uit één of meer bronsystemen, geoptimaliseerd voor analyse en rapportage. Inmon's klassieke definitie: subject-oriented, integrated, time-variant en non-volatile, ter ondersteuning van management beslissingen.

Wat is het verschil tussen OLTP en OLAP?

OLTP is geoptimaliseerd voor veel kleine transacties op een genormaliseerd schema (operationele databases). OLAP is geoptimaliseerd voor weinig grote analytische queries op gedenormaliseerde star schemas met columnstore-opslag (datawarehouses).

Wat is het verschil tussen een datawarehouse en een data lake?

Een DWH bewaart gestructureerde data met schema-on-write voor SQL en BI. Een data lake bewaart alle datatypes goedkoop op object storage met schema-on-read. Een lakehouse combineert beide via Delta Lake, Iceberg of Hudi.

Wanneer heb je een datawarehouse nodig?

Bij meerdere bronsystemen, behoefte aan historische analyses, en analytische queries die je productiedatabase belasten. Met één bron, kleine dataset en weinig vragen volstaat een read replica of een rapportagetool met direct query.

Wat is het verschil tussen Inmon en Kimball?

Inmon werkt top-down met een centraal genormaliseerd 3NF-warehouse waar marts uit worden afgeleid. Kimball werkt bottom-up met dimensionele star schemas per business proces. Beide zijn geldig — Kimball voor snelheid, Inmon of Data Vault voor enterprise-governance.

Wat zijn de drie lagen van een datawarehouse?

Staging/bronze (ruwe kopie), integratie/silver (opgeschoond en geharmoniseerd) en presentatie/gold (dimensionele modellen voor BI). Deze medallion-aanpak komt in vrijwel elk patroon terug.

Architectuurprincipes en -patronen

De drie grote stromingen

18 min leestijd Beginner-Gevorderd

De vraag "hoe ontwerp je een datawarehouse?" heeft drie klassieke antwoorden, elk met een eigen filosofie en sterke aanhang in de community. Het is geen religieuze keuze — moderne projecten mengen vaak elementen — maar je moet ze kennen om te begrijpen waarom je iets doet zoals je het doet.

1. Kimball — dimensionele bottom-up aanpak

Ralph Kimball publiceerde in 1996 *The Data Warehouse Toolkit*, dat tot op de dag van vandaag de facto bijbel is voor BI-developers. Zijn aanpak:

- **Bottom-up**: bouw eerst data marts per business proces (sales, inventory, finance), die later samen het enterprise warehouse vormen.
- **Dimensioneel modelleren**: star schemas met fact tables (metingen) en dimension tables (context).
- **Conformed dimensions**: dezelfde klantdimensie wordt hergebruikt in elke data mart, zodat marts onderling vergelijkbaar zijn.
- **Bus matrix**: een tabel met business processen op de rij-as en dimensies op de kolom-as, die laat zien welke dimensies waar gebruikt worden.

Voorbeeld van een minimaal Kimball star schema:

```sql
 -- Fact table: één rij per verkoopregel
CREATE TABLE fact_sales (
  sales_key       BIGINT IDENTITY,
  date_key        INT,          -- FK naar dim_date
  customer_key    INT,          -- FK naar dim_customer
  product_key     INT,          -- FK naar dim_product
  store_key       INT,          -- FK naar dim_store
  quantity        INT,
  unit_price      DECIMAL(10,2),
  revenue         DECIMAL(12,2),
  load_timestamp  TIMESTAMP
);

-- Dimension table: contextuele attributen, gedenormaliseerd
CREATE TABLE dim_customer (
  customer_key    INT IDENTITY PRIMARY KEY,    -- surrogate key
  customer_id     VARCHAR(50),                 -- business key
  name            VARCHAR(200),
  segment         VARCHAR(50),
  country         VARCHAR(50),
  city            VARCHAR(100),
  valid_from      DATE,
  valid_to        DATE,
  is_current      BOOLEAN
);
```

De surrogate keys (`customer_key`) ontkoppelen het warehouse van de bron-IDs. Dat lijkt overdreven, maar het maakt SCD Type 2-historie mogelijk en beschermt je tegen wijzigingen aan de business key.

> **👍 Sterk in**
>
> Snelle time-to-value, intuïtief voor business users, uitstekend voor BI-tools (Power BI, Tableau, Looker zijn er allemaal op gebouwd). Je levert binnen weken een werkende data mart op.

Slowly Changing Dimensions in Kimball

Een dimensie is zelden statisch — klanten verhuizen, productcategorieën worden hernoemd, salesgebieden gereorganiseerd. Kimball heeft hiervoor zeven SCD-types gedefinieerd, waarvan deze drie het meest gebruikt worden:

- **SCD Type 1 — overschrijven**: oude waarde wordt vervangen, geen historie. Geschikt voor correcties van typefouten of velden waar historie niet relevant is.
- **SCD Type 2 — historische rijen**: bij elke wijziging een nieuwe rij met `valid_from` , `valid_to` en `is_current` . De gouden standaard voor analyses op het juiste moment in de tijd.

- **SCD Type 3 — vorige waarde naast huidige**: voeg een kolom `previous_segment` toe naast `segment`. Compact, maar onthoudt slechts één wijziging terug.

SCD Type 2 is in 90% van de gevallen wat je wilt voor business-kritische dimensies. Type 1 voor velden waar niemand naar terugkijkt, Type 3 voor zeldzame "voor en na"-vergelijkingen.

> ⚠️ **Wanneer Kimball NIET de juiste keuze is**
>
> Vermijd Kimball als single source of truth wanneer je tientallen bronsystemen integreert die elkaar overlappen of tegenspreken — je krijgt dan conformed dimensions die voortdurend gestuurd moeten worden. Ook bij strikte regulatory requirements (financieel, medisch) loop je tegen de auditgrens aan: dimensionele modellen leggen niet automatisch herkomst en wijzigingsgeschiedenis vast op recordniveau.

2. Inmon — top-down enterprise warehouse

Bill Inmon ziet het anders: bouw eerst een centraal, genormaliseerd **Enterprise Data Warehouse** in 3NF, en lever data marts daaruit af. Dat staat bekend als de **Corporate Information Factory** (CIF).

- **Top-down**: eerst een ondernemingsbreed model, dan pas use-case-specifieke data marts.
- **3NF in de kern**: minimale redundantie, hoge data integriteit.
- **Single version of the truth**: één centrale waarheid, marts zijn afgeleiden.

Voorbeeld 3NF-fragment in Inmon-stijl:

```
CREATE TABLE customer (
  customer_id   INT PRIMARY KEY,
  name          VARCHAR(200),
  segment_id    INT REFERENCES segment(segment_id),
  address_id    INT REFERENCES address(address_id),
  created_at    TIMESTAMP
);

CREATE TABLE address (
  address_id    INT PRIMARY KEY,
  street        VARCHAR(200),
  city_id       INT REFERENCES city(city_id),
  postal_code   VARCHAR(10)
);

CREATE TABLE city (
  city_id       INT PRIMARY KEY,
  city_name     VARCHAR(100),
  country_id    INT REFERENCES country(country_id)
);
```

Voor analyses bouw je vervolgens dimensionele marts *op basis van* dit centrale model. Het kost meer initieel, maar het schaalt voor grote organisaties met veel bronsystemen.

> **Wanneer Inmon NIET de juiste keuze is**
>
> Inmon vraagt veel investering vooraf — modelleurs die de hele organisatie moeten doorgronden voordat je iets oplevert. Voor scale-ups, mkb's en projecten met snelle time-to-value-eisen is dit vaak een dealbreaker. Ook als je business agile werkt en wekelijks nieuwe data marts wil opleveren, blokkeert het centrale 3NF-model meer dan dat het helpt.

3. Data Vault — agility en auditbaarheid

Dan Linstedt's **Data Vault** (DV2.0) is de jongste van de drie. Het probeert het schaalvoordeel van Inmon te combineren met de agility van Kimball, en voegt sterke auditbaarheid toe — bruikbaar in regulated industries (finance, pharma, insurance).

Data Vault kent drie soorten tabellen:

- **Hubs** — bevatten alleen business keys + load metadata. Eén hub per kernentiteit (klant, product, order).
- **Links** — many-to-many relaties tussen hubs.
- **Satellites** — alle beschrijvende attributen + historie, gehangen aan een hub of link.

```sql
 -- Hub: alleen business key + metadata
CREATE TABLE hub_customer (
  customer_hk     CHAR(32),          -- hash key
  customer_id     VARCHAR(50),       -- business key
  load_dts        TIMESTAMP,
  record_source   VARCHAR(50)
);

-- Satellite: beschrijvende data, historisch
CREATE TABLE sat_customer_details (
  customer_hk     CHAR(32),
  load_dts        TIMESTAMP,
  hash_diff       CHAR(32),          -- detecteert wijzigingen
  name            VARCHAR(200),
  segment         VARCHAR(50),
  country         VARCHAR(50),
  record_source   VARCHAR(50),
  PRIMARY KEY (customer_hk, load_dts)
);

-- Link: relatie tussen hubs
CREATE TABLE link_customer_order (
  link_hk         CHAR(32),
  customer_hk     CHAR(32),
  order_hk        CHAR(32),
  load_dts        TIMESTAMP,
  record_source   VARCHAR(50)
);
```

De volledige historie en herkomst (`record_source` , `load_dts`) zit ingebouwd in elke tabel — handig voor audits en GDPR. Nadeel: zonder een goede automatiseringslaag (bijvoorbeeld dbtvault, Wherescape, Vaultspeed) wordt Data Vault snel een berg boilerplate-code.

Hash keys en hash diff — twee patronen die je móet kennen

Het hashen van business keys (bijvoorbeeld MD5 of SHA-256) doet drie dingen: het maakt parallel laden mogelijk (geen lookup nodig), het maakt bron-overschrijdende joins triviaal (zelfde input → zelfde hash), en het levert een vaste sleutellengte op. De `hash_diff` in satellites is een hash over alle beschrijvende attributen — als hij gelijk is aan de vorige rij, wordt geen nieuwe rij toegevoegd. Zo voorkom je duplicaten zonder dure rij-vergelijkingen.

> **Wanneer Data Vault NIET de juiste keuze is**
>
> Voor kleine teams zonder ervaring met DV2.0 en zonder automatiseringstool is Data Vault zelden verstandig. De methodologie genereert per bronentiteit al snel 3-5 tabellen plus laadlogica, en

zonder generator wordt dat onbeheersbaar. Voor één bronsysteem en één analytisch domein is Kimball gewoon sneller en simpeler.

Vergelijking

Aspect	Kimball	Inmon	Data Vault
Aanpak	Bottom-up	Top-down	Modulair / agile
Modelvorm	Star schema	3NF	Hubs / Links / Satellites
Time-to-value	Weken	Maanden tot jaren	Weken (met automatisering)
Schaalbaarheid	Goed	Uitstekend	Uitstekend
Flexibiliteit bij verandering	Matig	Lastig	Hoog (additief)
Auditbaarheid	Beperkt	Beperkt	Ingebouwd
Leercurve	Laag	Hoog	Hoog
BI-tool friendly	Direct	Via marts	Via marts (Information Mart Layer)

De medallion architecture (modern)

Sinds de opkomst van het lakehouse (Delta Lake, Iceberg) is de **medallion architecture** populair geworden — vooral in Databricks en Microsoft Fabric:

- **Bronze** — ruwe ingest van bronsystemen, immutable, append-only.
- **Silver** — opgeschoond, geharmoniseerd, deduplicaten verwijderd. Vaak in een Data Vault- of 3NF-stijl.
- **Gold** — business-ready dimensionele modellen, meestal Kimball-stijl, voor BI en ML.

De medallion-laagjes komen overeen met staging / integratie / presentatie uit hoofdstuk 1, maar dan op een lakehouse-platform. Dezelfde principes, modernere tooling.

> **Praktische combinatie**
>
> Veel moderne projecten combineren: Data Vault in silver (voor auditbaarheid en flexibiliteit), Kimball in gold (voor BI-tooling). Dat geeft je het beste van beide werelden — auditbare opslag onder de motorkap, intuïtieve modellen voor de eindgebruiker.

Hoe kies je het juiste patroon?

Een eenvoudige beslissingsboom:

1. **Klein team, één bedrijfstak, snel resultaat nodig?** → Kimball.
2. **Grote enterprise met tientallen bronnen, governance kritisch?** → Inmon of Data Vault.
3. **Strikte audittrail nodig (financieel, medisch, verzekering)?** → Data Vault.
4. **Lakehouse-platform met machine learning naast BI?** → Medallion (Bronze/Silver/Gold).
5. **Mengsel?** → Hybride, met dimensionele goldlaag op een Data Vault-silverlaag.

Praktijkscenario: een hybride aanpak in een retail-omgeving

Een Nederlandse retailer met 12 webshops in vijf landen had twee problemen: marketing wilde elke week een nieuwe campagne-rapportage, finance vroeg een GDPR-proof audittrail op klantniveau. Een pure Kimball-aanpak loste het tweede niet op, een pure Data Vault-aanpak was te traag voor het eerste. De gekozen architectuur:

- **Bronze (Delta Lake)** — alle CDC-streams van Kafka, één bestand per source per uur, append-only.
- **Silver (Data Vault 2.0)** — hubs voor klant, order, product, store; satellites met SCD2-historie. dbtvault genereerde 80% van de DDL en transformaties.
- **Gold (Kimball star schemas)** — fact_sales, fact_inventory, dim_customer (Type 2), dim_product (Type 1 voor naamswijzigingen). Power BI rapporten draaiden direct op gold.

Resultaat: marketing kreeg dashboards binnen een sprint, finance kreeg een herleidbare audittrail per record, en nieuwe bronsystemen (een nieuwe webshop, een logistieke partner) konden in dagen worden toegevoegd zonder bestaande modellen te raken. De prijs: drie ETL-lagen onderhouden in plaats van één — maar elk laag had een duidelijk doel.

Anti-patterns om te vermijden

- **"Sterk-hub-schema"** — een fact-tabel die direct met productie-IDs joint zonder surrogate keys. Bij elke schema-change in de bron breekt je warehouse.
- **"One Big Table"** — één enorme gedenormaliseerde tabel waar alles in zit. Werkt voor demo's, breekt onder schaal.
- **"Spaghettistraal"** — geen integratielaag, marts trekken rechtstreeks uit verschillende bronnen. Definities lopen uiteen, garbagedata.
- **"Te diep genormaliseerd in goldlaag"** — eindgebruikers in BI-tools willen wide tables, geen 12-way joins.

Key takeaways

- Kimball, Inmon en Data Vault zijn geen vijanden — ze beantwoorden verschillende vragen.
- Kimball is het snelst, Inmon het meest gestructureerd, Data Vault het meest flexibel.
- De medallion architecture is de moderne herinterpretatie van de drielagen-aanpak.
- Een hybride keuze (DV in silver, Kimball in gold) is in de praktijk vaak optimaal.
- Surrogate keys zijn niet optioneel — ze zijn de levensader van een goed warehouse.

Veelgestelde vragen

Wat is het verschil tussen Kimball en Inmon?

Kimball werkt bottom-up met dimensionele star schemas per business proces. Inmon werkt top-down met een centraal genormaliseerd 3NF Enterprise Data Warehouse, waar data marts uit worden afgeleid. Kimball levert sneller resultaat op, Inmon schaalt beter voor grote organisaties met veel bronsystemen en strikte governance-eisen.

Wanneer kies je voor Data Vault?

Data Vault is de juiste keuze bij strikte audit-eisen (banken, verzekeraars, pharma), bij veel bronsystemen die regelmatig wijzigen, en wanneer je flexibel nieuwe bronnen wilt toevoegen zonder bestaande modellen te breken. Niet kiezen als je team klein is en geen automatiseringstool inzet — dan word je begraven in boilerplate.

Wat is medallion architecture?

Medallion architecture verdeelt data in drie lagen op een lakehouse: bronze (ruwe ingest, immutable), silver (geschoond, geharmoniseerd) en gold (business-ready dimensionele modellen). Het is de moderne herinterpretatie van de klassieke staging/integratie/presentatie-aanpak, populair bij Databricks en Microsoft Fabric met Delta Lake of Apache Iceberg als opslagformaat.

Kun je Kimball en Data Vault combineren?

Ja, dit is in de praktijk vaak de optimale keuze. Data Vault in silver voor auditbaarheid en flexibiliteit, Kimball-stijl star schemas in gold voor BI-tools. Zo combineer je sterke historiek onder de motorkap met intuïtieve modellen voor eindgebruikers — het beste van beide werelden.

Welke architectuur past bij Microsoft Fabric of Databricks?

Beide platforms zijn geoptimaliseerd voor medallion architecture met Delta Lake. In gold leg je dimensionele modellen aan in Kimball-stijl, omdat Power BI en Tableau daarop geoptimaliseerd zijn. Voor compliance-zware projecten kun je in silver een Data Vault aanleggen — dbtvault werkt prima op beide platforms.

Hoeveel SCD-types gebruik je in de praktijk?

In de praktijk kom je vooral SCD Type 1 (overschrijven, geen historie) en SCD Type 2 (historische rijen met valid_from/valid_to) tegen. SCD Type 3 (vorige waarde naast huidige) is bruikbaar voor "voor en na"-vergelijkingen. Types 4 t/m 7 zijn academisch interessant maar zelden nodig.

Requirements en Ontwerp

Beginnen bij de business

16 min leestijd Beginner

De grootste reden dat datawarehouse-projecten falen is niet techniek — het is dat de business nooit echt heeft uitgesproken wat ze van het systeem verwachten. Een goed ontwerp begint bij vragen, niet bij tabellen. Dit hoofdstuk loopt door het traject van requirements naar technisch ontwerp, met de templates die je in de praktijk kunt gebruiken.

Het belang van een goed begin

Onderzoek door Gartner en TDWI laat consequent zien dat 50-70% van datawarehouse-trajecten de oorspronkelijke doelstellingen niet of slechts gedeeltelijk haalt. De oorzaken liggen vrijwel nooit bij gebrek aan technische kunde — ze liggen bij onduidelijke scope, gebrekkig stakeholder-management en verkeerde aannames over wat de business eigenlijk wil meten. Investeer twee tot vier weken aan requirements en design voor je één tabel bouwt; je verdient die tijd in de bouwfase dubbel terug.

Stap 1: stakeholders identificeren

Voordat je één regel SQL schrijft, moet helder zijn wie je opdrachtgever is en wie je gebruikers zijn. Onderscheid:

- **Sponsor** — wie betaalt en krijgt de waarde? Vaak een directielid (CFO, CCO).
- **Power users** — analisten en BI-developers die dagelijks queries draaien.
- **Casual consumers** — managers die dashboards bekijken.
- **Data owners** — eigenaren van de bronsystemen (CRM, ERP, e-commerce).
- **Operators** — DevOps / platform-teams die het draaiend houden.

Houd één-op-één-gesprekken met deze groepen, geen plenaire sessies. In een groep zwijgt de minderheid.

Stap 2: functionele requirements

Functionele eisen beschrijven *wat* het warehouse moet kunnen. De meest praktische techniek is om met stakeholders een lijst van **analytical questions** op te stellen — concrete vragen die ze willen kunnen beantwoorden. Bijvoorbeeld:

- "Wat is de gemiddelde orderwaarde per klantsegment per maand?"
- "Hoeveel procent van de nieuwe klanten heeft binnen 90 dagen een tweede order geplaatst?"
- "Welke producten worden vaak samen gekocht?"

Deze vragen worden je **acceptatiecriteria**. Aan het einde van een sprint test je: kan ik deze vraag in < 10 seconden beantwoorden met één SQL-query?

> **💡 Anti-pattern: alleen velden vragen**
>
> Een veelgemaakte fout is vragen wat de stakeholder "in het rapport wil zien". Je krijgt dan een lijst velden zonder context. Vraag in plaats daarvan welke *beslissingen* ze willen nemen — daaruit volgen de velden vanzelf.

Stap 3: niet-functionele requirements

Even belangrijk en vaak vergeten:

- **Data freshness** — moet de data realtime zijn, dagelijks, wekelijks?
- **Latency / performance** — hoe snel moet een dashboard laden? < 3 sec is een gangbare BI-norm.
- **Concurrency** — hoeveel gelijktijdige gebruikers?
- **Volume** — hoeveel rijen per dag, hoeveel historie bewaren we?
- **Security** — row-level security, kolom-masking voor PII, audittrail?
- **Compliance** — GDPR, financiële regels, retentiebeleid?
- **Beschikbaarheid** — werktijden of 24/7? RPO/RTO bij uitval?
- **Budget** — max kosten per maand voor compute en storage?

Deze antwoorden bepalen de platformkeuze (hoofdstuk 7). Een freshness van 5 minuten is niet hetzelfde architectuurprobleem als een freshness van 24 uur.

Stap 4: bus matrix opstellen

De **bus matrix** (Kimball) is een eenvoudige maar krachtige tabel: business processen op de rij-as, dimensies op de kolom-as. Een vinkje in een cel betekent: "deze dimensie is relevant voor dit proces".

Business proces ↓ / Dimensie →	Datum	Klant	Product	Winkel	Medewerker	Leverancier
Verkoop	✓	✓	✓	✓	✓	
Inkoop	✓		✓	✓	✓	✓
Voorraad	✓		✓	✓		
Marketing	✓	✓	✓			
HR	✓			✓	✓	

De waarde zit niet in de tabel zelf, maar in het gesprek dat eraan voorafgaat. Het dwingt stakeholders om expliciet te zijn over wat ze nodig hebben, en het laat zien welke dimensies *conformed* moeten worden — als datum, klant en product overal terugkomen, moet je die maar één keer goed bouwen.

Stap 5: drie modelleerniveaus

Modelleer je warehouse in drie iteraties:

1. **Conceptual model** — entiteiten en relaties, business-taal. Geen attributen, geen technische keuzes. Past op één A3.
2. **Logical model** — entiteiten met attributen en datatypes, primary/foreign keys. Patroon-keuze (Kimball / Inmon / Data Vault) komt hier binnen.
3. **Physical model** — concrete DDL voor je gekozen platform. Partitioning, clustering, distribution keys, indexen.

Een fragment van een conceptual model in pseudo-notatie:

```
 -- Conceptueel: alleen entiteiten en relaties
Klant   ──── plaatst ────► Order
Order   ──── bevat ──────► OrderRegel
Product ──── op ─────────► OrderRegel
Winkel  ──── verkoopt ──► Order
```

En het logische model dat eruit volgt (Kimball-stijl):

```
CREATE TABLE fact_order_line (
  order_line_key  BIGINT,
  date_key        INT,
  customer_key    INT,
  product_key     INT,
  store_key       INT,
  order_id        VARCHAR(50),    -- degenerate dimension
  quantity        INT,
  unit_price      DECIMAL(10,2),
  discount_amount DECIMAL(10,2),
  net_amount      DECIMAL(12,2)
);
```

`order_id` staat hier als *degenerate dimension* in de fact table: het is een attribuut zonder eigen dimension table, omdat er verder geen interessante context aan hangt.

Stap 5b: dataprofiling vóór je modelleert

Tussen logical en physical model hoort altijd een ronde dataprofiling op de bron. Aannames die in de praktijk vrijwel altijd fout zijn:

- **"De business key is uniek"** — profiel het. Dubbele klant-IDs door fusies, herstarts of test-records komen vaker voor dan je denkt.
- **"Dit veld is altijd gevuld"** — een NULL-percentage van 12% verandert je model.
- **"De datatypes kloppen"** — dat datum-veld is een VARCHAR met 14 verschillende formaten.
- **"De cardinaliteit klopt"** — een veld dat 50 unieke waarden hoort te hebben heeft er 12.000 wegens vrije invoer.

Tools: `SELECT`-queries op de bron, dbt's source freshness checks, of dedicated profilers als Great Expectations en Soda Core. Documenteer bevindingen in je STTM voordat je modeldecisies in beton giet.

Stap 6: naming conventions

Naming is geen detail. Niets vertraagt onboarding zoveel als inconsistente namen. Spreek *vooraf* af:

- **Lowercase met underscores**: `fact_sales`, niet `FactSales` of `fact-sales`.
- **Prefixen voor type**: `dim_`, `fact_`, `stg_` (staging), `int_` (intermediate), `hub_`, `sat_`, `link_` (Data Vault).
- **Suffix** `_key` voor surrogate keys, `_id` voor business keys.
- **Booleans** beginnen met `is_` of `has_` (`is_active`, `has_subscription`).
- **Datums**: `_date` voor DATE, `_at` of `_ts` voor TIMESTAMP (`created_at`, `load_ts`).
- **Bedragen**: kolomnaam met valuta indien meerdere (`amount_eur`, `amount_usd`).
- **Bron-prefix in staging**: `stg_salesforce__accounts`, `stg_shopify__orders`.

> **Schrijf het op**
>
> Leg de afspraken vast in een `NAMING.md` in je repo. Verwijs ernaar in elke code review. Inconsistentie sluipt er anders in zodra een nieuw teamlid begint.

Stap 7: data dictionary

Voor elke tabel en kolom leg je vast: betekenis, herkomst, datatype, voorbeeld, toegestane waarden, verantwoordelijke. Tools die dit automatiseren: dbt docs, DataHub, Collibra, Alation. Voor kleinere projecten volstaat een markdown-file of een schemafile in YAML.

```
 # schema.yml — dbt-stijl data dictionary
version: 2
models:
  - name: dim_customer
    description: "Eén rij per klant, SCD Type 2 — actuele én historische vers
    columns:
      - name: customer_key
        description: "Surrogate key. Wijzigt bij elke SCD Type 2-mutatie."
        tests: [unique, not_null]
      - name: customer_id
        description: "Business key uit Salesforce."
        tests: [not_null]
      - name: segment
        description: "Klantsegment: SMB / Mid-Market / Enterprise."
        tests:
          - accepted_values:
              values: ['SMB', 'Mid-Market', 'Enterprise']
```

Stap 8: prioriteren met MoSCoW

Je krijgt nooit alles in v1. Prioriteer met MoSCoW:

- **Must have** — zonder dit faalt het project.
- **Should have** — belangrijk maar niet kritiek voor v1.
- **Could have** — leuk om te hebben.
- **Won't have** — expliciet uit scope, nu niet.

Een eerste werkende versie ("MVP") moet de Must-haves dekken voor één business proces. Niet alles voor alle processen tegelijk — dat is een recept voor vertraging.

Source-to-target mapping (STTM)

Tussen logical model en implementatie hoort een Source-to-Target Mapping — per doelkolom: welke bron, welke transformatie, welke business rule. Vaak in Excel of een dbt-yaml, maar onmisbaar bij audits, onboarding van nieuwe engineers en bij debugging. Een minimaal voorbeeld:

```
 target_table: dim_customer
target_column: segment
source: salesforce.account.classification__c
transformation: |
  CASE
    WHEN classification__c IN ('SMB','Small') THEN 'SMB'
    WHEN classification__c = 'MidMarket'      THEN 'Mid-Market'
    WHEN classification__c = 'Enterprise'     THEN 'Enterprise'
    ELSE 'Unknown'
  END
business_rule: "Segment-mapping bevroren in 2024 door Sales-Ops."
data_steward: "alice@example.com"
last_review: "2026-04-01"
```

Veelvoorkomende valkuilen in de requirements-fase

- **"We willen alle data"** — een non-requirement. Dwing tot focus op concrete vragen en business processen.
- **Geen sponsor met beslissingsmandaat** — projecten zonder een directielid die knopen kan doorhakken modderen jarenlang voort.
- **Velddefinities die per afdeling verschillen** — wat is een "actieve klant"? Documenteer dit centraal voor je bouwt, niet erna.
- **Te veel processen tegelijk** — eerst sales, dan inkoop, dan voorraad. Niet drie tegelijk vanaf dag één.
- **Geen moment van "klaar"** — definieer per analytical question expliciet de acceptatiecriteria, anders blijven requirements eindeloos open.

Een ontwerpdocument-template

Vat alles samen in een 5-10 pagina's lang ontwerpdocument:

1. Doel en scope (wat wel, wat niet)
2. Stakeholders en rollen
3. Business processen en analytical questions
4. Niet-functionele requirements
5. Bus matrix
6. Conceptual model
7. Patroon-keuze met motivatie (Kimball / Inmon / DV / hybride)
8. Platformkeuze met motivatie (Snowflake / BigQuery / Synapse / Fabric)
9. Logical model — kerntabellen
10. Naming conventions
11. Implementatieplan en mijlpalen

> **Key takeaways**
>
> - Begin bij de business — vraag naar beslissingen, niet naar velden.
> - Niet-functionele eisen bepalen de platformkeuze, onderschat ze niet.
> - De bus matrix dwingt expliciete keuzes en vindt conformed dimensions.
> - Drie modelleerniveaus (conceptual / logical / physical) voorkomen verwarring.
> - Naming conventions zijn niet optioneel; leg ze vast voor je begint.
> - MoSCoW houdt scope onder controle. Build a slice, not a slab.

Veelgestelde vragen

Hoe verzamel je requirements voor een datawarehouse?

Begin bij de business met één-op-één-gesprekken. Stel een lijst van analytical questions op — concrete vragen die stakeholders willen beantwoorden. Deze vragen worden je acceptatiecriteria. Vraag naar beslissingen, niet naar velden.

Wat is een bus matrix?

Business processen op de rij-as, dimensies op de kolom-as, met vinkjes waar dimensies relevant zijn. Het laat zien welke dimensies conformed moeten worden gebouwd, en dwingt stakeholders tot expliciete scope.

Wat zijn functionele en niet-functionele requirements?

Functionele requirements beschrijven wat het warehouse moet kunnen. Niet-functionele requirements beschrijven hoe goed: data freshness, latency, concurrency, volume, security, compliance, beschikbaarheid en budget. Niet-functionele eisen bepalen de platformkeuze.

Wat is het verschil tussen conceptual, logical en physical model?

Conceptual model: alleen entiteiten en relaties in business-taal. Logical model: voegt attributen, datatypes en patroonkeuze toe. Physical model: concrete DDL voor je platform met partitioning, clustering en distribution keys.

Welke naming conventions gebruik je?

Lowercase met underscores. Prefixen: dim_, fact_, stg_, int_, hub_, sat_, link_. Surrogate keys op _key, business keys op _id. Booleans op is_/has_. Datums op _date of _at/_ts. Leg afspraken vast in NAMING.md.

Wat is MoSCoW-prioritering?

Must have (zonder dit faalt het), Should have (belangrijk niet kritiek), Could have (leuk om te hebben), Won't have (expliciet uit scope). Een MVP dekt de Must-haves voor één business proces — niet alles voor alle processen tegelijk.

Data Extract, Transform, Load (ETL)

De pipeline als product

22 min leestijd Beginner-Gevorderd

Vraag tien data engineers wat ETL is en je krijgt tien antwoorden. De afkorting staat voor **Extract, Transform, Load** — data uit bronnen halen, transformeren naar het juiste formaat, en laden in het warehouse. Klinkt eenvoudig. In de praktijk gaan hier de meeste projecten kapot, want de hoeken zitten in robuustheid, herstartbaarheid en kosten.

ETL versus ELT

De volgorde van de operaties is veranderd door de cloud. In de pre-cloud-era was compute schaars: je transformeerde data in een tussenstap (op een ETL-server), omdat het warehouse te duur was om die werkdruk te dragen. Met Snowflake, BigQuery en consorten heeft **ELT** de overhand genomen: laad de ruwe data eerst in het warehouse, en transformeer met SQL *in* het warehouse zelf. dbt is hier de standaardtool voor.

Aspect	ETL (traditioneel)	ELT (modern)
Transformatie	In ETL-engine (Informatica, SSIS)	In warehouse (SQL/dbt)
Tussenopslag	Vaak file-based / staging server	Stagingtabellen in warehouse
Schaal	Beperkt door ETL-server	Onbeperkt (warehouse compute)
Schema-on	Write	Vaak read (raw blijft ruw)
Tools	Informatica, Talend, SSIS	Fivetran/Airbyte + dbt
Kostenmodel	Licenties + servers	Pay-per-query

> **Wanneer kies je nog wel voor klassieke ETL?**
>
> ELT in het warehouse werkt geweldig zolang je ruwe data in het warehouse mág komen. Bij PII-zware bronnen waar pseudonimisering of tokenisatie verplicht is voordat data je analyse-omgeving raakt, moet je transformeren *vóór* het laden. Hetzelfde geldt voor compliance-trajecten met data residency-eisen, of bronnen waar de volumes te groot zijn om ruw te bewaren (telemetrie, IoT). In die gevallen is ETL — bijvoorbeeld met een streaming engine als Flink of Kafka Streams — gewoon de juiste keuze.

Extract: data uit de bron halen

De extract-stap kent drie hoofdstrategieën:

- **Full extract** — kopieer elke run de hele bron. Eenvoudig, maar duur en traag bij grote datasets.
- **Incremental extract** — haal alleen rijen op die zijn gewijzigd sinds de laatste run, op basis van een *watermark* (bijv. `updated_at`).
- **Change Data Capture (CDC)** — abonneer op de transactielog van de bron-database (bijv. via Debezium, AWS DMS, Fivetran). Detecteert ook deletes, wat watermark-gebaseerde extracts missen.

Een eenvoudige incremental extract in Python:

```
 import pandas as pd
from sqlalchemy import create_engine

src = create_engine("postgresql://user:pwd@source-db/app")
dwh = create_engine("snowflake://user:pwd@account/warehouse")

# Lees laatste watermark uit DWH metadata-tabel
with dwh.begin() as cx:
    last_ts = cx.execute(
        "SELECT MAX(load_ts) FROM meta.last_load WHERE table_name = 'orders'"
    ).scalar() or "1900-01-01"

# Haal alleen gewijzigde rijen op
query = f"""
    SELECT * FROM orders
    WHERE updated_at > '{last_ts}'
"""
df = pd.read_sql(query, src)

# Schrijf naar staging
df.to_sql("stg_orders", dwh, if_exists="append", index=False)

# Update watermark
with dwh.begin() as cx:
    cx.execute(
        "INSERT INTO meta.last_load VALUES ('orders', :ts)",
        {"ts": df["updated_at"].max()}
    )

print(f"{len(df)} rijen geladen, watermark naar {df['updated_at'].max()}")
```

> **Pas op met watermarks**
>
> Watermarks falen als bronrijen retroactief gewijzigd worden zonder `updated_at` te bumpen, of als de klok van de bron-database afwijkt. Geef je watermark altijd een kleine overlap (`WHERE`

`updated_at >= last_ts - INTERVAL '5 minutes'`) en zorg voor idempotentie aan de loadkant.

CDC met Debezium en Kafka — een minimaal opzet

Voor een betrouwbare CDC-pipeline op een PostgreSQL-bron leg je een Debezium-connector op de transactielog (WAL) en publiceer je de wijzigingen op een Kafka-topic. Een sink-connector schrijft elke wijziging naar een staging-tabel in het warehouse:

```
 # debezium-postgres-connector.yaml
name: orders-cdc
config:
  connector.class: io.debezium.connector.postgresql.PostgresConnector
  database.hostname: source-db.internal
  database.port: 5432
  database.user: cdc_reader
  database.dbname: app
  database.server.name: app_prod
  table.include.list: public.orders,public.customers
  plugin.name: pgoutput
  publication.name: dbz_app_pub
  slot.name: dbz_app_slot
  tombstones.on.delete: true
  topic.prefix: cdc
```

De CDC-events bevatten `op` ("c" = create, "u" = update, "d" = delete) en zowel de *before*- als *after*-state van elke rij. Daarmee kun je in silver een correcte SCD2-historie reconstrueren, ook bij deletes — iets wat een watermark-extract structureel mist.

Transform: opschonen en harmoniseren

De transform-laag doet meer dan formatteren. Typische taken:

- **Cleansing** — trimmen, casing, null-handling, datatype-conversies.
- **Deduplicatie** — late-arriving data, retries, dubbele extracts.
- **Joining / lookups** — bron-IDs vertalen naar surrogate keys.
- **Business rules** — een "actieve klant" is iemand met een order in laatste 90 dagen, etc.
- **Aggregations** — pre-aggregeren voor performance.
- **Versioning** — SCD Type 2 historie bouwen.

Hier een dbt-model dat orders dedupliceert en business logic toepast:

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

WITH source AS (
    SELECT * FROM {{ ref('stg_orders') }}
    {% if is_incremental() %}
      WHERE updated_at > (SELECT MAX(updated_at) FROM {{ this }})
    {% endif %}
),
deduped AS (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY updated_at DESC) AS
    FROM source
)
SELECT
    order_id,
    customer_id,
    order_date::DATE                    AS order_date,
    UPPER(TRIM(status))                 AS status,
    COALESCE(total_amount, 0)           AS total_amount,
    CASE WHEN total_amount > 1000 THEN 'high'
         WHEN total_amount > 100  THEN 'medium'
         ELSE 'low'
    END                                 AS order_size_tier,
    updated_at
FROM deduped
WHERE rn = 1
```

Load: idempotency is alles

Een load is **idempotent** als je hem twee keer kunt draaien zonder dat de uitkomst verandert. Bij een retry, een netwerkstoring of een herstart: de tabel moet er hetzelfde uit blijven zien als bij één geslaagde run.

Drie veelgebruikte loadstrategieën:

- **Append-only** — voeg nieuwe rijen toe, raak bestaande niet aan. Werkt voor immutable feiten (verkooptransacties).
- **Upsert / MERGE** — update als bestaand, anders insert. Voor mutable entiteiten (klanten, producten).
- **Full refresh** — drop en herlaad. Voor kleine dimensies of als de bron geen `updated_at` heeft.

De universele MERGE-statement (werkt in Snowflake, BigQuery, Redshift, Synapse):

```
 MERGE INTO dim_customer AS tgt
USING stg_customers AS src
   ON tgt.customer_id = src.customer_id

WHEN MATCHED AND (
       tgt.name      <> src.name
    OR tgt.segment   <> src.segment
    OR tgt.country   <> src.country
) THEN UPDATE SET
       name      = src.name,
       segment   = src.segment,
       country   = src.country,
       updated_at = CURRENT_TIMESTAMP

WHEN NOT MATCHED THEN INSERT (customer_id, name, segment, country, created_at
   VALUES (src.customer_id, src.name, src.segment, src.country, CURRENT_TIMES
```

De `WHEN MATCHED AND`-clausule is een belangrijk detail: zonder die check update je *elke* rij elke run, ook als er niets veranderd is. Dat geeft onnodige churn, lastigere debugging en hogere kosten.

Slowly Changing Dimensions in praktijk

Voor SCD Type 2 (historie bewaren) wordt het complexer. Hoofdstuk 5 gaat hier diep op in, maar als voorproefje:

```
 -- 1. Sluit verlopen versies
UPDATE dim_customer
SET    valid_to = CURRENT_DATE - 1,
       is_current = FALSE
FROM   stg_customers s
WHERE  dim_customer.customer_id = s.customer_id
  AND  dim_customer.is_current = TRUE
  AND  (dim_customer.segment <> s.segment OR dim_customer.country <> s.countr

-- 2. Voeg nieuwe versie toe
INSERT INTO dim_customer (customer_id, name, segment, country, valid_from, va
SELECT s.customer_id, s.name, s.segment, s.country, CURRENT_DATE, '9999-12-31
FROM   stg_customers s
LEFT JOIN dim_customer d
  ON d.customer_id = s.customer_id AND d.is_current = TRUE
WHERE  d.customer_id IS NULL
   OR  d.segment <> s.segment
   OR  d.country <> s.country;
```

Orkestratie

Je pipelines moeten ergens worden gestart, in de juiste volgorde, met dependencies. Populaire orkestrators:

- **Apache Airflow** — de standaard, Python-DAGs, breed ecosysteem.
- **Prefect / Dagster** — moderner, type-safe, betere developer experience.
- **Azure Data Factory** — visueel, native in Microsoft-stack.
- **dbt Cloud / Cron** — voor SQL-only transformaties.
- **GitHub Actions / Cloud Composer** — voor lichte workloads.

Een Airflow-DAG voor een dagelijkse load:

```
 from datetime import datetime
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.dbt.cloud.operators.dbt import DbtCloudRunJobOperator

with DAG(
    dag_id="dwh_daily_load",
    start_date=datetime(2026, 1, 1),
    schedule="0 4 * * *",          # elke nacht om 04:00
    catchup=False,
    max_active_runs=1,             # voorkom overlappende runs
    default_args={"retries": 2, "retry_delay": 300},
) as dag:

    extract_orders   = PythonOperator(task_id="extract_orders",   python_call
    extract_customers = PythonOperator(task_id="extract_customers", python_ca

    transform = DbtCloudRunJobOperator(task_id="dbt_build", job_id=12345)

    quality = PythonOperator(task_id="quality_checks", python_callable=run_ge

    [extract_orders, extract_customers] >> transform >> quality
```

Foutafhandeling die echt werkt

- **Retries met exponential backoff** — netwerkstoringen lossen zichzelf op.
- **Dead Letter Queue** — rijen die niet door validatie komen apart bewaren, niet weggooien.
- **Atomic loads** — laad in een staging-tabel, doe een atomic `SWAP` of rename. Geen halve states.
- **Alerting** — een gefaalde run om 04:00 mag niet pas om 09:00 ontdekt worden.
- **Re-runbaarheid** — kun je zonder repercussies een dag opnieuw laden?

Backfill: hoe je historie veilig opnieuw laadt

Vroeg of laat ontdek je een bug in je transformaties die het laatste kwartaal aan data corrumpeert. Of een bronsysteem doet retroactieve correcties op rijen van vorige maand. In beide gevallen heb je een betrouwbare backfill-strategie nodig:

1. **Parametriseer je pipeline op een datumbereik**, niet op "gisteren". Een DAG die `--start-date` en `--end-date` accepteert is herbruikbaar voor backfills, hercorrupted dagen en initial loads.
2. **Bewaar bronhistorie in bronze**. Als bronze append-only is, kun je elke transformatie deterministisch opnieuw draaien zonder de bron opnieuw te bevragen.
3. **Gebruik partitioning op load-datum** in silver/gold. Een backfill kan dan veilig één partitie overschrijven zonder de rest te raken — bijvoorbeeld via Snowflake's `DELETE` + `INSERT` in transactie, of BigQuery's partition-overwrite.
4. **Schaal compute tijdens de backfill omhoog**. Op Snowflake een grotere warehouse-grootte, op Databricks meer workers. Backfills zijn intensiever dan reguliere loads.

Performance-tips voor zware loads

- **Bulk in plaats van row-by-row** — een COPY INTO is 100-1000× sneller dan losse INSERT-statements.
- **Cluster keys / partitioning** — partitioneer fact tables op event-datum, niet op load-datum.
- **Vermijd UDFs in hot paths** — een Python-UDF in een SQL-statement breekt vectorisatie. Schrijf het als pure SQL waar het kan.
- **Monitor data skew** — een join op een kolom met 95% NULL-waarden gaat één worker overbelasten. Debug met `EXPLAIN`.
- **Materialise tussenresultaten** als ze door meerdere downstream models gebruikt worden — beter één keer berekenen dan tien keer.

Anti-patterns

- **"Truncate + reload"** in productie tijdens werkuren — zichtbare gaps voor gebruikers.
- **Hardcoded watermarks in SQL** — onmogelijk om historisch opnieuw te laden.
- **Geen logging** — als het pipeline-debug-uur slaat, weet je niets meer terug.
- **Eén grote DAG** die uren draait — maak hem modulair, met losse tabellen die je apart kunt herladen.
- **Business logic in extract-laag** — de extract moet ruwheid bewaren, transformatie hoort in silver.

Key takeaways

- ELT is de moderne standaard — laad ruw, transformeer met SQL/dbt.
- CDC vangt deletes; watermark-extracts niet. Kies bewust.
- Idempotentie is geen luxe; bouw alle loads zo dat ze veilig herstart kunnen worden.
- MERGE is je beste vriend voor mutable tabellen — met expliciete change-detection.
- Atomic swaps voorkomen halve states tijdens loads.
- Goede orkestratie maakt of breekt operationele rust.

Veelgestelde vragen

Wat is het verschil tussen ETL en ELT?

Bij ETL transformeer je data buiten het warehouse, op een aparte ETL-server, en laad je het resultaat. Bij ELT laad je de ruwe data direct in het warehouse en doe je de transformatie binnen het warehouse, meestal met SQL en dbt. ELT is de moderne standaard omdat cloud-warehouses als Snowflake en BigQuery enorm veel compute leveren tegen lage kosten.

Wat is Change Data Capture (CDC)?

Change Data Capture leest wijzigingen direct uit de transactielog van de bron-database (bijvoorbeeld via Debezium, AWS DMS of Fivetran). Het detecteert inserts, updates en deletes near-real-time en zet ze op een stream. Het grote voordeel ten opzichte van een watermark-extract: deletes worden ook gedetecteerd, en de bron wordt niet belast met queries.

Wat betekent idempotentie in een ETL-pipeline?

Een idempotente load is een load die je twee keer of vaker kunt draaien met exact dezelfde uitkomst. Bij retries, herstarts en netwerkstoringen blijft de doeltabel ongewijzigd na een geslaagde rerun. Praktisch betekent dit: gebruik MERGE in plaats van blinde INSERT, gebruik unique keys, en doe atomische swaps in plaats van TRUNCATE+INSERT.

Welke orkestratietool is het beste?

Apache Airflow is de defacto standaard met het breedste ecosysteem. Prefect en Dagster zijn moderner, type-safe en hebben betere developer experience. Azure Data Factory past goed in de Microsoft-stack. Voor SQL-only workloads is dbt Cloud of dbt + cron vaak voldoende.

Hoe voorkom je dubbele rijen bij retries?

Gebruik MERGE (UPSERT) met een unique key zoals een natural key of hash. Bij append-only tabellen voeg je een unieke run_id of dedupliceer je achteraf met ROW_NUMBER. Op streaming-pipelines werk je met exactly-once semantics via tools als Kafka Streams of Flink. Bouw retries en idempotentie in vanaf dag één.

Wanneer gebruik je full extract en wanneer incremental?

Full extract is geschikt voor kleine, statische bronnen (referentietabellen, metadata) en als de bron geen betrouwbaar updated_at-veld heeft. Incremental extract — via een watermark of CDC — gebruik je voor grote, vaak wijzigende tabellen om bandbreedte en kosten te besparen. Een hybride aanpak (wekelijks full, dagelijks incremental) lost vaak corruptie en gemiste deletes op.

Data Modelling Technieken

Het hart van het warehouse

24 min leestijd Gevorderd

Modellering is de plek waar slechte beslissingen jaren doorwerken. Een goede sterrenstructuur is leesbaar voor BI-developers, snel voor de query-engine en flexibel genoeg om mee te groeien met de business. Dit hoofdstuk gaat dieper dan de basis: niet alleen *wat* een fact en dimensie is, maar welke varianten er zijn en wanneer je ze inzet.

Star schema versus snowflake

In een **star schema** is elke dimensie één gedenormaliseerde tabel. In een **snowflake schema** worden dimensies verder genormaliseerd in subtabellen. Het verschil:

```sql
 -- Star: één dim_product
CREATE TABLE dim_product (
  product_key      INT PRIMARY KEY,
  product_id       VARCHAR(50),
  product_name     VARCHAR(200),
  category         VARCHAR(100),
  subcategory      VARCHAR(100),
  brand            VARCHAR(100),
  brand_country    VARCHAR(50)
);

-- Snowflake: dim_product genormaliseerd in subtabellen
CREATE TABLE dim_product (
  product_key      INT PRIMARY KEY,
  product_id       VARCHAR(50),
  product_name     VARCHAR(200),
  brand_key        INT,
  category_key     INT
);
CREATE TABLE dim_brand (
  brand_key        INT PRIMARY KEY,
  brand_name       VARCHAR(100),
  country_key      INT
);
CREATE TABLE dim_category (
  category_key     INT PRIMARY KEY,
  category_name    VARCHAR(100),
  subcategory_name VARCHAR(100)
);
```

De praktische regel: kies star, tenzij je een hele goede reden hebt om te snowflaken. Star is sneller voor query engines (minder joins), simpeler voor BI-tools en intuïtiever voor gebruikers. Snowflake bespaart marginaal opslag op enorme dimensies — maar in cloud-warehouses met columnstore-compressie is dat geen issue meer.

> **Wanneer is snowflake wel verdedigbaar?**
>
> Drie scenario's: (1) een dimensie heeft attributen die afzonderlijk een eigen analytische rol spelen — een productcategorie die ook gebruikers- of regelgevings-attributen heeft. (2) Outrigger-dimensies in SCD Type 5: snelwisselende attributen los van langzaam wisselende. (3) Een Inmon-stijl 3NF-laag onder een Kimball-goldlaag (zie hoofdstuk 2). Buiten deze gevallen creëer je vooral extra joins zonder waarde.

Fact tables: drie hoofdtypen

Niet elke fact-tabel is hetzelfde. Kimball onderscheidt drie typen, elk met eigen toepassingen:

1. Transaction fact

Eén rij per gebeurtenis (verkoop, klik, login). Meest voorkomende type. Granular en flexibel — je kunt altijd nog aggregeren, maar nooit detail terughalen dat je niet hebt opgeslagen.

```
CREATE TABLE fact_sales (
  sales_key       BIGINT IDENTITY,
  date_key        INT,
  customer_key    INT,
  product_key     INT,
  store_key       INT,
  order_id        VARCHAR(50),    -- degenerate dimension
  quantity        INT,
  unit_price      DECIMAL(10,2),
  discount        DECIMAL(10,2),
  net_amount      DECIMAL(12,2),
  load_ts         TIMESTAMP
);
```

2. Periodic snapshot fact

Eén rij per entiteit per periode (dagelijks saldo, maandelijkse voorraad). Geweldig voor trends en KPI's, maar grof.

```
CREATE TABLE fact_inventory_daily (
  date_key        INT,
  product_key     INT,
  store_key       INT,
  units_on_hand   INT,
  units_sold      INT,
  units_received  INT,
  inventory_value DECIMAL(12,2),
  PRIMARY KEY (date_key, product_key, store_key)
);
```

3. Accumulating snapshot fact

Eén rij per business proces dat door fasen gaat (een order van geplaatst naar geleverd). Datums voor elke mijlpaal — rij wordt geüpdatet als er fasen voorbij gaan.

```
CREATE TABLE fact_order_lifecycle (
  order_id          VARCHAR(50) PRIMARY KEY,
  customer_key      INT,
  date_placed_key   INT,
  date_paid_key     INT,
  date_shipped_key  INT,
  date_delivered_key INT,
  hours_to_pay       INT,
  hours_to_ship      INT,
  hours_to_deliver   INT,
  total_amount       DECIMAL(12,2)
);
```

Elke fase update dezelfde rij. Maakt cycle-time-analyses (gemiddeld 36 uur tot levering?) triviaal.

Additieve, semi-additieve en niet-additieve metingen

Niet elke meting in een fact-tabel kan op elke dimensie gesommeerd worden. Drie categorieën:

- **Additief** — werkt op alle dimensies. `quantity`, `net_amount`, `units_sold`. Som over tijd, klant, product, alles.
- **Semi-additief** — werkt op de meeste, maar niet op tijd. `units_on_hand` (voorraad), `account_balance` (saldo). Sommatie over klant of regio is correct, sommatie over tijd niet — daar moet je gemiddelde of laatste waarde nemen.
- **Niet-additief** — werkt op geen enkele dimensie. `unit_price`, `conversion_rate`, `profit_margin_pct`. Berekend vanuit additieve componenten op queryniveau.

Documenteer dit per fact-kolom. BI-developers die niet-additieve metingen optellen leveren rapportages die er goed uitzien én structureel fout zijn.

Slowly Changing Dimensions: 0 t/m 7

Klantgegevens veranderen. Hoe ga je daarmee om? Kimball benoemt zeven SCD-types:

Type	Strategie	Wanneer
0	Nooit wijzigen — bevries originele waarde	Onveranderlijke attributen (geboortedatum)
1	Overschrijven — geen historie	Correcties van fouten
2	Nieuwe rij per wijziging — volledige historie	Standaard voor business-relevante wijzigingen
3	Extra kolommen voor "previous" waarde	Beperkte historie (alleen vorige stand)
4	Mini-dimensie met snelwisselende attributen	Klanten met regelmatig wisselende segmenten
5	Combinatie van 1 en 4 met outrigger	Beperkte type-2-overhead nodig

Type	Strategie	Wanneer
6	Combinatie van 1, 2 en 3	Zowel huidige als historische analyses
7	Dual surrogate keys: huidig + historisch	"As-is" én "as-was" rapporteren

In de praktijk gebruik je 95% van de tijd **Type 1** of **Type 2**. De overige varianten zijn voor specifieke edge cases.

SCD Type 2 in detail

De volledige implementatie met effective dating:

```sql
CREATE TABLE dim_customer (
  customer_key     INT IDENTITY PRIMARY KEY,
  customer_id      VARCHAR(50),
  name             VARCHAR(200),
  email            VARCHAR(200),
  segment          VARCHAR(50),
  country          VARCHAR(50),
  valid_from       DATE,
  valid_to         DATE,
  is_current       BOOLEAN,
  hash_diff        CHAR(64)        -- voor change-detection
);

-- Trick: gebruik een hash om te detecteren of er iets is veranderd
-- in plaats van per kolom te vergelijken
WITH src AS (
  SELECT customer_id, name, email, segment, country,
         SHA2(CONCAT_WS('|', name, email, segment, country), 256) AS hash_dif
  FROM stg_customers
),
to_change AS (
  SELECT s.*
  FROM   src s
  JOIN   dim_customer d ON d.customer_id = s.customer_id AND d.is_current
  WHERE  d.hash_diff <> s.hash_diff
)
-- 1. Verlopen sluiten
UPDATE dim_customer
   SET valid_to = CURRENT_DATE - 1, is_current = FALSE
 WHERE customer_id IN (SELECT customer_id FROM to_change)
   AND is_current = TRUE;

-- 2. Nieuwe rijen
INSERT INTO dim_customer (customer_id, name, email, segment, country,
                          valid_from, valid_to, is_current, hash_diff)
SELECT customer_id, name, email, segment, country,
       CURRENT_DATE, '9999-12-31', TRUE, hash_diff
FROM   to_change

UNION ALL

-- 3. Volledig nieuwe klanten
SELECT s.customer_id, s.name, s.email, s.segment, s.country,
       CURRENT_DATE, '9999-12-31', TRUE, s.hash_diff
FROM   src s
```

```
LEFT   JOIN dim_customer d ON d.customer_id = s.customer_id
WHERE  d.customer_id IS NULL;
```

> 💡 **Hash-based change detection**
>
> In plaats van elke kolom expliciet te vergelijken, hash je alle SCD-2-attributen samen tot één `hash_diff`-kolom. Eén vergelijking en je weet of er iets veranderd is. Sneller, leesbaarder, minder bug-gevoelig.

Geavanceerde dimension-patronen

Junk dimension

Heb je een dozijn vlaggetjes en codes (is_promo, payment_method, channel) die niet de moeite waard zijn voor aparte dimensies? Stop ze in één *junk dimension* met alle mogelijke combinaties.

Role-playing dimension

Eén `dim_date`, maar meerdere keren joinen via aliassen (`order_date`, `ship_date`, `delivery_date`). Geen aparte tabellen nodig.

```
SELECT
    o.order_id,
    od.full_date AS order_date,
    sd.full_date AS ship_date,
    dd.full_date AS delivery_date
FROM   fact_order_lifecycle o
JOIN   dim_date od ON od.date_key = o.date_placed_key
JOIN   dim_date sd ON sd.date_key = o.date_shipped_key
JOIN   dim_date dd ON dd.date_key = o.date_delivered_key;
```

Bridge table (multi-valued)

Een patiënt heeft meerdere diagnoses, een product heeft meerdere kleuren. Een bridge-tabel met weight factors lost dit op zonder rijen te dubbeltellen.

```
CREATE TABLE bridge_product_color (
  product_key   INT,
  color_key     INT,
  weight_factor DECIMAL(5,4)    -- som per product = 1.0
);
```

Mini-dimensie

Snel-wisselende attributen (klant-credit-rating, marketing-segment) trekken SCD-2 explosies aan. Zet ze in een aparte mini-dimensie en koppel die direct aan de fact, niet aan de hoofd-dimensie.

Conformed dimensions

Een dimensie is *conformed* als hij identiek wordt gebruikt over meerdere fact tables. `dim_date` is altijd conformed. `dim_customer` hoort dat te zijn, maar wordt het vaak niet als verschillende teams hun eigen versie bouwen. Conformed dimensions zijn de enige manier om consistent te aggregeren over business processen heen.

Factless fact tables

Soms is "het feit dat iets gebeurde" zelf het feit, zonder meetwaarde. Een student die zich inschrijft voor een cursus, een klant die een nieuwsbrief opent. Een factless fact bevat alleen foreign keys, en wordt geteld in plaats van gesommeerd.

Surrogate keys versus natural keys

Bron-IDs (`customer_id = 'C-12345'`) lijken een goede primary key voor je dimensie. Tot een SAP-migratie de IDs hernummert, een fusie twee klantenbestanden samenvoegt, of een leverancier zijn productcodes uitbreidt van 6 naar 8 tekens. Surrogate keys — een synthetische integer of hash, alleen zinvol binnen het warehouse — beschermen je tegen al deze pijn.

Conventies die zich in de praktijk uitbetalen:

- Elke dimensie heeft een `{entity}_key` (surrogate, integer of hash) én een `{entity}_id` (de business key uit de bron, bewaard voor traceerbaarheid).
- Fact tables refereren altijd naar `_key` , nooit naar `_id` .
- Reserveer `-1` of `0` voor "Unknown" — een rij in elke dimensie met dummy-waarden zodat fact-rijen die nog geen match hebben toch aansluiting houden. Dit voorkomt dat outer joins data weggooien.

Late-arriving dimensies en facts

De werkelijkheid is wreed: soms komt een verkoop binnen voordat de bijbehorende klant in de klantdimensie staat. Of een SCD2-update arriveert dagen na de fact die ernaar verwijst. Twee patronen lossen dit op:

- **Late-arriving dimensies** — wanneer een fact-rij binnenkomt zonder dimensie-match, voeg je een placeholder-rij toe aan de dimensie met `customer_id` en NULL voor alle andere attributen. Wanneer de echte klantgegevens later arriveren, update je de placeholder (Type 1) of maak je de placeholder als eerste SCD2-versie en open je een tweede.
- **Late-arriving facts** — een verkoop die drie dagen later wordt gemeld moet gekoppeld worden aan de juiste SCD2-versie van de klant op het moment van verkoop, niet aan de huidige. Join op `customer_id` én op de `order_date` binnen `valid_from` en `valid_to` .

De grain — de heiligste regel

Voor je een fact-tabel ontwerpt: bepaal de **grain** — wat één rij voorstelt. "Eén rij per orderregel per dag." Schrijf het op. Mix nooit grains. Een fact die soms één rij per order is en soms één per orderregel is een tijdbom voor aggregaties.

Key takeaways

- Star > snowflake voor leesbaarheid en performance.
- Drie fact-types: transaction, periodic snapshot, accumulating snapshot — elk voor een ander doel.
- SCD Type 2 met hash-based change detection is de moderne standaard.
- Junk, role-playing, bridge en mini-dimensies lossen specifieke problemen op.
- Conformed dimensions maken cross-process analyses mogelijk.
- Definieer altijd je grain — en wijk er nooit van af.
- Documenteer of metingen additief, semi-additief of niet-additief zijn.

Veelgestelde vragen

Wat is het verschil tussen een star schema en een snowflake schema?

In een star schema is elke dimensie één gedenormaliseerde tabel die direct aan de fact-tabel hangt. In een snowflake schema worden dimensies verder genormaliseerd in subtabellen. Star is sneller voor query engines, simpeler voor BI-tools en de praktische standaard in moderne cloud-warehouses.

Wat is een Slowly Changing Dimension?

Een SCD is een dimensie waarvan de attributen in de loop van de tijd wijzigen. Kimball definieert zeven types die elk een andere strategie hebben: Type 1 overschrijft, Type 2 maakt een nieuwe rij per wijziging met valid_from/valid_to, Type 3 bewaart de vorige waarde in een extra kolom. In de praktijk gebruik je vooral Type 1 en Type 2.

Wat is een fact table?

Een fact table bevat metingen of gebeurtenissen — verkopen, klikken, voorraadstanden — met foreign keys naar dimensietabellen voor context. Er zijn drie hoofdtypen: transaction facts (één rij per gebeurtenis), periodic snapshot facts (één rij per entiteit per periode) en accumulating snapshot facts (één rij per business proces dat door fasen gaat).

Wat is een conformed dimension?

Een conformed dimension is een dimensie die identiek wordt gebruikt over meerdere fact tables — dim_date of dim_customer met dezelfde definities en surrogate keys, ongeacht of je hem joint met fact_sales of fact_inventory. Conformed dimensions zijn de enige manier om consistent te aggregeren over business processen heen.

Wat is de grain van een fact table?

De grain is wat één rij in de fact table voorstelt — bijvoorbeeld 'één rij per orderregel per dag'. Bepaal de grain vooraf, schrijf hem op en wijk er nooit van af. Een fact-tabel met gemixte grains maakt elke

aggregatie onbetrouwbaar.

Wanneer gebruik je een bridge table?

Een bridge table los je multi-valued dimensions mee op — situaties waar één entiteit meerdere waarden in een andere kan hebben (een patiënt met meerdere diagnoses, een product met meerdere kleuren). De bridge bevat de relatie plus een weight_factor zodat aggregaties niet dubbeltellen.

Data Kwaliteit en Governance

Waarom kwaliteit niet "later" kan

20 min leestijd Beginner-Gevorderd

Een datawarehouse waarin niemand vertrouwen heeft is een dure database die niemand gebruikt. Eén verkeerd dashboard in een bestuursvergadering kost meer geloofwaardigheid dan tien correcte rapporten verdienen. Kwaliteit en governance zijn geen "luxe" of "fase 2" — ze zijn de fundering.

De ware kosten van slechte datakwaliteit

IBM publiceerde de inmiddels vaak geciteerde schatting dat slechte datakwaliteit alleen al de Amerikaanse economie 3,1 biljoen dollar per jaar kost. Voor een individuele organisatie speelt het op drie niveaus:

- **Operationeel** — verkeerde voorraadbeslissingen, dubbele klantmailings, foute facturen, mislukte migraties.
- **Strategisch** — directie neemt beslissingen op basis van fout dashboards. Een investering van miljoenen op een rapport dat 15% afwijkt is duur.
- **Reputationeel** — als finance en sales verschillende omzetcijfers presenteren, verliest het hele team aan vertrouwen, ongeacht wie er gelijk heeft.

De ironie: kwaliteit voorkomen kost typisch 1× de moeite, kwaliteit detecteren 10×, kwaliteit corrigeren in productie 100×. Test vroeg en automatisch — dat is geen perfectionisme, dat is economie.

De zes dimensies van data quality

Data quality is meetbaar langs zes assen. Een test bedekt vrijwel altijd één van deze:

- **Completeness** — geen ongewenste NULL's. (`not_null`-test op verplichte velden.)
- **Uniqueness** — geen duplicaten waar dat niet mag. (`unique`-test op primary / business keys.)
- **Validity** — waarden voldoen aan formaat / domein. (E-mailpatroon, postcode, accepted values.)
- **Accuracy** — komt overeen met de werkelijkheid. (Moeilijker te testen — vereist vergelijking met externe bron.)
- **Consistency** — over tabellen heen kloppend. (Ordertotaal = som van orderregels.)
- **Timeliness** — actueel genoeg. (Data niet ouder dan X uur.)

Data profiling

Voor je tests kunt schrijven, moet je weten *wat* er in je data zit. Profiling is het systematisch onderzoeken van een dataset op patronen, distributies en uitschieters. Een eenvoudig profile in Python:

```python
 import pandas as pd

def profile(df: pd.DataFrame) -> pd.DataFrame:
    out = []
    for col in df.columns:
        s = df[col]
        out.append({
            "column":        col,
            "dtype":         str(s.dtype),
            "n_rows":        len(s),
            "n_null":        s.isna().sum(),
            "pct_null":      round(s.isna().mean() * 100, 2),
            "n_distinct":    s.nunique(),
            "min":           s.min() if s.dtype.kind in "biufM" else None,
            "max":           s.max() if s.dtype.kind in "biufM" else None,
            "sample":        s.dropna().head(3).tolist(),
        })
    return pd.DataFrame(out)

profile(orders).to_csv("orders_profile.csv", index=False)
```

Voor productie zijn er volwassen tools: **ydata-profiling** (voorheen pandas-profiling), **Great Expectations**, **Soda** of de ingebouwde profilers in dbt en DataHub.

Tests in dbt

Voor wie met dbt werkt: tests zijn declaratief in YAML, op kolomniveau:

```yaml
 version: 2
models:
  - name: dim_customer
    columns:
      - name: customer_key
        tests: [unique, not_null]
      - name: customer_id
        tests: [unique, not_null]
      - name: email
        tests:
          - not_null
          - dbt_utils.expression_is_true:
              expression: "REGEXP_LIKE(email, '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.
      - name: segment
        tests:
          - accepted_values:
              values: ['SMB', 'Mid-Market', 'Enterprise']
      - name: country
        tests:
          - relationships:
              to: ref('dim_country')
              field: country_code
```

Plus eigen SQL-tests onder `tests/` voor business rules:

```sql
 -- tests/orders_total_matches_lines.sql
-- Faalt als ordertotaal <> som van regels (afwijking > 0.01 EUR)
SELECT o.order_id,
       o.total_amount AS header_total,
       SUM(l.net_amount) AS line_total
FROM   {{ ref('fact_orders') }}   o
JOIN   {{ ref('fact_order_lines') }} l USING (order_id)
GROUP BY o.order_id, o.total_amount
HAVING ABS(o.total_amount - SUM(l.net_amount)) > 0.01
```

Great Expectations

Voor diepere validatie en statistische checks (drift in distributies, veranderend null-percentage) is Great Expectations populair:

```python
import great_expectations as gx

context = gx.get_context()
batch = context.sources.add_or_update_pandas("dwh") \
    .add_csv_asset("orders", "data/orders.csv") \
    .build_batch_request()

validator = context.get_validator(batch_request=batch)

validator.expect_column_values_to_not_be_null("order_id")
validator.expect_column_values_to_be_unique("order_id")
validator.expect_column_values_to_be_between("total_amount", min_value=0, max_
validator.expect_column_values_to_match_regex("email",
    r"^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}$")
validator.expect_column_proportion_of_unique_values_to_be_between("customer_i
    min_value=0.7, max_value=1.0)

results = validator.validate()
if not results.success:
    raise SystemExit(f"Quality gate gefaald: {results}")
```

Data contracts

Een **data contract** is een formele afspraak tussen producer en consumer over schema, kwaliteit, freshness en breaking changes. Verandert het bronsysteem een kolomtype? Het contract dwingt af dat dat aangekondigd en geverifieerd wordt voor het breekt.

```yaml
 # contracts/orders.yml
dataset: orders
owner: team-commerce
schema:
  - name: order_id
    type: string
    constraints: [not_null, unique]
  - name: customer_id
    type: string
    constraints: [not_null]
  - name: total_amount
    type: decimal(12,2)
    constraints: [not_null, ge: 0]
  - name: order_date
    type: date
    constraints: [not_null]
sla:
  freshness_max_hours: 6
  completeness_min_pct: 99.5
versioning:
  breaking_change_notice_days: 14
```

Tools die dit ondersteunen: **Great Expectations**, **Soda**, **Monte Carlo**, **Datafold**, **DataContract.com**. Of zelf bouwen op basis van YAML + CI-checks.

Data lineage

Lineage is het antwoord op "waar komt deze kolom vandaan?". Cruciaal voor:

- Impact analysis bij schema-changes
- GDPR-verzoeken (alle plekken waar persoonsgegevens van klant X staan)
- Debugging — een fout in gold tot bron herleiden
- Documentatie

dbt genereert automatisch lineage als je `{{ ref() }}` gebruikt. Voor enterprise-brede lineage (dwars door tools heen) zijn DataHub, OpenLineage, Atlan en Collibra de gangbare keuzes.

Master Data Management (MDM)

Wat als "klant" in CRM een andere identiteit heeft dan in ERP? MDM is het proces om één gouden record per entiteit op te bouwen, vaak via een combinatie van:

- **Deterministische matching** — exacte e-mail of IBAN
- **Probabilistische matching** — fuzzy match op naam + adres
- **Survivorship rules** — bij conflict, welke bron wint?

Een eenvoudige fuzzy match in Python:

```python
from rapidfuzz import fuzz

def match_score(a: dict, b: dict) -> float:
    name = fuzz.token_sort_ratio(a["name"], b["name"])
    addr = fuzz.token_sort_ratio(a["address"], b["address"])
    pc   = 100 if a["postal_code"] == b["postal_code"] else 0
    return 0.5 * name + 0.3 * addr + 0.2 * pc

# > 90: zekere match, 70-90: review, < 70: geen match
```

GDPR en datawarehouses

GDPR raakt elke DWH met persoonsgegevens. Drie kernverplichtingen:

- **Right to access** — alle data over één persoon kunnen exporteren. Lineage helpt enorm.
- **Right to be forgotten** — verwijderen, ook in backups en SCD-historie. Praktisch los je dit vaak op met *pseudonymisatie*: vervang PII door een hash, behoud analysebare data.
- **Purpose limitation** — data verzameld voor X mag niet voor Y gebruikt worden zonder grondslag.

Pseudonymisatie-pattern in SQL:

```sql
-- Persoonsgegevens in aparte tabel, pseudonym in fact / dim
CREATE TABLE pii_customer (
  pseudonym_id    CHAR(64) PRIMARY KEY,    -- SHA-256 van customer_id
  customer_id     VARCHAR(50),             -- echte ID, encrypted at rest
  full_name       VARCHAR(200),
  email           VARCHAR(200),
  phone           VARCHAR(50)
);

CREATE TABLE dim_customer (
  customer_key    INT PRIMARY KEY,
  pseudonym_id    CHAR(64),                -- alleen pseudoniem
  segment         VARCHAR(50),
  country         VARCHAR(50)
);
```

Met deze opzet kun je analytics doen zonder PII te raken. Voor "right to be forgotten" verwijder je alleen uit `pii_customer` — historische analyses blijven werken op het pseudoniem.

Governance: people, process, tech

Governance is geen tool maar een combinatie:

- **People** — data owners per dataset, data stewards per domein, een data council voor cross-domain beslissingen.

- **Process** — change management, schema-deprecation policy, security reviews bij nieuwe data.
- **Tech** — catalog (DataHub, Atlan), lineage, access control, masking.

> **Begin klein**
>
> Probeer geen enterprise-governance-platform op dag één. Begin met: een README per dataset, dbt-tests op kritische tabellen, één gedeelde slide met je naming conventions. Volwassenheid komt in stappen — niet via tooling-keuzes.

Data observability

Modernere stack: **data observability** automatiseert quality monitoring. Tools (Monte Carlo, Bigeye, Soda Cloud) detecteren via ML afwijkingen in volume, freshness en distributie zonder dat jij elke regel test schrijft. Handig op grote schaal, prijzig voor kleine teams.

Quality gates in CI/CD

Tests die niet automatisch draaien, draaien niet. Bouw quality gates in op drie momenten:

- **Pull request** — bij elke wijziging in dbt-modellen draait `dbt build --select state:modified+` tegen een test-omgeving. Failed tests blokkeren merge.
- **Pre-deploy** — voor elke productie-deploy draaien quality checks tegen sample-data of een snapshot uit productie.
- **Post-load** — na elke ELT-run draaien tests op de nieuwe data. Bij failure: alert + halt downstream consumers in plaats van garbage publiceren.

Een veelgebruikte aanpak: dbt's `error`-severity stopt de pipeline; `warn`-severity logt alleen. Reserveer `error` voor invarianten waar geen rapportage zin heeft als ze breken (unique keys op fact-tabellen, ordertotaal = som van regels). Zet `warn` op zachtere checks (volume binnen 10% van vorige run).

Documentatie als first-class citizen

Documentatie verouderd zodra hij in een aparte Confluence-pagina staat. Behandel hem als code:

- **Beschrijvingen in de schema.yml** naast je dbt-modellen — leeft mee met je code, gegenereerd in `dbt docs`.
- **One-line column descriptions** minimaal verplicht — een column zonder beschrijving is een failed PR.
- **Business-glossarium centraal** — wat is een "actieve klant", wat is "omzet", wat is een "campagne". Tools als DataHub, Atlan en Collibra koppelen termen aan kolommen.
- **Voorbeelden in de docs** — een rapport-vraag plus de SQL-oplossing helpt nieuwe gebruikers tien keer meer dan abstracte kolomdefinities.

> **Key takeaways**
>
> - Zes data-quality-dimensies: completeness, uniqueness, validity, accuracy, consistency, timeliness.

- Profile eerst, test daarna — anders test je op verkeerde aannames.
- dbt tests of Great Expectations dekken 80% van praktijktests.
- Data contracts maken schema-changes hanteerbaar.
- GDPR vereist pseudonymisatie of fysieke separatie van PII en analytics.
- Governance is people + process + tech — geen tool-aankoop.
- Quality gates in CI/CD voorkomen dat slechte data productie haalt.

Veelgestelde vragen

Wat zijn de zes dimensies van data quality?

Completeness, uniqueness, validity, accuracy, consistency en timeliness. Vrijwel elke data quality test bedekt één van deze zes dimensies.

Wat is een data contract?

Een formele afspraak tussen producer en consumer over schema, kwaliteit, freshness en breaking changes. Het dwingt aankondiging en verificatie af voor schema-wijzigingen. Tools: Great Expectations, Soda, Monte Carlo, Datafold.

Hoe ga je om met GDPR in een datawarehouse?

Right to access (lineage), right to be forgotten (pseudonymisatie of fysieke separatie van PII) en purpose limitation. Bewaar PII in een aparte tabel en koppel via een pseudoniem-hash zodat analytics blijven werken na verwijdering.

Wat is data lineage?

Het antwoord op 'waar komt deze kolom vandaan?'. Cruciaal voor impact analysis, GDPR-verzoeken en debugging. dbt genereert lineage automatisch via ref(); enterprise-breed gebruik je DataHub, OpenLineage, Atlan of Collibra.

Wat is Master Data Management?

Het proces om één gouden record per entiteit op te bouwen door deterministische matching (exacte e-mail of IBAN), probabilistische matching (fuzzy match) en survivorship rules te combineren wanneer dezelfde entiteit in meerdere bronsystemen staat.

Wat is data observability?

Geautomatiseerde quality monitoring met ML-modellen die afwijkingen in volume, freshness en distributies detecteren. Tools: Monte Carlo, Bigeye, Soda Cloud. Handig bij honderden tabellen, overkill voor kleine teams.

Cloud Datawarehouses

De cloud heeft het spel veranderd

22 min leestijd Beginner-Gevorderd

Tien jaar geleden kostte een serieus datawarehouse miljoenen aan hardware en licenties, met inkoopcycli van zes maanden. Vandaag tekent een startup een Snowflake-account op een dinsdagmiddag en draait nog dezelfde week productieworkloads. De cloud heeft drie dingen veranderd: storage en compute zijn los gekoppeld, schaal is elastisch, en betalen gaat per gebruik.

De vier grote spelers

Vier platforms domineren de moderne markt:

- **Snowflake** — multi-cloud, marktleider qua mindshare. Scheidt storage en compute strikt.
- **Google BigQuery** — serverless, slot-based, sterk geïntegreerd in GCP.
- **Amazon Redshift** — oudgediende, RA3-instances en serverless varianten.
- **Microsoft Fabric / Azure Synapse** — Microsoft's antwoord, geïntegreerd met Power BI en het Microsoft-ecosysteem.

Snowflake

Snowflake's **multi-cluster shared data architecture** is fundamenteel anders dan klassieke MPP. Drie lagen:

- **Storage layer** — micro-partitions van 50-500MB op cloud object storage (S3, ADLS, GCS), columnar-formatted en gecomprimeerd.
- **Compute layer** — virtual warehouses (clusters van VM's) die je naar believen aan- en uitzet. Meerdere warehouses kunnen dezelfde data tegelijk lezen zonder elkaar te hinderen.
- **Cloud services layer** — metadata, query parsing, transactiebeheer, security.

Een typische Snowflake-warehouse opzet:

```sql
 -- Klein warehouse voor BI dashboards
CREATE WAREHOUSE bi_wh
    WAREHOUSE_SIZE = 'X-SMALL'
    AUTO_SUSPEND   = 60                    -- pauzeer na 60 sec inactief
    AUTO_RESUME    = TRUE
    MIN_CLUSTER_COUNT = 1
    MAX_CLUSTER_COUNT = 4                  -- multi-cluster bij concurrency-p
    SCALING_POLICY = 'ECONOMY';

-- Groot warehouse voor zware ELT-jobs
CREATE WAREHOUSE etl_wh
    WAREHOUSE_SIZE = 'LARGE'
    AUTO_SUSPEND   = 300
    AUTO_RESUME    = TRUE;
```

Sterk in: ergonomie, multi-cloud, secure data sharing, time travel (queries op data van X dagen geleden zonder backups). **Minder in:** kosten lopen op zonder discipline, ML-tooling minder native dan Databricks.

Google BigQuery

BigQuery is volledig **serverless**: geen warehouses te kiezen, geen clusters te managen. Je betaalt voor data scanned (on-demand) of voor capaciteit in *slots* (flat-rate / editions).

```sql
 -- Partitioning op datum + clustering op customer_id
CREATE TABLE retail.fact_sales (
  sales_id      INT64,
  date          DATE,
  customer_id   STRING,
  product_id    STRING,
  amount        NUMERIC
)
PARTITION BY date
CLUSTER BY customer_id;

-- BigQuery rekent alleen de gescande kolommen + partitions
SELECT customer_id, SUM(amount) AS revenue
FROM   retail.fact_sales
WHERE  date BETWEEN '2026-01-01' AND '2026-03-31'
GROUP  BY customer_id;
```

Het pricing model dwingt discipline af: een `SELECT *` zonder partition filter scant je hele tabel en rekent dat af. Je leert al snel om *altijd* partition filters mee te geven.

Sterk in: schaalbaarheid (PB-niveau), ML via BigQuery ML, native streaming via Pub/Sub, geo-spatial.
Minder in: on-demand pricing kan onverwachts duur zijn, ecosystem buiten GCP minder ontwikkeld.

Amazon Redshift

Redshift is de oudste cloud-DWH (gelanceerd 2013). De moderne architectuur — **RA3 nodes** en **Redshift Serverless** — heeft de oude shared-nothing-MPP-architectuur grotendeels achter zich gelaten. Storage is nu losgekoppeld via **Redshift Managed Storage** (RMS).

```
 -- Klassieke distribution + sort keys (relevant in provisioned mode)
CREATE TABLE fact_sales (
  sales_id      BIGINT,
  date          DATE,
  customer_id   INT,
  product_id    INT,
  amount        DECIMAL(12,2)
)
DISTKEY (customer_id)         -- co-locatie van rijen per klant
SORTKEY (date);                -- snelle range queries op datum
```

Sterk in: AWS-integratie, Spectrum (federated queries op S3), kostprijs in vol gebruik. **Minder in:** dist/sort keys vereisen meer werk dan andere platforms — al neutraliseert serverless dat grotendeels.

Microsoft Fabric en Azure Synapse

Microsoft heeft de afgelopen jaren stevig opnieuw gepositioneerd. **Azure Synapse Analytics** bestaat nog (dedicated SQL pools, serverless SQL pools), maar de strategische focus ligt op **Microsoft Fabric** — een SaaS-platform dat OneLake (Delta-formaat lakehouse), Data Factory, Synapse Engineering en Power BI bundelt onder één licentie.

```
 -- Fabric Warehouse: T-SQL bovenop Delta in OneLake
CREATE TABLE fact_sales (
  sales_id      BIGINT,
  date          DATE,
  customer_id   INT,
  product_id    INT,
  amount        DECIMAL(12,2)
);

-- Synapse Serverless SQL Pool: query files in ADLS direct
SELECT TOP 100 *
FROM   OPENROWSET(
    BULK 'https://datalake.dfs.core.windows.net/raw/orders/*.parquet',
    FORMAT = 'PARQUET'
) AS r;
```

Sterk in: Power BI integratie (DirectLake), één capacity-licentie, ideaal voor Microsoft-stack organisaties.
Minder in: jonger product (Fabric), feature gaps, multi-cloud niet aanwezig.

Vergelijkingstabel

Eigenschap	Snowflake	BigQuery	Redshift	Fabric / Synapse
Cloud	AWS / Azure / GCP	GCP only	AWS only	Azure only
Compute model	Virtual warehouses	Serverless slots	Provisioned + Serverless	Capacity units
Storage	Eigen object storage	Colossus	RMS (managed)	OneLake (Delta)
Pricing	Per seconde compute	Per TB scanned of slots	Per node-hour of RPU	Per CU-hour
Time travel	Tot 90 dagen	7 dagen	Backups	Delta time travel
Data sharing	Native (Snowflake Marketplace)	Analytics Hub	Data Sharing	OneLake shortcuts
BI integratie	Universeel	Looker (best)	QuickSight (best)	Power BI (DirectLake)
ML-native	Snowpark ML	BigQuery ML	Redshift ML	Fabric Data Science

Kosten in de praktijk

Cloud DWH kosten komen ruwweg in drie categorieën:

- **Storage** — typisch $20-25/TB/maand, vergelijkbaar over platforms.
- **Compute** — varieert sterk per platform en gebruiksmodel. Bij Snowflake X-Small ~$2/uur, BigQuery on-demand ~$5/TB scanned.
- **Egress** — data eruit halen kan een verrassing zijn. Houd hier rekening mee bij multi-cloud.

Pricing-modellen vergeleken

Platform	Pricing-model	Voordeel	Risico
Snowflake	Per seconde compute (warehouse-uur)	Voorspelbaar bij gedisciplineerd auto-suspend	Idle warehouses verbranden geld; ad-hoc queries op grote warehouses
BigQuery on-demand	Per TB gescand	Betalen alleen voor wat je leest	Eén SELECT * zonder partition filter kan honderden euro's kosten
BigQuery editions	Slot-based (flat-rate)	Voorspelbaar bij stabiele workloads	Bij spikes kun je op slot-tekort lopen

Platform	Pricing-model	Voordeel	Risico
Redshift RA3	Per node-uur (provisioned)	Voorspelbaar bij 24/7 gebruik	Onderbenutting straft niet automatisch terug
Redshift Serverless	Per RPU-seconde	Geen idle-kosten	Cold start latency, monitoring lastiger
Fabric	Capacity Units (F-SKU)	Eén licentie voor heel het platform	Throttling bij overschrijding capacity

Een rekenvoorbeeld voor een mid-size mkb

Stel: 5 TB analytische data, dagelijkse ELT van 4 uur, 50 Power BI-gebruikers met 2 uur dashboard-traffic per dag. Indicatieve kosten per maand:

- **Snowflake** — storage 100 dollar, ELT op M-warehouse (4×30×16 ≈ 2.000 dollar), BI op XS multi-cluster (~600 dollar) = circa 2.700 dollar.
- **BigQuery edition Enterprise** — storage 100 dollar, 100 slots autoscale ≈ 2.500 dollar = circa 2.600 dollar.
- **Redshift Serverless** — storage 100 dollar, ELT en queries op 32 base RPUs ≈ 2.200 dollar = circa 2.300 dollar.
- **Microsoft Fabric F32** — alles inbegrepen onder één capacity ≈ 2.800 dollar (inclusief Power BI Pro-equivalent voor BI-gebruikers).

Disclaimer: dit zijn ruwe schattingen. Echte kosten hangen af van regio, kortingen, query-discipline en groei. Maar de orde van grootte ligt vrij dicht bij elkaar — de keuze maak je zelden op puur op prijs.

> **Reken altijd op je eigen data**
>
> Benchmarks van vendors zijn cherry-picked. Doe een eigen Proof-of-Concept met je *echte* queries op je *echte* data. De winnaar in een POC is bijna nooit de winnaar van de marketing-slide.

Hoe kies je?

Een pragmatische beslissingsboom:

1. **Microsoft-stack (Power BI, Azure)?** → Fabric. Bijna altijd het juiste antwoord puur op TCO en ergonomie.
2. **Volledig in GCP?** → BigQuery. Geen reden om buiten GCP te kijken.
3. **AWS-only en al Redshift in productie?** → Redshift Serverless als upgrade-pad.
4. **Multi-cloud, vrijheid belangrijk, toekomstbestendig?** → Snowflake.
5. **Veel ML naast BI, semi-structured data dominant?** → Snowflake of Databricks (lakehouse).

Multi-cloud strategie

Multi-cloud klinkt strategisch maar is duur. Twee patronen die wel werken:

- **Disaster recovery** — Snowflake-account in tweede cloud, replicatie via Snowflake's eigen replication.
- **Acquisition / merger** — overgenomen bedrijf draait in andere cloud, integreer pragmatisch via federated queries of Iceberg / OneLake shortcuts.

Vermijd "multi-cloud voor de zekerheid" — dat verdubbelt je operationele complexiteit zonder meetbaar voordeel.

Data sharing zonder kopiëren

Een sterke trend: data delen met externe partijen zonder fysiek te kopiëren. Elk platform heeft daar inmiddels een antwoord op:

- **Snowflake Secure Data Sharing** — verleen leesrechten op een database aan een ander Snowflake-account. Geen kopie, geen ETL, real-time. Snowflake Marketplace voor publieke datasets.
- **BigQuery Analytics Hub** — vergelijkbaar concept binnen GCP, met linked datasets en publishing-controles voor data providers.
- **Microsoft Fabric OneLake shortcuts** — virtuele referenties naar data in andere workspaces of zelfs externe ADLS-accounts. Eén kopie, meerdere lezers.
- **Databricks Delta Sharing** — open protocol over Delta-tabellen, werkt cross-platform en niet alleen binnen Databricks.

Dit verandert hoe organisaties data uitwisselen met partners, leveranciers en klanten. Een leverancier die dagelijkse voorraad publiceert, een marketingbureau dat campagne-resultaten deelt, een bank die geanonimiseerde benchmark-data aanbiedt — allemaal zonder het ETL-circus dat dit tien jaar geleden zou hebben gevergd.

Security en compliance op cloud DWH

Vier security-bouwstenen die je op elk platform tegenkomt:

- **Encryptie** — at-rest en in-transit standaard aan; voor compliance-zwaardere trajecten gebruik je customer-managed keys (CMK) die je zelf beheert in een KMS / Key Vault.
- **Network isolation** — Private Link / VPC-endpoints zorgen dat verkeer naar je warehouse niet over het publieke internet gaat. Verplicht voor de meeste enterprise compliance frameworks.
- **Row- en column-level security** — Snowflake heeft row access policies, BigQuery row-level security via authorized views, Synapse/Fabric dynamic data masking. Cruciaal voor multi-tenant analytics en GDPR-toepassingen.
- **Audit logging** — elke query, elke schema-wijziging, elke role-grant moet herleidbaar zijn. Vraag bij elke vendor naar retention en exportmogelijkheden naar je SIEM.

Voor Nederlandse organisaties speelt data residency vaak een rol. Snowflake heeft regio's in West Europe en Amsterdam; BigQuery heeft `europe-west4` (Eemshaven); Fabric draait op Azure Netherlands. Selecteer bewust en documenteer dit voor je DPIA.

De lakehouse-trend

De grenzen vervagen. Snowflake leest Iceberg / Delta direct, Fabric heeft OneLake, BigQuery heeft BigLake. De keuze "DWH of lakehouse" wordt minder relevant — je kunt steeds vaker Delta- of Iceberg-tabellen vanuit elke engine queryen. Voor data-engineers betekent dit: kies een open table format (Delta, Iceberg) en je opties blijven open.

Key takeaways

- De vier grote platforms zijn allemaal volwassen — er is geen "verkeerd" antwoord.
- Cloud-keuze (AWS / Azure / GCP) is meestal de eerste filter.
- Snowflake voor multi-cloud en ergonomie; Fabric voor Microsoft-shops; BigQuery voor GCP; Redshift bij bestaand AWS-investment.
- Doe POC's met eigen data — vendor benchmarks zijn niet representatief.
- Open table formats (Delta, Iceberg) verminderen vendor lock-in.
- Multi-cloud kost dubbel; gebruik het alleen met goede reden.

Veelgestelde vragen

Wat is het beste cloud datawarehouse in 2026?

Er is geen universeel 'beste' — de juiste keuze hangt af van je cloud-stack en use case. Microsoft-omgeving met Power BI: Fabric. GCP: BigQuery. AWS-only met bestaande investering: Redshift Serverless. Multi-cloud of strategische vrijheid: Snowflake. Doe altijd een POC met je eigen data.

Wat is het verschil tussen Snowflake en BigQuery?

Snowflake gebruikt virtual warehouses die je expliciet start, stopt en schaalt. BigQuery is volledig serverless met slot-based of on-demand pricing. Snowflake werkt op AWS, Azure én GCP; BigQuery alleen in GCP. Voor multi-cloud strategieën is Snowflake de logische keuze.

Wat is Microsoft Fabric en hoe verhoudt het zich tot Synapse?

Microsoft Fabric is het strategische SaaS-platform dat OneLake (Delta), Data Factory, Synapse Engineering en Power BI bundelt onder één capacity-licentie. Synapse bestaat nog, maar nieuwe investeringen gaan naar Fabric. Voor Microsoft-stack organisaties is Fabric meestal de juiste vervolgstap.

Wat kost een cloud datawarehouse?

Storage rond 20-25 dollar per TB per maand. Compute varieert sterk per gebruiksmodel. Een mid-size mkb met dagelijkse ELT en Power BI dashboards zit doorgaans op 1.500-5.000 euro per maand, ongeacht het gekozen platform.

Wat is het verschil tussen een datawarehouse en een lakehouse?

Een datawarehouse is geoptimaliseerd voor gestructureerde, analytische workloads met ACID-garanties. Een lakehouse combineert data lake-flexibiliteit (open formats als Delta of Iceberg, ML-tooling) met DWH-functionaliteit. De grenzen vervagen — Snowflake leest Iceberg, Fabric draait op Delta.

Moet ik kiezen tussen Snowflake en Databricks?

Snowflake is gestroomlijnder voor BI en analytische rapportages; Databricks is krachtiger voor ML, streaming en data engineering met Spark. Veel grote organisaties gebruiken beide — Databricks voor silver-engineering en ML, Snowflake voor de gold-laag en BI.

Performance Tuning en Optimalisatie

Performance is een kostenpost

21 min leestijd Gevorderd

In een cloud DWH zijn snelheid en kosten direct gekoppeld: een trage query die meer compute verbruikt is letterlijk een duurdere query. Dit hoofdstuk gaat over de zes hefbomen die in vrijwel elk modern warehouse werken — onafhankelijk van het platform.

1. Lees de query plan

Tunen zonder query plan is gokken. `EXPLAIN` (of het visuele equivalent in de UI) laat zien wat de engine echt doet. Wat te zoeken:

- **Full table scans** waar partitions zouden helpen
- **Cartesian joins** (geen join-condition)
- **Spills naar disk** bij sorteren of hashen
- **Pruning ratio** — Snowflake toont hoeveel micro-partitions overgeslagen zijn
- **Skew** — één node die alles doet terwijl andere wachten

```
 -- Snowflake: query profile + statistieken
EXPLAIN USING JSON
SELECT customer_id, SUM(amount)
FROM   fact_sales
WHERE  date_key BETWEEN 20260101 AND 20260331
GROUP  BY customer_id;

-- BigQuery: dry run kost niets en geeft scanned bytes
SELECT customer_id, SUM(amount)
FROM   `proj.retail.fact_sales`
WHERE  date BETWEEN '2026-01-01' AND '2026-03-31'
GROUP  BY customer_id;
-- Klik 'dry run' in console → "This query will process 2.3 GB"
```

2. Partitioning

Partitioning splitst een tabel fysiek in stukken op basis van een kolom (vaak datum). Queries met een filter op die kolom slaan irrelevante partitions over — *partition pruning*.

```sql
 -- BigQuery: time-unit partitioning
CREATE TABLE retail.fact_sales (
  sales_id INT64, date DATE, customer_id STRING, amount NUMERIC
)
PARTITION BY date
OPTIONS (require_partition_filter = TRUE);   -- dwingt filter af!

-- Snowflake: micro-partitions zijn automatisch (op insertvolgorde)
-- Voor expliciete clustering:
ALTER TABLE fact_sales CLUSTER BY (date_key);

-- Redshift: distribution + sort key in provisioned mode
CREATE TABLE fact_sales (...)
DISTKEY (customer_id)
SORTKEY (date);
```

De gouden regel: partitioneer op de kolom waarop bijna elke query filtert. Voor een DWH is dat 90% van de tijd een datumkolom.

> **Niet over-partitioneren**
>
> Eén partition per dag bij 5 jaar historie = 1.825 partitions. Prima. Eén partition per minuut = 2.6M partitions. Catastrofe — metadata-overhead overstijgt het voordeel. Houd partitions tussen ~50 MB en ~10 GB.

3. Clustering / sort keys

Waar partitioning grof is (per dag), is clustering fijnzinnig. Het sorteert rijen *binnen* partitions op één of meer kolommen, zodat lookups op die kolommen sneller worden.

```sql
 -- BigQuery: clustering tot 4 kolommen
CREATE TABLE retail.fact_sales (...)
PARTITION BY date
CLUSTER BY customer_id, product_id;

-- Snowflake: clustering key
ALTER TABLE fact_sales CLUSTER BY (date_key, customer_id);

-- Check Snowflake clustering depth (lager = beter)
SELECT SYSTEM$CLUSTERING_INFORMATION('fact_sales', '(date_key, customer_id)')
```

Clustering werkt het best op kolommen die je vaak in `WHERE`-filters of `JOIN`-condities gebruikt, en die niet super-laag-cardinaliteit hebben (een bool-kolom is geen goede cluster key).

4. Materialized views

Als dezelfde aggregatie tien keer per minuut gequeryed wordt, doe je het werk tien keer voor niets. Een materialized view bewaart het resultaat en updatet automatisch (of bijna).

```
 -- BigQuery: materialized view met smart refresh
CREATE MATERIALIZED VIEW retail.daily_revenue AS
SELECT date, SUM(amount) AS revenue, COUNT(*) AS orders
FROM   retail.fact_sales
GROUP  BY date;

-- Snowflake: materialized view (Enterprise edition)
CREATE MATERIALIZED VIEW daily_revenue AS
SELECT date_key, SUM(amount) AS revenue, COUNT(*) AS orders
FROM   fact_sales
GROUP  BY date_key;

-- Redshift: materialized view, manual refresh
CREATE MATERIALIZED VIEW daily_revenue AS
SELECT date, SUM(amount), COUNT(*)
FROM   fact_sales
GROUP  BY date;

REFRESH MATERIALIZED VIEW daily_revenue;
```

Maar pas op: materialized views zijn niet gratis. Elke base-table-update triggert (geheel of incrementeel) een refresh. Op snel veranderende tabellen kan de refresh-kost het query-voordeel opeten.

5. Caching

Elk modern DWH heeft meerdere cache-lagen:

- **Result cache** — identieke query in laatste 24 uur, gratis hit. (Snowflake, BigQuery, Synapse.)
- **Metadata cache** — query planning op recente tabel-info.
- **Local disk cache** — warehouse / slot heeft data nog op SSD.

Concrete impact: zorg dat dashboards *identieke* queries genereren (geen `NOW()`-injecties), zodat de result cache hit. BI-tools kunnen vaak parameters cachen — verifieer.

6. Concurrency tuning

Te weinig concurrency: gebruikers wachten op elkaar. Te veel: alles draait, niets is snel. De oplossing verschilt per platform:

- **Snowflake** — multi-cluster warehouse met auto-scaling op queue-tijd.
- **BigQuery** — slot reservations met workload management.
- **Redshift** — Workload Management (WLM) queues, Concurrency Scaling.

- **Synapse / Fabric** — resource classes / capacity-units.

```
-- Snowflake multi-cluster bij concurrency-pieken
CREATE WAREHOUSE bi_wh
  WAREHOUSE_SIZE = 'MEDIUM'
  MIN_CLUSTER_COUNT = 1
  MAX_CLUSTER_COUNT = 5            -- schaalt automatisch tot 5 clusters
  SCALING_POLICY = 'STANDARD';      -- 'ECONOMY' = zuiniger maar langzamer sc
```

SQL-patterns die je moet vermijden

- `SELECT *` — leest alle kolommen, breekt columnstore-voordeel.
- **Functies op join-kolommen** — `WHERE UPPER(name) = 'BOB'` sluit indexen uit.
- **Niet-sargable predicates** — `WHERE date_part('year', d) = 2026` in plaats van `WHERE d >= '2026-01-01'`.
- **Correlated subqueries** in plaats van window functions.
- **SELECT DISTINCT** als hack tegen duplicaten — los het oorzakelijk op.
- **Wide-open joins** zonder partition-filter op fact tables.

Voor en na

```
-- ❌ Trager
SELECT *
FROM   fact_sales
WHERE  EXTRACT(YEAR FROM date) = 2026
   AND UPPER(country) = 'NL';

-- ✅ Sneller
SELECT sales_id, customer_id, amount
FROM   fact_sales
WHERE  date >= '2026-01-01' AND date < '2027-01-01'
   AND country = 'NL';
```

Kostencontrole

De technische tuning bovenstaand levert direct besparingen op. Maar ook:

- **Resource monitors** — Snowflake suspend warehouses bij budgetoverschrijding.
- **Capacity caps** — BigQuery slot reservations begrenzen max-spend.
- **Auto-suspend** — Snowflake warehouses idle > 60s pauzeren.
- **Right-sizing** — een Medium die 30% draait is goedkoper dan een Large die 10% draait.
- **Storage tiering** — oude data naar archive / external (S3 Glacier, ADLS Cool).
- **Tabel-clean-up** — temp en oude staging tabellen kosten storage zonder waarde.

```
 -- Snowflake: resource monitor
CREATE RESOURCE MONITOR rm_etl
  WITH CREDIT_QUOTA = 1000
  TRIGGERS
    ON 75 PERCENT DO NOTIFY
    ON 90 PERCENT DO SUSPEND
    ON 100 PERCENT DO SUSPEND_IMMEDIATE;

ALTER WAREHOUSE etl_wh SET RESOURCE_MONITOR = rm_etl;
```

Specifieke patterns voor zware joins

Een join tussen een fact-tabel met miljarden rijen en een dimensie met miljoenen rijen is een klassieke performance-killer. Drie patronen die dit oplossen:

- **Broadcast join** — kleine dimensie wordt naar elke worker gerepliceerd, geen shuffle nodig. Snowflake en BigQuery doen dit automatisch onder een drempel; in Spark/Databricks expliciet via `broadcast()`.
- **Bucketing op join-key** — co-locatie van rijen met dezelfde join-key op dezelfde fysieke locatie. In BigQuery via clustering, in Redshift via DISTKEY, in Spark via bucketBy.
- **Pre-join in silver** — als dezelfde join in 20 gold-modellen voorkomt, doe hem één keer in silver en bewaar het resultaat. Materialiseer geen joins die niemand uitvoert; materialiseer wel joins die overal terugkomen.

Workload-isolatie

Eén warehouse voor alles is een klassieke valkuil. Een zware dbt-build zet een dashboard in de wachtrij; een ad-hoc data scientist trekt een hele cluster naar zich toe. Splits werklasten:

- **ELT-warehouse** — groter, draait op schema, mag spike's vertonen.
- **BI-warehouse** — kleiner met multi-cluster, prioriteit op responsiviteit, lage queue-tijd.
- **Ad-hoc / data science warehouse** — eigen budget, eigen monitor, geen impact op productie.

Op BigQuery los je dit op met aparte slot reservations; op Synapse/Fabric met aparte capacities. De extra kosten zijn klein vergeleken met de voorspelbare performance.

De 80/20-regel van performance

In de praktijk komt 80% van de winst uit:

1. Partitionering met enforced filter (`require_partition_filter = TRUE`).
2. Geen `SELECT *` — alleen kolommen die je echt gebruikt.
3. Query plans bekijken voor de top-10 zwaarste queries.
4. Right-sized warehouses / slot-reservations.
5. Auto-suspend op 60-300s.

Materialized views, custom clustering keys en advanced WLM-tuning zijn de overige 20%, en kosten meer onderhoud. Begin bij de basics.

Compaction en file-size tuning op lakehouses

Op Delta Lake, Iceberg en Hudi (Databricks, Microsoft Fabric, Snowflake Iceberg) is een aparte performance-overweging: **file size**. Streaming ingestion en frequente kleine commits creëren duizenden kleine bestanden, en kleine bestanden zijn ramp voor scan-performance.

- **Doel-bestandsgrootte**: typisch 128 MB tot 1 GB per Parquet-bestand. Onder 50 MB gaat performance hard achteruit.
- **Compaction / OPTIMIZE**: Delta's `OPTIMIZE`, Iceberg's `rewrite_data_files`, periodiek draaien — wekelijks op grote tabellen, dagelijks op streaming-fact-tabellen.
- **Z-Ordering / liquid clustering** (Delta) — herorganiseert data op meerdere kolommen tegelijk binnen bestanden voor multidimensionale skipping.
- **Vacuum / expire snapshots** — verwijdert oude versies die je niet meer voor time-travel nodig hebt; bespaart storage.

Benchmarken in productie

Tunen zonder meten is gokken. Monitor minimaal:

- P50/P95/P99 query latency per dashboard / pipeline
- Bytes scanned per query (BigQuery) of credits per query (Snowflake)
- Wachtrij-tijd voor warehouse / slot-reservering
- Top-10 duurste queries van de afgelopen 24 uur
- Storage-groei per schema

Key takeaways

- Performance en kosten zijn in cloud-DWH twee zijden van dezelfde medaille.
- Partitioneren op datum + verbieden van filter-loze queries is de grootste winst.
- Clustering is het tweede niveau — fijnzinniger dan partitioning.
- Materialized views alleen op stabiele aggregaties.
- Vermijd `SELECT *`, niet-sargable predicates en hidden cartesian joins.
- Auto-suspend, resource monitors en right-sizing besparen direct geld.
- Splits werklasten in eigen warehouses om elkaar niet te storen.

Veelgestelde vragen

Hoe optimaliseer je query performance?

Begin bij partitioning op datum met require_partition_filter aan. Voeg clustering toe op kolommen in joins en filters. Vermijd SELECT * en niet-sargable predicates. Lees query plans om de top-10 zwaarste

queries gericht te tunen.

Wat is het verschil tussen partitioning en clustering?

Partitioning splitst de tabel fysiek (vaak per datum) zodat partitions worden overgeslagen bij filter. Clustering sorteert rijen binnen partitions op extra kolommen voor snellere lookups. Een typische combinatie: partitioneer op datum, cluster op customer_id.

Wanneer gebruik je een materialized view?

Wanneer een aggregatie veel vaker gequeryed wordt dan dat de base table wijzigt. Bij snel wijzigende tabellen kan de refresh-kost het query-voordeel opeten — meet eerst, materialiseer dan.

Hoe houd je de kosten onder controle?

Auto-suspend op 60-300 sec, resource monitors met budget-cap, right-sizing, partition filters afdwingen, storage tiering voor oude data en temp-tabellen opruimen. Monitor de top-10 duurste queries dagelijks.

Wat zijn niet-sargable predicates?

Predicates die geen index of partition pruning kunnen gebruiken. WHERE EXTRACT(YEAR FROM date) = 2026 is niet-sargable; WHERE date >= '2026-01-01' AND date < '2027-01-01' wel. Functies of casts op filter-kolommen breken meestal performance.

Hoe lees je een query execution plan?

Zoek full table scans waar partitions zouden helpen, cartesian joins, spills naar disk en data skew. Snowflake toont pruning ratio, BigQuery scanned bytes. Gebruik de visuele query profilers van beide UI's.

Monitoring en Onderhoud

Observability voor data

18 min leestijd Gevorderd

Software-engineering heeft observability volwassen gemaakt: logs, metrics, traces. Data engineering loopt achter, maar haalt in. Een modern monitoring-stack voor een DWH dekt vier pijlers: **freshness**, **volume**, **schema** en **distributie**. Plus klassieke pipeline-orchestratie monitoring.

Disaster recovery: RPO en RTO voor data

Hoe lang mag je DWH stilliggen voordat het pijn doet? En hoeveel data mag je verliezen? Twee getallen die je expliciet moet hebben:

- **RPO (Recovery Point Objective)** — hoeveel data mag je verliezen in een ramp? RPO = 1 uur betekent dat je in het ergste geval het laatste uur opnieuw moet ingest, geen meer.
- **RTO (Recovery Time Objective)** — hoe snel moet je weer draaien na een storing? RTO = 4 uur betekent dat het systeem binnen 4 uur volledig hersteld moet zijn.

Cloud-warehouses bieden hier ingebouwde antwoorden. Snowflake's *time travel* (tot 90 dagen) en *fail-safe* (7 extra dagen) ondervangen de meeste data-corruptie scenario's. BigQuery heeft 7 dagen time travel. Microsoft Fabric en Delta Lake gebruiken Delta time travel. Dit alles dekt vooral logische rampen (verkeerde DELETE, foute deploy); voor regio-uitval heb je cross-region replicatie nodig — beschikbaar maar duur, alleen activeren als je RTO/RPO het vragen.

De vier pijlers van data observability

- **Freshness** — is de data recent genoeg? Wanneer is de laatste rij geladen?
- **Volume** — krijgen we het verwachte aantal rijen? Plotseling 0 rijen of 10x meer is verdacht.
- **Schema** — zijn er kolommen toegevoegd, verdwenen, of van type veranderd?
- **Distributie** — verandert het percentage NULL's, gemiddeldes, of categorieverdelingen plotseling?

Pillar 1: freshness monitoring

Het meest voorkomende incident is "data is oud" — een failed pipeline die niemand opmerkte. dbt heeft hier ingebouwde support:

```yaml
# sources.yml
sources:
  - name: app
    schema: raw_app
    tables:
      - name: orders
        loaded_at_field: updated_at
        freshness:
          warn_after:  {count: 6,  period: hour}
          error_after: {count: 12, period: hour}
      - name: customers
        loaded_at_field: updated_at
        freshness:
          warn_after:  {count: 24, period: hour}
          error_after: {count: 48, period: hour}
```

```sql
-- Run als onderdeel van CI/CD of orkestrator
-- $ dbt source freshness
-- Faalt als data te oud is, met een nette samenvatting
```

Voor non-dbt projecten een eenvoudig SQL-statement:

```sql
-- Geeft > 0 als de data te oud is
SELECT
    'orders' AS table_name,
    MAX(updated_at) AS last_loaded,
    DATEDIFF('hour', MAX(updated_at), CURRENT_TIMESTAMP) AS hours_old
FROM   raw_app.orders
HAVING hours_old > 6;
```

Pillar 2: volume monitoring

Verwacht aantal nieuwe rijen per dag verschilt per tabel — maar drift uit een baseline is altijd verdacht.

```sql
 -- Vergelijking met 7-daags rolling gemiddelde
WITH daily_loads AS (
  SELECT DATE_TRUNC('day', load_ts) AS load_date,
         COUNT(*) AS rows_loaded
  FROM   stg_orders
  GROUP  BY 1
),
baseline AS (
  SELECT load_date, rows_loaded,
         AVG(rows_loaded) OVER (
           ORDER BY load_date
           ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING
         ) AS avg_7d
  FROM   daily_loads
)
SELECT *
FROM   baseline
WHERE  load_date = CURRENT_DATE - 1
  AND  ABS(rows_loaded - avg_7d) / NULLIF(avg_7d, 0) > 0.5;    -- > 50% drift
```

Pillar 3: schema drift

Een producer zet stilletjes `order_total` om naar `order_total_eur` en je dashboard wijst nullen. Schema-drift detectie kan zo simpel als een snapshot zijn:

```sql
 -- Bewaar dagelijks de huidige schema's
INSERT INTO meta.schema_snapshots
SELECT CURRENT_DATE,
       table_schema,
       table_name,
       LISTAGG(column_name || ':' || data_type, ',') WITHIN GROUP (ORDER BY o
FROM   information_schema.columns
WHERE  table_schema NOT IN ('INFORMATION_SCHEMA','PG_CATALOG')
GROUP  BY 1, 2, 3;

-- Detecteer veranderingen ten opzichte van gisteren
SELECT today.table_schema, today.table_name,
       today.cols  AS today_cols,
       yest.cols   AS yest_cols
FROM   meta.schema_snapshots today
JOIN   meta.schema_snapshots yest
  ON   today.table_schema = yest.table_schema
 AND   today.table_name   = yest.table_name
 AND   yest.snapshot_date = today.snapshot_date - 1
WHERE  today.cols <> yest.cols;
```

Pillar 4: distribution drift

Stel: 95% van je orders heeft normaal status='completed'. Vandaag is het 60%. Je dataset is technisch in orde, maar er zit een upstream probleem.

```
 SELECT
    order_status,
    COUNT(*) * 1.0 / SUM(COUNT(*)) OVER () AS pct_today,
    LAG(COUNT(*) * 1.0 / SUM(COUNT(*)) OVER ()) OVER (ORDER BY snapshot_date)
FROM   fact_orders
WHERE  load_date IN (CURRENT_DATE, CURRENT_DATE - 1)
GROUP  BY snapshot_date, order_status;
```

Tools die dit automatiseren met ML-baselining: **Monte Carlo**, **Bigeye**, **Soda**, **Anomalo**. Voor lichtere setups: **Elementary** (open source dbt-package).

Pipeline / orchestrator monitoring

Buiten data zelf moeten ook je pipelines bewaakt worden:

- Run-status (success / failed / overdue)
- Run-duur drift (gisteren 10 min, vandaag 90 min?)
- Retry-aantallen
- Compute-kosten per run

Airflow biedt een eenvoudige Slack-callback bij fouten:

```
 from airflow.operators.python import PythonOperator
from urllib import request
import json, os

def slack_on_failure(context):
    payload = {
      "text": f":x: *{context['task_instance'].dag_id}* — task `{context['tas
              f"Run: {context['run_id']}\n"
              f"Log: {context['task_instance'].log_url}"
    }
    req = request.Request(
        os.environ["SLACK_WEBHOOK"],
        data=json.dumps(payload).encode(),
        headers={"Content-Type": "application/json"})
    request.urlopen(req)

# DAG default args
default_args = {
    "on_failure_callback": slack_on_failure,
    "retries": 2,
    "retry_delay": timedelta(minutes=5),
}
```

SLI's, SLO's en SLA's voor data

Net als webservices kunnen data-producten doelen krijgen:

- **SLI (Indicator)** — meetbaar getal: bijvoorbeeld "% queries onder 5 seconden".
- **SLO (Objective)** — interne doelstelling: bijvoorbeeld 99% < 5 sec.
- **SLA (Agreement)** — formele afspraak met afnemer met consequenties.

Voor een datawarehouse zijn typische SLI's:

- Freshness — % runs op tijd
- Completeness — % verwachte rijen geladen
- Query latency P95
- Dashboard beschikbaarheid

Alert design — niet schreeuwen

De grootste valkuil is alert fatigue: te veel meldingen, niemand kijkt nog. Regels:

- **Alert op symptoom, niet op oorzaak** — "data is > 12 uur oud" is bruikbaar; "Airflow task X failed" is meestal ruis.
- **Severity-tiers** — INFO (e-mail), WARN (Slack-channel), CRIT (PagerDuty).
- **Snooze / auto-resolve** — een alert die zichzelf herstelt moet zichzelf sluiten.

- **Eigenaarschap** — elke alert routeert naar een team, niet naar "@channel".

> **Test je alerts**
>
> Een alert die nog nooit gevuurd heeft, weet je niet of hij werkt. Disable af en toe een pipeline op de staging environment om te zien of de juiste mensen ge-paged worden. Geen surprise op zaterdagochtend.

Onderhoud: de stille killer

Een DWH dat niet onderhouden wordt rot weg. Plan periodiek:

- **Storage cleanup** — drop temp tabellen, ongebruikte tabellen, oude development schemas.
- **Statistics refresh** — Redshift `ANALYZE`, BigQuery automatisch, Snowflake automatisch.
- **Vacuum** — Redshift `VACUUM SORT` op grote tabellen.
- **Re-cluster** — Snowflake auto-clustering kostmonitoring.
- **Permission audits** — kwartaalreview wie nog toegang heeft.
- **Documentation review** — tabellen die niemand meer kent.

On-call rotatie en runbooks

Wanneer een DWH bedrijfskritisch wordt, hoort er een on-call rotatie bij. Twee elementen die het verschil maken tussen rustige en woelige weken:

- **Runbooks per veelvoorkomende alert** — een korte stap-voor-stap die de on-call door de eerste vijf minuten loodst. "Pipeline Y is failed: 1) check de log, 2) vergelijk met laatste succesvolle run, 3) ofwel rerun, ofwel pagina-wijzig de eigenaar." Geen lange architectuurstukken — operationele acties.
- **Heldere escalatiepaden** — wie wordt gewekt om 3:00 uur en bij welke severity? Documenteer dit in PagerDuty/Opsgenie zelf, niet alleen in Confluence. De on-call moet binnen 60 seconden weten wat te doen.

Capacity planning en groei-monitoring

Een DWH groeit. Wat vandaag past in een Medium-warehouse vraagt over een jaar een Large. Monitor minimaal:

- **Storage-groei per schema, week-over-week** — onverwachte spikes wijzen op een loop, log-tabel die explodeert of een ongepaste full-load.
- **Query-volume en concurrency over tijd** — als BI-gebruikers ineens drie keer zoveel queries draaien, plan capacity-uitbreiding voordat queues groeien.
- **Top-N gebruikers en pipelines op kosten** — de meeste kosten komen typisch van een handvol bronnen. Kennen wie ze zijn maakt fairness en optimalisatie mogelijk.
- **Onbenutte capacity** — overgedimensioneerde resources zijn verspilling. Snowflake auto-suspend, BigQuery flex slots, Redshift Serverless RPU-tuning kunnen vrijwel direct geld besparen.

Postmortems

Na elk significant incident: een blame-free postmortem met:

1. Tijdlijn (wanneer ontdekt, wanneer hersteld, hoe lang impact?)
2. Root cause analysis (de "5 whys")
3. Wat ging goed — herhaalbare reflexen
4. Wat ging slecht — verbeterpunten
5. Concrete actiepunten met eigenaar en deadline

Niet om te wijzen, maar om te leren. Een postmortem die voor de derde keer dezelfde root cause noteert verdient een actie op systeemniveau, niet weer een training.

Documentatie als onderhoud

Documentatie hoort bij onderhoud, niet bij features. Tools die het meegroeien afdwingen: dbt docs (auto-generated), DataHub, Atlan. Minimaal vereist per tabel:

- Owner
- Beschrijving in business-taal
- Bron(nen)
- Refresh-frequentie en SLA
- Bekende beperkingen

Key takeaways

- Vier pijlers: freshness, volume, schema, distributie. Dek ze allemaal.
- Alert op user-impact (data oud, dashboard kapot), niet op interne oorzaak.
- SLI's en SLO's geven data-producten dezelfde discipline als web-producten.
- Pipeline-monitoring zit naast data-monitoring — beide nodig.
- Postmortems zijn blame-free en eindigen met actiepunten met eigenaar.
- Documentatie en cleanup zijn onderdeel van onderhoud, niet "later".
- Capacity planning vóór de pijngrens, niet erna.

Veelgestelde vragen

Wat zijn de vier pijlers van data observability?

Freshness (recent genoeg?), volume (juiste aantal rijen?), schema (kolommen ongewijzigd?) en distributie (verandert NULL-percentage of categorieverdeling?). Plus klassieke pipeline-orchestratie monitoring.

Hoe monitor je data freshness?

dbt source freshness checks via loaded_at_field met warn_after en error_after thresholds. Voor non-dbt projecten: MAX(updated_at) tegen CURRENT_TIMESTAMP. Tools als Monte Carlo, Bigeye en Soda automatiseren met ML-baselines.

Wat is het verschil tussen SLI, SLO en SLA?

SLI is een meetbaar getal (% queries onder 5 sec), SLO is je interne doelstelling (99%), SLA is een formele afspraak met afnemers met consequenties. Voor data: freshness, completeness, query latency, dashboard beschikbaarheid.

Hoe voorkom je alert fatigue?

Alert op symptoom, niet op oorzaak. Severity-tiers: INFO/WARN/CRIT. Routeer naar specifieke teams. Test alerts periodiek. Een alert die nooit gevuurd heeft is onbekend kwantiteit.

Wat hoort bij periodiek onderhoud?

Storage cleanup, statistics refresh, VACUUM op Redshift, re-clustering monitoring, permission audits, documentation review. Op lakehouses ook compaction en vacuum/expire snapshots op Delta- of Iceberg-tabellen.

Wat is een postmortem?

Blame-free analyse na incident: tijdlijn, root cause (5 whys), wat ging goed, wat ging slecht, actiepunten met eigenaar. Doel: leren, niet wijzen. Terugkerende root cause vraagt systeemverbetering.

Case Study: Volledige Implementatie

RetailCo: een fictieve klant

30 min leestijd Alle niveaus

Genoeg theorie. In dit slothoofdstuk lopen we van begin tot eind door de bouw van een echt datawarehouse. RetailCo is fictief, maar de keuzes en problemen zijn ontleend aan echte projecten. We doen het in chronologische volgorde — niet zoals het mooi opgeschreven kan worden, maar zoals het op kantoor werkt.

1. Bedrijfscontext

RetailCo is een mid-market modeketen: 87 winkels in de Benelux, één e-commerce platform, ~120 medewerkers op het hoofdkantoor. Jaaromzet ~€85M. Bestaande BI: Excel-rapporten uit het ERP, met handmatige consolidatie naar e-commerce. Frustratie: cijfers kloppen niet over teams heen, en CEO ziet pas op woensdag de cijfers van vorige week.

Sponsor: CFO. Doel: dagelijkse, geconsolideerde KPI's voor sales, voorraad en marketing — beschikbaar om 08:00 op een Power BI dashboard.

2. Stakeholders en analytical questions

Na vier 1-op-1's komt deze prioriteitslijst:

- "Wat was de gisteromzet per winkel, vergeleken met vorige week en vorig jaar?"
- "Welke productcategorieën hebben de hoogste gross margin?"
- "Wat is de voorraadrotatie per artikel per winkel?"
- "Hoeveel procent van de e-commerce-bezoekers converteert?"
- "Wat is de gemiddelde klantwaarde (CLV) per acquisitiekanaal?"

3. Niet-functionele requirements

- Freshness: T-1 (gisterdata om 06:00 beschikbaar).
- Latency: BI dashboard < 5 sec per visual.
- Concurrency: piek 25 gelijktijdige BI-gebruikers, ~5 power users.
- Volume: ~1.2M order lines / dag, 5 jaar historie nodig.
- Budget: max €4.000/maand inclusief tooling.
- Security: rolgebaseerde toegang, PII gescheiden van analytics.
- Compliance: GDPR — right to be forgotten ondersteund.

4. Architectuurkeuze

Met "Power BI eindgebruikers, beperkt budget, mid-market schaal, T-1 freshness" wordt de keuze:

- **Warehouse:** Snowflake (X-Small + Medium warehouses). Past in budget, multi-cloud-veilig, eenvoudige operations.
- **Ingest:** Fivetran voor SaaS-bronnen (Shopify, Salesforce), Airbyte zelf-gehost voor on-prem ERP.
- **Transform:** dbt Core in GitHub + dbt Cloud Job Scheduler.
- **Orkestratie:** dbt Cloud (geen aparte Airflow nodig op deze schaal).
- **BI:** Power BI met DirectQuery / Import-mix.
- **Monitoring:** Elementary (open-source dbt-package) + Snowflake Resource Monitors.

5. Bronsystemen

Bron	Type	Connector	Refresh	Volume / dag
Microsoft Dynamics ERP	SQL Server on-prem	Airbyte (CDC)	Hourly	~1M rijen
Shopify (e-commerce)	SaaS API	Fivetran	Hourly	~50k rijen
Salesforce CRM	SaaS API	Fivetran	Daily	~5k rijen
Klaviyo (marketing)	SaaS API	Fivetran	Daily	~10k rijen
POS systeem	Daily CSV via SFTP	Custom Python	Daily 02:00	~200k rijen

6. Data model — bus matrix

Proces ↓ / Dimensie →	Datum	Klant	Product	Winkel	Kanaal	Promotie
Verkoop (POS + e-com)	✓	✓	✓	✓	✓	✓
Voorraad	✓		✓	✓		
Marketing	✓	✓			✓	✓
Inkoop	✓		✓			

7. Fysiek model — gold layer

```sql
 -- DIMENSIES
CREATE TABLE dim_date (
  date_key      INT PRIMARY KEY,
  full_date     DATE,
  year          INT,
  quarter       INT,
  month         INT,
  month_name    VARCHAR(20),
  week_of_year  INT,
  day_of_week   INT,
  weekday_name  VARCHAR(20),
  is_weekend    BOOLEAN,
  is_holiday    BOOLEAN
);

CREATE TABLE dim_customer (
  customer_key  INT IDENTITY PRIMARY KEY,
  pseudonym_id  CHAR(64),                  -- referentie naar pii_customer
  customer_id   VARCHAR(50),
  segment       VARCHAR(50),
  acq_channel   VARCHAR(50),
  country       VARCHAR(50),
  signup_date   DATE,
  valid_from    DATE,
  valid_to      DATE,
  is_current    BOOLEAN,
  hash_diff     CHAR(64)
);

CREATE TABLE dim_product (
  product_key   INT IDENTITY PRIMARY KEY,
  product_id    VARCHAR(50),
  sku           VARCHAR(50),
  name          VARCHAR(200),
  category      VARCHAR(100),
  subcategory   VARCHAR(100),
  brand         VARCHAR(100),
  unit_cost_eur DECIMAL(10,2),
  unit_price_eur DECIMAL(10,2),
  valid_from    DATE,
  valid_to      DATE,
  is_current    BOOLEAN,
  hash_diff     CHAR(64)
);

CREATE TABLE dim_store (
```

```
  store_key     INT IDENTITY PRIMARY KEY,
  store_id      VARCHAR(20),
  store_name    VARCHAR(100),
  region        VARCHAR(50),
  country       VARCHAR(50),
  m2            INT,
  opened_date   DATE,
  is_active     BOOLEAN
);

CREATE TABLE dim_channel (
  channel_key   INT PRIMARY KEY,
  channel_name  VARCHAR(50)         -- 'POS', 'Web', 'App', 'Marketplace'
);

-- FACTS
CREATE TABLE fact_sales (
  sales_key     BIGINT IDENTITY,
  date_key      INT,
  customer_key  INT,
  product_key   INT,
  store_key     INT,
  channel_key   INT,
  order_id      VARCHAR(50),        -- degenerate dim
  quantity      INT,
  unit_price    DECIMAL(10,2),
  unit_cost     DECIMAL(10,2),
  discount      DECIMAL(10,2),
  net_amount    DECIMAL(12,2),
  gross_margin  DECIMAL(12,2),
  load_ts       TIMESTAMP
)
CLUSTER BY (date_key);

CREATE TABLE fact_inventory_daily (
  date_key      INT,
  product_key   INT,
  store_key     INT,
  units_on_hand INT,
  units_received INT,
  units_sold    INT,
  inventory_value_eur DECIMAL(12,2),
  PRIMARY KEY (date_key, product_key, store_key)
);
```

8. dbt project structuur

```
retailco_dwh/
├── dbt_project.yml
├── models/
│   ├── staging/
│   │   ├── erp/
│   │   │   ├── stg_erp__sales_lines.sql
│   │   │   └── stg_erp__inventory.sql
│   │   ├── shopify/
│   │   │   ├── stg_shopify__orders.sql
│   │   │   └── stg_shopify__customers.sql
│   │   └── salesforce/
│   │       └── stg_salesforce__accounts.sql
│   ├── intermediate/
│   │   ├── int_sales_unioned.sql        -- POS + e-com
│   │   └── int_customers_mastered.sql   -- MDM matching
│   └── marts/
│       ├── dim_date.sql
│       ├── dim_customer.sql              -- SCD Type 2
│       ├── dim_product.sql
│       ├── dim_store.sql
│       ├── dim_channel.sql
│       ├── fact_sales.sql
│       └── fact_inventory_daily.sql
├── tests/
│   ├── assert_sales_total_matches_lines.sql
│   └── assert_no_negative_quantities.sql
└── macros/
    └── generate_surrogate_key.sql
```

9. Voorbeeldmodel — fact_sales

```sql
-- models/marts/fact_sales.sql
{{ config(
    materialized='incremental',
    incremental_strategy='merge',
    unique_key='sales_key',
    cluster_by=['date_key']
) }}

WITH s AS (
    SELECT *
    FROM   {{ ref('int_sales_unioned') }}
    {% if is_incremental() %}
      WHERE load_ts > (SELECT MAX(load_ts) FROM {{ this }})
    {% endif %}
)
SELECT
    {{ dbt_utils.generate_surrogate_key(['s.order_id', 's.line_no']) }} AS sa
    TO_NUMBER(TO_CHAR(s.order_date, 'YYYYMMDD'))     AS date_key,
    c.customer_key,
    p.product_key,
    st.store_key,
    ch.channel_key,
    s.order_id,
    s.quantity,
    s.unit_price,
    p.unit_cost_eur                                  AS unit_cost,
    s.discount,
    s.quantity * (s.unit_price - s.discount)         AS net_amount,
    s.quantity * (s.unit_price - p.unit_cost_eur)    AS gross_margin,
    CURRENT_TIMESTAMP                                AS load_ts
FROM   s
LEFT JOIN {{ ref('dim_customer') }} c
       ON s.customer_id = c.customer_id AND c.is_current
LEFT JOIN {{ ref('dim_product') }} p
       ON s.product_id  = p.product_id  AND p.is_current
LEFT JOIN {{ ref('dim_store') }} st
       ON s.store_id    = st.store_id
LEFT JOIN {{ ref('dim_channel') }} ch
       ON s.channel     = ch.channel_name
```

10. Quality tests

```yaml
 version: 2
models:
  - name: fact_sales
    columns:
      - name: sales_key
        tests: [unique, not_null]
      - name: date_key
        tests:
          - not_null
          - relationships: { to: ref('dim_date'), field: date_key }
      - name: customer_key
        tests:
          - relationships: { to: ref('dim_customer'), field: customer_key }
      - name: net_amount
        tests:
          - dbt_utils.expression_is_true: { expression: ">= 0" }
      - name: gross_margin
        tests:
          - dbt_utils.expression_is_true: { expression: "<= net_amount" }

  - name: dim_customer
    columns:
      - name: customer_key
        tests: [unique, not_null]
      - name: customer_id
        tests:
          - dbt_utils.unique_combination_of_columns:
              combination_of_columns: [customer_id, valid_from]

sources:
  - name: erp
    schema: raw_erp
    freshness: { warn_after: { count: 6, period: hour }, error_after: { count
    tables:
      - name: sales_lines
        loaded_at_field: _loaded_at
```

11. Pipeline-schema

```
 -- dbt Cloud Job: 'daily_full_build'
-- Schedule: 0 4 * * *  (04:00 UTC)
--
-- Stappen:
--   1. dbt source freshness  → fail-fast als bron-data te oud is
--   2. dbt run --select staging       (alle stg_ modellen)
--   3. dbt run --select intermediate
--   4. dbt run --select marts
--   5. dbt test                       (alle quality tests)
--   6. dbt docs generate              (auto-deploy)
--   7. Slack notify on success/failure
```

12. Voorbeeld dashboard query

```
 -- Dagelijkse sales per winkel met YoY-vergelijking
WITH this_year AS (
    SELECT s.store_key, st.store_name, st.region,
           SUM(f.net_amount) AS revenue_ytd,
           SUM(f.gross_margin) AS margin_ytd
    FROM   fact_sales f
    JOIN   dim_store st ON st.store_key = f.store_key
    WHERE  f.date_key BETWEEN 20260101 AND TO_NUMBER(TO_CHAR(CURRENT_DATE, 'Y
    GROUP  BY 1, 2, 3
),
last_year AS (
    SELECT s.store_key,
           SUM(f.net_amount) AS revenue_ly
    FROM   fact_sales f
    WHERE  f.date_key BETWEEN 20250101 AND TO_NUMBER(TO_CHAR(DATEADD(YEAR, -1
    GROUP  BY 1
)
SELECT t.store_name, t.region, t.revenue_ytd, t.margin_ytd,
       l.revenue_ly,
       (t.revenue_ytd - l.revenue_ly) / NULLIF(l.revenue_ly, 0) AS yoy_growth
FROM   this_year t
LEFT JOIN last_year l ON l.store_key = t.store_key
ORDER  BY t.revenue_ytd DESC;
```

13. Wat ging mis (en wat we leerden)

Geen project zonder hobbels. Drie incidenten uit RetailCo's eerste jaar:

- **Maand 2:** POS-CSV bevat soms negatieve quantities (returns) — onze test brak. Oplossing: returns expliciet als aparte transactie modelleren, niet als negatieve verkoop.

- **Maand 4:** Shopify hernoemde een veld in de API — onze stg_-laag negeerde stille schema-drift. Oplossing: schema-drift-detectie ingevoerd (zie hoofdstuk 9), met dagelijks snapshot.
- **Maand 7:** Snowflake-rekening verdubbelde door één junior-analyst die `SELECT * FROM fact_sales` in een dashboard-filter gebruikte. Oplossing: queryreviews op power-user-rollen, plus resource monitor met hard limit.

14. Resultaten na 9 maanden

- Dagelijks dashboard om 07:00 voor de directie — geen handmatige consolidatie meer.
- 26 dbt-modellen, 78 tests, 99.4% on-time runs.
- Maandelijkse Snowflake-kosten: €1.840 (binnen budget).
- Twee analisten kunnen nu zelfstandig nieuwe analyses bouwen — voorheen tickets bij IT.
- CEO citaat: "Eindelijk één getal voor omzet."

15. Wat we de volgende keer anders zouden doen

- Vroeger investeren in MDM voor klant — pas in maand 5 begonnen, retroactief lastig.
- PII-scheiding vanaf dag één in plaats van bij eerste GDPR-vraag in maand 6.
- Power BI dataset-grootte beperken — DirectQuery-mix had eerder ingeregeld kunnen worden.
- Documentatie als deliverable per sprint, niet als slotklus.

> **🏁 Slotwoord**
>
> Een datawarehouse bouwen is geen technisch project — het is een project waarin techniek het slotsom is van duidelijke afspraken, gedeelde definities en operationele discipline. Elk hoofdstuk van dit ebook raakt een hoek van die discipline. De magie zit niet in een specifieke tool of patroon, maar in de combinatie: weten wat je business vraagt, kiezen wat past, uitvoeren met aandacht voor kwaliteit, en blijven onderhouden.
>
> Veel succes met je eigen warehouse — en denk eraan: de eerste versie is altijd de slechtste. Bouw, leer, verbeter.

> **🤝 Verder met DataPartner365**
>
> Dit ebook is geschreven om vrij beschikbaar te zijn. Heb je hulp nodig bij een echt project — van architectuurkeuze tot implementatie of doorontwikkeling? Neem contact op voor een vrijblijvend gesprek.
>
> Contact opnemen · Lees de blog · Andere leerpaden