DB3: Datawarehousing

Van Overveldt Jan

Ven Johan

De Reys Tom

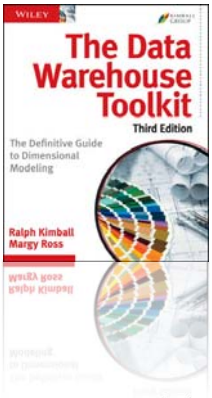
Karel de Grote Hogeschool

Cursusmateriaal

Slides

Blackboard

Aanbevolen literatuur



27-9-

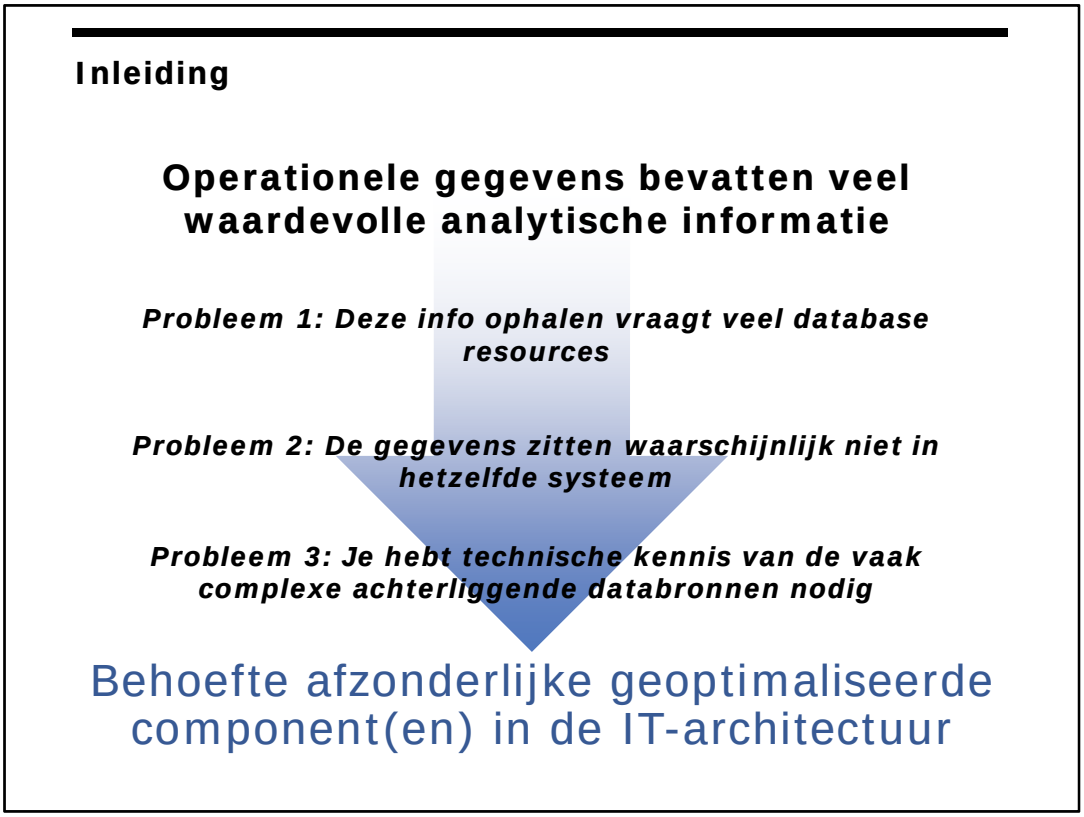
Gebruik deze slides als leidraad door de leerstof. Op blackboard is ook een tutorial beschikbaar waarmee je zelf een eenvoudig datawarehouse zal opzetten.

The Data Warehouse Toolkit is een vlot leesbaar boek met duidelijke voorbeelden. De meeste leerstof bij dit opleidingsonderdeel is terug te vinden in de eerste 4 hoofdstukken.

Inhoud

1. **Inleiding**
2. Ontwerp van een datawarehouse
3. Architectuur

27-9-



Voorbeeld: *Een analyseteam wil het verband tussen klachten van klanten en het opzeggen van een abonnement onderzoeken.*
De abonnementen worden beheerd in het centrale operationele systeem. De klachtenafhandeling zit in een afzonderlijk Customer Relationship Management systeem.
Probleem 1: Een dergelijke query uitvoeren op het operationele systeem zorgt voor zware reads op de databank. We moeten immers van alle klanten zoeken naar het verband tussen klachten en opzeggingen. De analysequery zou bijgevolg het operationele systeem helemaal kunnen lam leggen.
Bij het ontwerp van een operationeel systeem werden indexen volledig afgestemd op heel gerichte en beperkte inserts en reads ("Toon de gegevens van klant Jos Vermeulen" of "annuleer het abonnement van Jos Vermeulen"). Analysequeries verwerken echter veel meer data en de joins zullen vaak op andere plaatsen gelegd worden. Hierdoor zullen analysequeries vaak weinig performant worden uitgevoerd op een operationeel systeem. Aanpassen van het operationele systeem om zo de analysequeries te kunnen optimaliseren heeft dan weer een nefaste impact op de performantie van het operationele systeem.

Probleem 2: Het is onmogelijk om één query uit te voeren omdat we gegevens uit beide systemen moeten combineren. Het zal dus een intensieve job zijn waarbij extracts uit beide systemen handmatig (vb. via Excel) moeten gecombineerd worden. Eén keer is dat acceptabel. Maar als dergelijke analyses vaak voorkomen moet er een structurelere oplossing voorzien worden.

Probleem 3: Als de query dan toch op één systeem zou kunnen uitgevoerd worden, dan nog is het voor data analysten niet vanzelfsprekend om de juiste queries te bouwen. De complexiteit van databases in operationele systemen is vaak zeer groot, veldnamen hebben vaak een technische en/of onduidelijke benaming en achter de inhoud zitten vaak bedrijfsregels verscholen die essentieel zijn voor een correcte interpretatie. Door het bouwen van een afzonderlijk systeem moet dit allemaal onderzocht worden door een team. In het afzonderlijke systeem zouden de data analysten een ondubbelzinnig beeld moeten hebben op de beschikbare gegevens.

Vb. een veld net_time is een veld waar de netto besteedde tijd staat ingevuld voor een reparatie van electronica aan huis. Het veld blijkt vaak niet ingevuld te zijn. Een data analyst kan dat interpreteren als:
- De besteedde tijd is niet gekend
- De netto besteedde tijd is 0

Daarnaast weet de data analyst evenmin wat de werkelijke betekenis is van 'net'. Het zou kunnen betekenen 'exclusief verplaatsing' maar daarvoor heeft hij meer informatie nodig.

In een gezonde data architectuur wordt voor analysedoeleinden een afzonderlijke component voorzien die helemaal is geoptimaliseerd voor deze doeleinden waarbij de opgesomde problemen worden weggewerkt.

Definitie: Datawarehouse

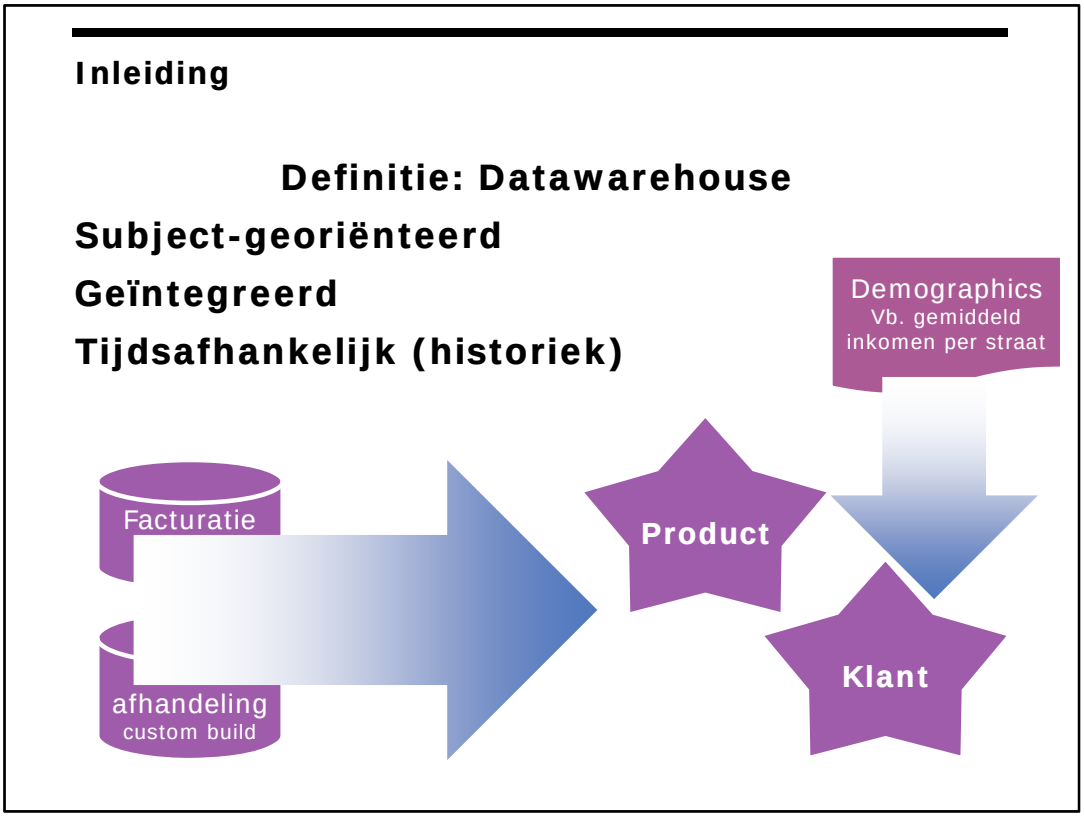
INMON

A data warehouse is a subject oriented, integrated, time-variant and non-volatile collection of data, designed to support the decision making process

KIMBALL

A data warehouse is the only queryable source of data in the enterprise. It's based on a frequently updated copy of transactional data, and it's specifically structured for query and analysis.

Begin jaren '90 woedde er een echte strijd tussen deze twee kampen. Als je "Kimball Inmon" ingeeft in google dan zal je merken dat hier nog veel over geschreven wordt. Hoewel we verder in de cursus voornamelijk "Kimball" zullen volgen is naar mijn mening de definitie van INMON de betere. Ze is minder op het technische aspect gefocust en de definitie doorstaat ook beter de tand des tijds.



- Subject georiënteerd: De organisatie van je data gebeurd volgens onderwerpen waarrond analysevragen kunnen gesteld worden. Hierboven zie je voorbeelden van typische onderwerpen zoals "Klant" en "Product". Binnen zo'n subject area is het de bedoeling zo veel mogelijk informatie waar het bedrijf over beschikt te gebruiken.

- Geïntegreerd: Hiermee wordt bedoeld dat je in een datawarehouse verschillende bronnen gaat gebruiken. Dit kunnen zowel interne (Facturatie en klachtafhandeling zitten binnen verschillende interne systemen) als externe (Demografische informatie van een overheidsinstelling) gegevensbronnen zijn.

- Tijdsafhankelijk: Je verwerkt in je datawarehouse ook historiek van de gegevens uit je bedrijf (vb. Klanten records die wijzigen)

Waarom: Datawarehouse
Decision making ondersteunen

27-9-

Het doel van datawarehouses is het faciliteren van "business intelligence". Om goede beslissingen te nemen moet het management 'betrouwbare' en zo volledig mogelijke informatie hebben.

Voorbeeld: HP had een aantal jaar geleden verschillende afdelingen waaronder "Printing", "PC/Laptop" en "Enterprise services" (vereenvoudiging van de realiteit). De twee eerste afdelingen deden het al een aantal jaar niet zo goed. Het management zou dus al lang hebben kunnen beslissen de twee eerste afdelingen af te stoten. Toch is die beslissing nog maar pas genomen.
Uit analyses bleek namelijk dat de synergie tussen "PC/Laptop" en "Enterprise services" zorgde voor betere resultaten in die laatste afdeling.
Toen bleek dat ook dit niet meer het geval is werd het bedrijf ook daadwerkelijk opgesplitst.

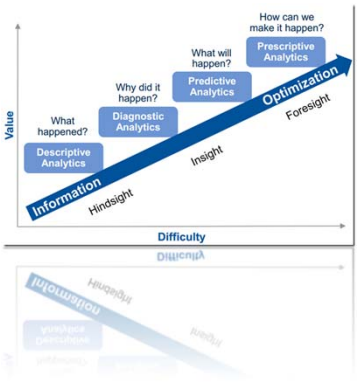
Inleiding

Datawarehouse

faciliteert Business Intelligence (BI)

BI toepassen heeft veel voordelen

- **Competitief voordeel:**
  - Beter inzicht in business dan andere bedrijven
- **Verhogen ROI:**
  - Optimalisatie bedrijfsprocessen
  - Reductie klantenretentie (Klanten behouden)
  - ...
- **Verhogen productiviteit beslissingsmakers:**
  - Door analyses sneller en beter inzicht als beslissingsbasis
  - ...



27-9-

Inleiding

Analyse types

27-9-

- Descriptive: Heel eenvoudige rapportjes maken
  vb. Sales per dag
- Diagnostic: Onderzoeken welke factoren invloed hebben op bepaalde getallen
  vb. Waarom zijn de sales van consoles zoveel hoger in november en december?
- Predictive: Voorspellen wat die getallen in de toekomst gaan zijn
  vb. Hoeveel consoles moet ik produceren om in november en december genoeg voorraad te hebben?
- Prescriptive: Systeem bekijkt mogelijke alternatieven en gaat zelf het beste alternatief voorschrijven
  vb. De analysetools van Amazon bekijkt gedurende de hele dag of hij prijzen moet aanpassen. Als een concurrent zijn prijs aanpast zal amazon automatisch bekijken of het hierdoor verstandig is zijn prijs aan te passen. Amazon weet perfect bij welk prijsverschil voor een dergelijk product de klant zal uitwijken naar een alternatieve leverancier. Er worden dagelijks 2,5 miljoen prijswijzigingen doorgevoerd bij Amazon

Inleiding

Waarom afzonderlijke IT component

Operationeel systeem vrijwaren

Ontwerp optimaliseren voor ander type query's

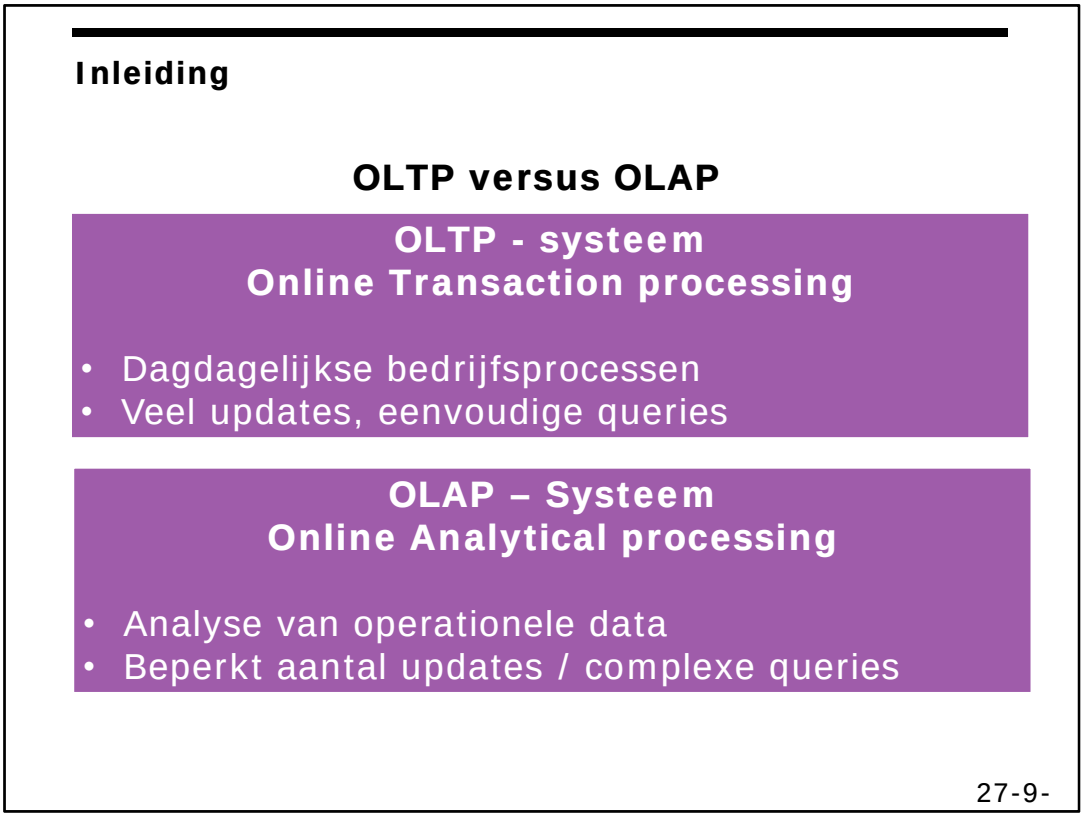
Data integratie vanuit verschillende interne en externe gegevensbronnen

27-9-

Analyses op draaiende operationele systemen hebben als risico dat ze de performantie van het operationele systeem naar beneden halen. Dit kan de operationele werking van het bedrijf in het gedrang brengen. Het is typisch moeilijk te voorspellen wat de impact van één of meerdere analysequeries zijn op het operationele systeem.

Operationele systemen zijn ook niet gemaakt om analyse queries uit te voeren. Voor dergelijke queries zijn vaak specifieke optimalisaties nodig. We bekijken deze verder in meer detail

De sterkte van een datawarehouse zit hem in het combineren van verschillende databronnen. Hierdoor kan je veel meer nuttige verbanden trekken die voorheen niet mogelijk waren.



De OLTP systemen zijn de operationele systemen die gebruikt worden voor dagdagelijkse bedrijfsprocessen (facturatie, voorraadbeheer enz.). Dergelijke systemen zijn geoptimaliseerd om massa's kleine updates en zeer gerichte operationele queries uit te voeren.

OLAP systemen zijn bedoeld om analyses uit te voeren. Ze worden geoptimaliseerd voor complexe queries. In het verleden was het aantal updates eerder beperkt door de limitatie van de vroegere systemen. Deze beperkingen vallen de laatste jaren wat weg (vb. BigData infrastructuur) waardoor het kenmerk van "beperkt aantal updates" stilaan wat vervaagt. Het blijft echter wel de focus om deze system te optimaliseren naar analyses toe.

Inhoud

1. **Inleiding**
2. **Ontwerp**
   1. **OLTP vs OLAP**
   2. Stermodel
   3. Database optimalisaties
   4. ETL Ontwerp overwegingen
3. **Architectuur**

27-9-

Ontwerp

OLTP

Genormaliseerd

- Focus op:
  - Efficiënte dataopslag
  - Frequente updates
  - Eenvoudige queries
- Veel tabellen
- Veel relaties

27-9-

Bij een OLTP systeem ligt de focus op:

- **Efficiënte dataopslag:** Een goed ontworpen operationeel systeem is volledig genormaliseerd. Hierdoor worden geen dubbele gegevens opgeslagen en wordt de opslag zo efficiënt mogelijk gebruikt

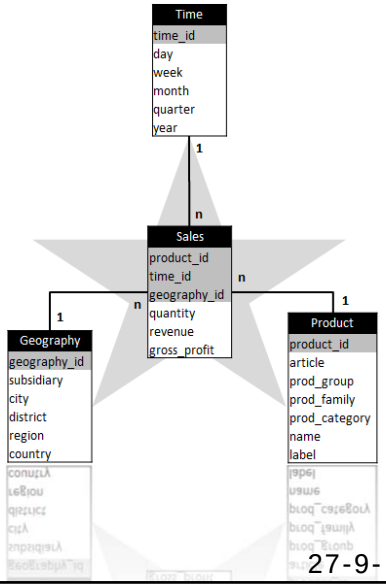
- **Frequente updates:** Gedurende de hele dag worden er voortdurend kleine wijzigingen aangebracht (vb. Nieuwe orders, Nieuwe klant, Update klant…)

- **Eenvoudige queries:** Als er een select-query wordt uitgevoerd is die meestal heel eenvoudig met maar enkele joins

Ontwerp

OLAP
gedenormaliseerd

- Focus op:
  - Efficiënte queries
  - Veel complexe leesoperaties
  - Beperkt aantal updates
- Weinig relaties verwezenlijkt door denormalisatie



27-9-

Bij een OLAP systeem ligt de focus op:

- Efficiënte query's: We moeten vaak heel veel data verwerken voor onze analyses. Daarom moeten we er voor zorgen dat we efficiënte queries kunnen schrijven. We geven hiervoor de efficiëntie opslag op.

- Veel leesoperaties: Het systeem moet in staat zijn veel leesoperaties te verwerken

- Beperkt aantal updates: Updates komen relatief gezien minder voor dan in operationele systemen. Vaak wordt het OLAP systeem niet in real time in sync gehouden met de andere systemen.

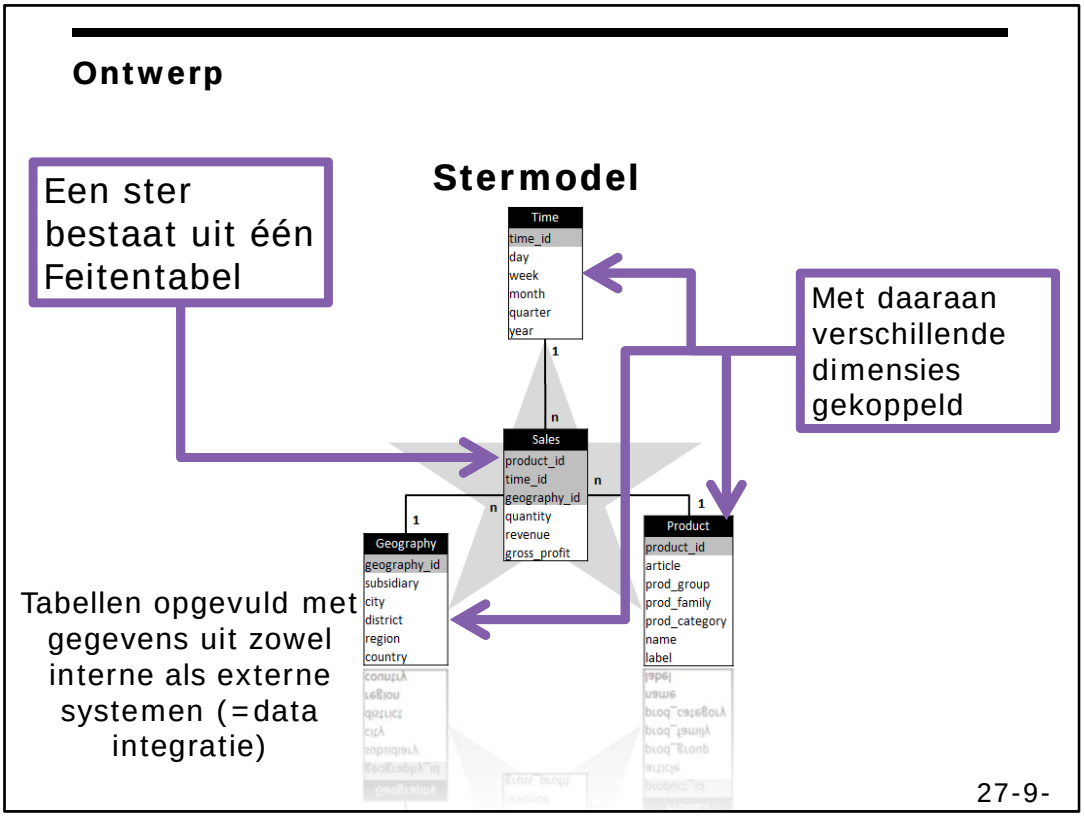
Om dit te realiseren moeten we onze data anders gaan opslaan. Hiervoor passen we denormalisatie toe. We drukken onze databank plat zodat er nooit sprake is van complexe joins.

- *In de tabel Product: prod_category verwijst niet naar een afzonderlijke tabel met 2 rijen. De product category "Food" komt meerdere keren voor als String type in de tabel Product*

- *In de tabel Sales: gross_profit is op voorhand (tijdens het importen van de data) berekend op basis van opbrengst van 1 product (uit product brontabel) – kostprijs 1 product (uit product brontabel) * aantal verkochte producten in sale (uit saleslineitem brontabel)*

Inhoud

27-9-



In een OLAP systeem wordt vaak gekozen voor een "stermodel". Een stermodel is een extreme vorm van. **Denormalisatie.** De regels die je de vorige jaren hebt leren toepassen rond normalisatie gelden niet meer als we een databank willen bouwen ten behoeve van analyse.
Centraal in de ster staat een "feit". Aan het feit zijn verschillende "dimensies" gekoppeld.

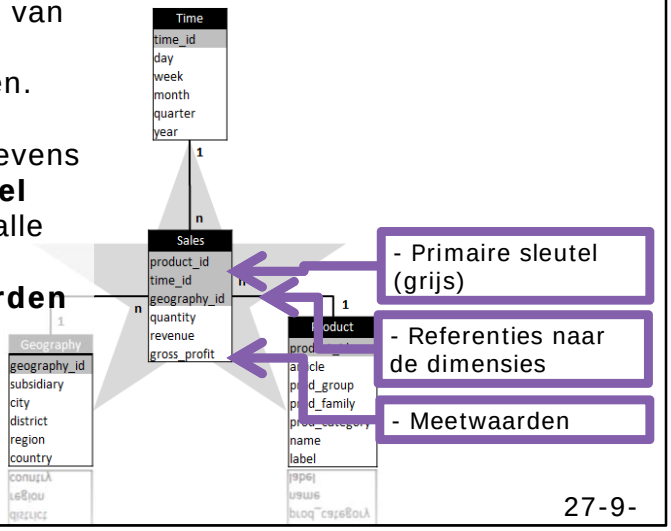
De tabellen worden gevuld met gegevens uit interne en externe systemen. Het combineren van verschillende bronnen noemen we data integratie:

- gegevens uit het operationele systeem
  product_family afkomstig uit operationeel systeem
- uit externe bronnen
  district uit geography wordt uit een externe bron gehaald

Ontwerp

Stermodel:Feitentabel

- Bevat feiten (numerieke performantie eenheden) van subjectarea waarover gerapporteerd zal worden.
- Het bevat volgende gegevens
  - **Een primaire sleutel**
  - **Foreign keys** naar alle dimensies
  - **Feiten = Meetwaarden**



27-9-

Centraal in een feitentabel zijn de effectieve feiten.
Het is een feit dat deze "SalesLine" 1000 EUR omzet heeft opgeleverd. (De titel van het feit is Sales. In tegenstelling wat je zou denken wordt hier niet het resultaat van een verkoop in opgeslagen, maar het resultaat van een verkoopslijn (salesline). Feiten worden idealiter bewaard op atomair niveau (niet de volledige sale als één rij, maar elke SalesLine als een rij). Atomaire rijen groeperen (salesline groeperen naar sale) is altijd mogelijk maar het is niet mogelijk een gegroepeerde rij correct op te splitsen in kleinere delen (uit een salerij kan je niet zomaar saleslines afleiden).

-Feiten zijn steeds **meetwaarden**
quantity, revenue, gross_profit

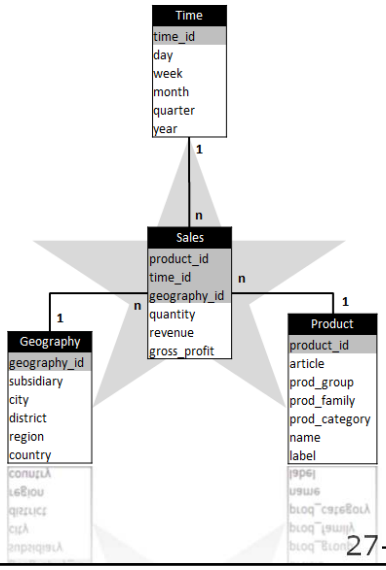
-Een feitentabel heeft ook een **primaire sleutel**. Dit is meestal een subset van de foreign keys. Deze velden maken de rij uniek. In bepaalde gevallen kan het aangeraden zijn een afzonderlijke surrogate key aan te maken als primaire sleutel.
product_id, time_id, geography_id

-Tenslotte heeft een feitentabel ook **foreign keys naar zijn dimensies**.
product_id, time_id, geography_id

Ontwerp

Stermodel:Feitentabel

- Meetwaarden moeten additieve waarden zijn. Het moet zin hebben om ze op te tellen
- Meetwaarden zijn de feiten waarrond je gaat rapporteren.
- Een feitentabel heeft meestal **veel** rijen maar een **beperkt aantal kolommen** die niet veel geheugenruimte in beslag nemen**.**



27-9-

Additief wil zeggen dat het optellen ervan nut heeft.
Hoeveelheid is additief, prijs niet want het optellen van een productprijs heeft niet veel zin in een rapport. Je kan productprijs moest het nodig zijn ook gewoon afleiden door revenu/quantitiy te berekeken.

Als we een rapport willen opstellen doen we dat meestal in functie van een feit. Stel dat we het aantal verkochte items per prod_family willen kennen: We zullen in dit geval de meetwaarde quantity gebruiken.

Een feitentabel heeft vaak heel veel rijen (vb. Alle **verkoopslijnen van alle verkopen** van Colruyt!). Daarom moet er naar gestreefd worden dat de rijen zo smal mogelijk zijn. De breedte van de rij kan beperkt worden door:

- Het aantal kolommen te beperken
- De grootte van de kolommen te beperken: Gebruik bij voorkeur enkel numerieke kolommen (zorg er dus voor dat de FK naar de dimensies numeriek zijn)

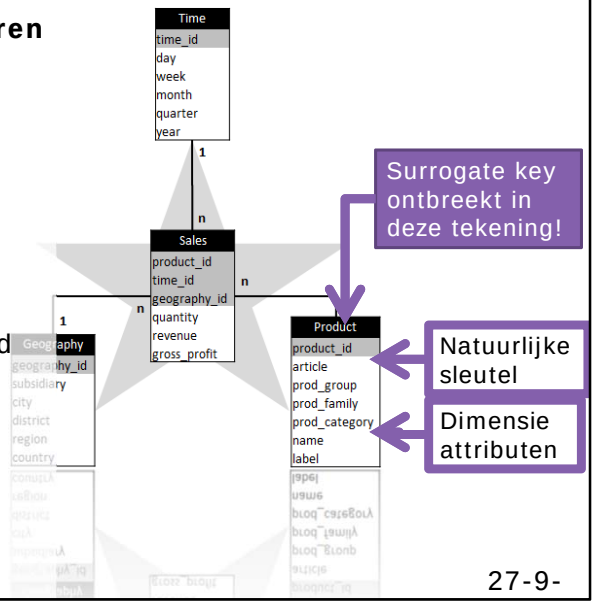
Ontwerp

Stermodel:Dimensietabel

- **Doel: Categoriseren en filteren feiten**

- Bevat volgende gegevens:
  - Surrogate_key
  - Natuurlijke sleutel
  - Dimensie attributen:
    - Hebben leesbare inhoud
    - Neem zo veel mogelijk nuttige categorieën op

- Dimensies bevatten best zo weinig mogelijk rijen maar ze mogen heel veel kolommen bevatten



27-9-

Een dimensie heeft als functie te categoriseren/groeperen en filteren.
Hoeveel producten heb ik verkocht met prod_category='Food ' (filteren) per month (groeperen)

Vaak wordt getwijfeld of het nodig is een surrogate_key te gebruiken. Er zijn verschillende redenen waarom je niet gewoon koppelt met een natuurlijke sleutel maar één van de belangrijkste reden is performantie. Een natuurlijke sleutel is vaak samengesteld uit verschillende attributen. Dat wil zeggen dat je bij een join ook moet koppelen op die verschillende attributen en dat de feitentabel ook extra kolommen moet opslaan. Dit heeft een negatief effect op de performantie. Daarnaast is er bij een SCD Type 2 dimensie sowieso een surrogate key nodig (zie verder)

Natuurlijke sleutel: De natuurlijke steutel voor de dimensie product is product_id. Deze zal in rapporten ook regelmatig gebruikt worden.

Dimensie attributen zijn allerhande attributen die op een rapport gebruikt kunnen worden. De inhoud van deze velden zijn leesbaar (vb. prod_category is "Food" of "Non-food"). De attributen worden namelijk rechtstreeks gebruikt in een rapport. Voeg in geen geval bijkomende tabellen toe voor de verschillende categoriewaarden (vb. tabel prod_category). Het aantal joins stijgt hierdoor enorm snel met een negatief effect op de query performantie.

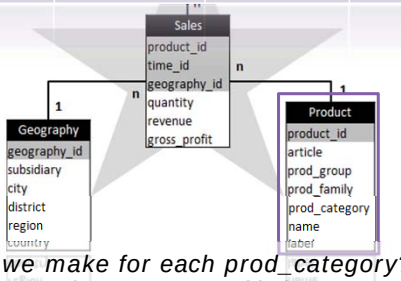
Een dimensietabel mag heel breed worden (m.a.w. veel velden hebben) maar heeft best relatief weinig rijen (t.o.v. het aantal feitrijen). Je moet je dus niet beperken in het aantal categorieën. Elke extra categorisatie kan nuttig zijn voor het rapporteren. Je moet er echter wel op toezien dat je je steeds beperkt tot het strikt noodzakelijke aantal rijen. Dit heeft te maken met het feit dat je dimensies steeds gebruikt om te categoriseren en filteren. M.a.w de databank gaat vaak de volledige dimensie moeten doorlopen. Hoe minder rijen er zijn hoe sneller de query zal uitgevoerd worden. In de feitentabel staat op elke FK een index. Zodra de databank de correcte PK te pakken heeft uit de dimensie zijn de gegevens uit de relatief compacte feitentabel zeer snel opgehaald.

Ontwerp

Stermodel:Dimensietabel

Time

Product_id	Article	Prod_group	Prod_family	Prod_category	Name
1	Sparkling wine	Wine	Beverage	Food	Cava
2	RollerPen	School	Writing material	Non-food	Stabilo RollerPen
3	Dry Wine	Wine	Beverage	Food	Bordeaux



How much gross_profit do we make for each prod_category?

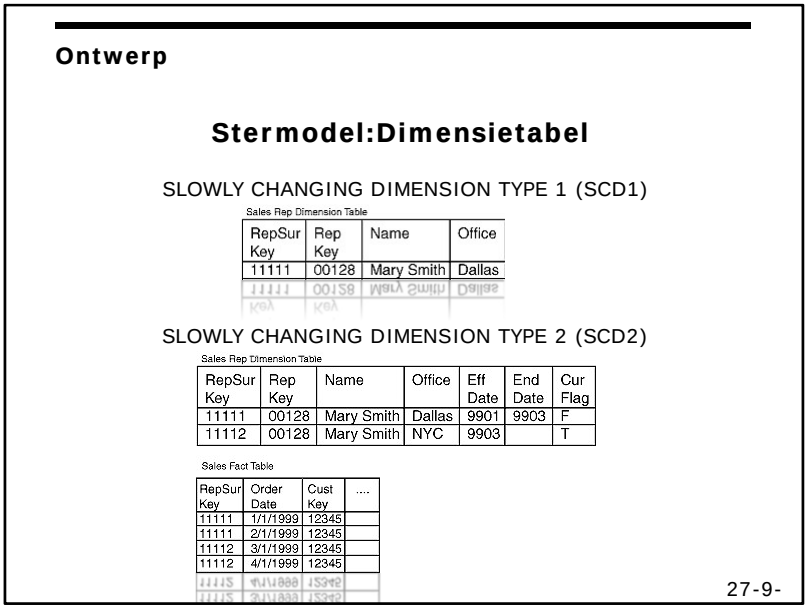
```
Select p.prod_category, sum(s.gross_profit)
from sales s
inner join product p ON  s.product_id = p.product_id
group by p.prod_category
```

27-9-

Hierboven wordt een extract van de dimensie Product weergegeven. Merk op dat veel velden vaak dezelfde waarde hebben. Dit is het gevolg van het denormaliseren en resulteert uiteraard in inefficiënte dataopslag. Dit is de prijs die betaald wordt voor efficiëntere queries.

Een rapportquery op een datawarehouse is in het algemeen niet complex. De nodige joins tussen de feiten tabel en de dimensies worden gelegd en in de select wordt aangegeven welke gegevens in het rapport opgenomen worden. In dit geval wordt er gegroepeerd op prod_category en wordt per category de winst gesommeerd.

Tracht ook zelf eens een query te maken waarbij je de zowel de dimensie Geography als Time gebruikt.



We nemen hier het voorbeeld van vertegenwoordigers. In de dimensie wordt zowel de naam als het kantoor waarvoor ze werken opgeslagen. Als we er van uitgaan dat we hier over een Sales ster spreken zouden we de dus eenvoudig volgende vragen kunnen stellen:

1. *Hoeveel omzet wordt er gegenereerd per "Vertegenwoordiger"?*
2. *Hoeveel omzet wordt er gegenereerd per "Kantoor"?*
3. *Hoeveel omzet wordt er gegenereerd door vertegenwoordigers die hun carrière gestart zijn in Dallas*

De inhoud van dimensies kan in de loopt der tijd wijzigen. Stel bijvoorbeeld dat Mary Smith vanaf 01-03-99 voor het kantoor in NYC gaat werken.
We kunnen dat op verschillende manieren uitwerken. Mijn spreekt hier over slowly changing dimensions (SCD). We beperken ons in dit geval tot 3 types

SCD Type 1:
We wijzigen de bestaande record bij een wijziging in het bronsysteem
Als de dimensietabel op 01-03-1999 geupdate wordt dan zal de rij met RepSurKey 11111 wijzigen. Het veld Office zal de waarde NYC krijgen.

Wat is het gevolg voor de rapporten? Stel dat je na 01-03-1999 een lijst wil maken voor analysevraag 2 voor de verkoopresultaten van februari. De verkopen van Mary zullen onterecht worden toegewezen aan Office NYC. Dat is uiteraard niet het resultaat dat we willen bekomen.

SCD Type 2:
We passen de bestaande record niet aan maar maken voor dezelfde natuurlijke sleutel een nieuwe rij aan. Om dit te realiseren voegen we een start en einddatum toe die de geldigheidsperiode van de rij voorstelt. Voor de reeds bestaande rij sluiten we die periode af (we stellen een einddatum in). Voor de nieuwe rij laten we de einddatum open (of gebruiken end of time) en zetten we het einddatum van de reeds bestaande rij.

Als de dimensietabel op 01-03-1999 geüpdatet wordt dan zal voor de rij met RepSurKey 11111 de einddatum worden ingevuld met 01-03-1999. Er wordt voor RepKey 00128 ook een tweede rij aangemaakt (RepSurKey 11112). In die rij komt als begindatum 01-03-1999 . Verder verschilt in beide rijen enkel het Office-veld

Wat is het gevolg voor de rapporten? Stel dat je na 01-03-1999 een lijst wil maken voor analysevraag 2 voor de verkoopresultaten van februari. De verkopen van Mary zullen correct worden toegewezen aan Office Dallas. In dit geval bekom je hier het juiste resultaat

Ontwerp

Stermodel:Dimensietabel

SLOWLY CHANGING DIMENSION TYPE 3 (SCD3)

Sales Rep Dimension Table

RepSur Key	Rep Key	Name	Office Current	Office Past
11111	00128	Mary Smith	NYC	Dallas

27-9-

SCD Type 3:
Bij type 3 kiezen we ervoor de bestaande rij aan te passen maar het office veld op te splitsen in Office Current en Office Past. Telkens de office wijzigt wordt bij Current de nieuwe Office geplaatst. Je merkt dat je hier de geschiedenis maar beperkt kan modelleren in twee momenten Heden en verleden.

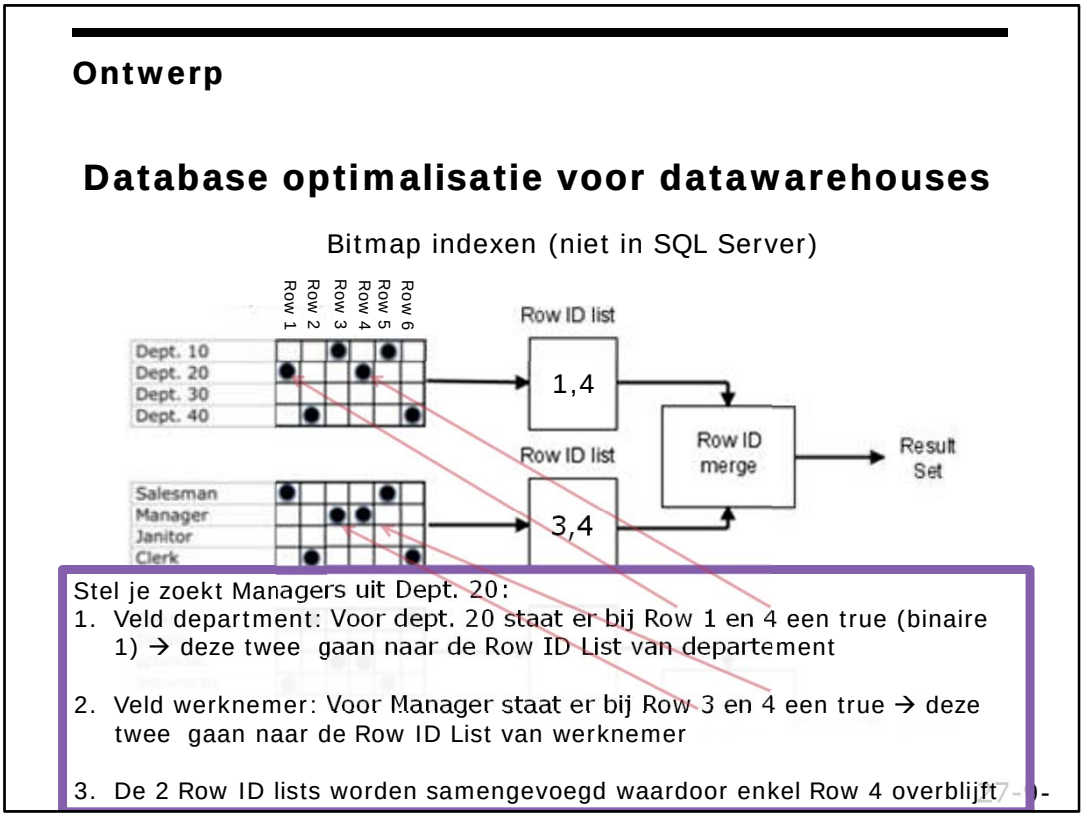
Als de dimensietabel op 01-03-1999 geupdate wordt dan zal voor de rij met RepSurKey 11111 de office current wijzigen van Dallas naar NYC. Bij past stond er al Dallas en blijkt het Dallas.
*Wat is het gevolg voor de rapporten? Stel dat je na 01-03-1999 een lijst wil maken de omzet van vertegenwoordigers die hun carriere gestart zijn in Dallas (**analysevraag 3**). Dit is mogelijk met deze uitwerking. Je kan daarnaast ook opvragen wat de resultaten zijn van de vertegenwoordigers die momenteel in NYC werken (los van het kantoor waar de vertegenwoordiger in het verleden werkte).*

Welke Type gebruik ik?
Het hangt sterk af van de analysevragen die je wil stellen welk type nodig is. Hierbij moet wel opgemerkt worden dat het gebruik van type 2 het aantal rijen in een dimensie snel kan laten toenemen. Er moet dus enkel overgegaan worden naar type 2 als dit echt nodig blijkt voor het rapporteren.

Zijn er nog andere types?
-Er worden nog andere types beschreven zoals type 6. Dit is eigenlijk een combinatie van type 2 en 3 (2*3) waarbij voor nieuwe records voor bepaalde velden ook hun oorspronkelijke waarden behouden worden. Dit combineert de voordelen van beide types.
-Je kan ook een overzichtje vinden via volgende link:
https://en.wikipedia.org/wiki/Slowly_changing_dimension

Inhoud

27-9-



Voor typische analysequeries moet de databank typisch heel veel rijen doorzoeken. Door het gebruik van een stermodel is in ieder geval het aantal joins in zo'n query beperkt wat een positieve impact heeft op de "cost" van de query. Daarnaast zijn er een aantal bijkomende optimalisatietechnieken die de "cost" van een query drastisch naar beneden halen.

Bitmap indexen:

In de meeste databanken is de standaard wijze van indexeren een B-tree index. De database zoekt in een boomstructuur naar de juiste waarde en hieraan is dan een rowID gekoppeld. Voor waarden die vaak voorkomen (vb. een categorie) is een B-treeindex vaak niet de beste oplossing.

Een veel gebruikt alternatief in datawarehouses is een Bitmap index. Een bitmapindex houdt voor elke mogelijke waarde in een veld een bitmap bij. Dit bitmap zet voor alle rijen een 0 als het veld die waarde niet bevat en een 1 als dat veld die bepaalde waarde niet bevat. Je merkt dat zo'n bitmap index snel heel groot kan worden. Bij het gebruik van een bitmapindex is het zoeken naar de juiste rowIDs in het algemeen een stuk sneller dan bij een B-tree index.

Algemeen wordt volgende regel aangehouden:
Indien "aantal distinct values / Aantal rijen <= 1%" gebruik dan een Bitmapindex

Het nadeel van bitmap indexen is dat ze redelijk wat rescources vergen bij de creatie ervan. Ze kunnen daarom best gebruikt worden in een omgeving met een beperkt aantal updates en inserts wat vaak het geval is in een datawarehouse.

Ontwerp

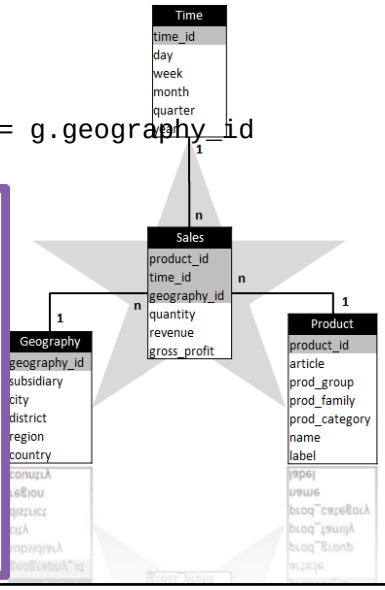
Database optimalisatie voor datawarehouses

Bitmap indexen (niet in SQL Server)

```
Select sum(quantity)
from sales s
inner join geography g on s.geography_id = g.geography_id
Where g.city = 'Antwerp'
```

Stel er staat een bitmap index op geography_id in Sales

1. Eerst wordt de g.geography_id voor 'Antwerp' gezocht in de dimensie Geography.
2. Die gevonden g.geography_id wordt gebruikt om met de bitmapindex op s.geography_id op zoek te gaan naar de sales row ids waarbij die specifieke id op true staat. De db engine filtert op de 1-tjes bij value 'Antwerp' in de bitmap index
3. Met die row ids kunnen de rijen snel uit sales opgehaald worden.



In een datawarehouse wordt heel vaak gebruik gemaakt van een bitmapindex op de foreign keys naar de dimensies in het feit. Het aantal rijen in een dimensie is vaak heel beperkt ten opzichte van het aantal rijen in het feit.

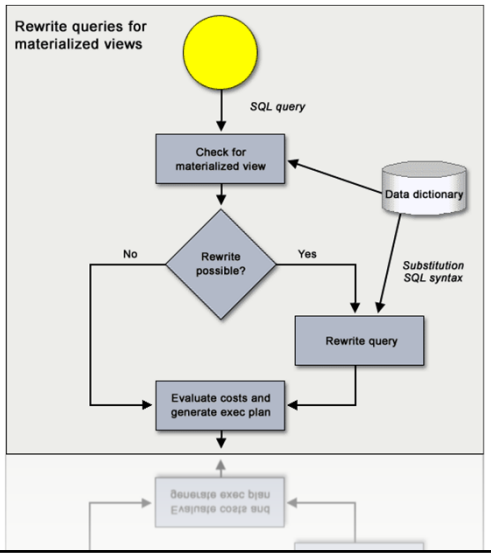
Stel dat in Geography 10 000 rijen zit en in Sales 2 000 000.
10 000 / 2 000 000 = 0,5% → Het is dus nuttig een bitmap index te leggen op geography_id in Sales

Op de slide kan je zien wat er zal gebeuren als je een eenvoudig rapportje opvraagt.

Ontwerp

Database optimalisatie voor datawarehouses

Materialized views/ Indexed views en query rewrite



- Materialized view (in SQLServer Indexed View) is een view die fysiek wordt opgeslagen
- Query rewrite: techniek die onderzoekt of een uit te voeren query kan uitgevoerd worden op een materialized view in plaats van op de originele tabel

27-9-

Feiten hebben de eigenschap vaak heel veel rijen te bevatten. Omdat een feit echter doorgaans enkel foreign keys bevat (die normaal gezien numeriek zijn) en meetwaarden (die ook numeriek zijn) nemen dergelijke rijen meestal weinig plaats in de databank. Toch kan bij heel grote hoeveelheden feitenrijen de performantie van queries niet meer naar behoren zijn. Sommige queries kunnen dan uren duren en zoals altijd krijgt het IT-departement daarvan de schuld.

Om dit te verhelpen hebben verschillende databankproducenten het concept van materialized views ontwikkeld. Een materialized view is eigenlijk het omzetten van view naar een fysieke tabel met inhoud.

Onlosmakelijk verbonden met een materialized view zijn query rewrites. Als een gebruiker een query lanceert gaat de databank engine bekijken of hij voor het uitvoeren van de query niet evengoed de materialized view kan gebruiken. De databank herschrijft de query en bekijkt dan de "cost" van beide oplossingen. Blijkt die met de materialized view kleiner dan zal hij die query uitvoeren in plaats van de originele.

Ontwerp

Database optimalisatie voor datawarehouses

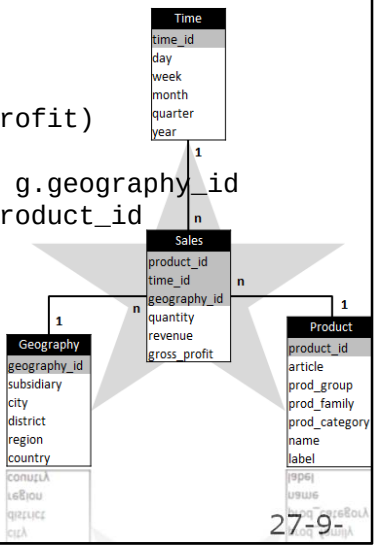
Vb.Materialized views en query rewrite

How much gross_profit do we make for each prod_category and country

```
select country, prod_category, sum(gross_profit)
from sales s
inner join geography g on s.geography_id = g.geography_id
inner join product p on p.product_id = s.product_id
group by g.country, p.prod_group
```

Stel:

- 90% van de rapporten maakt geen gebruik van de dimensie Time.
- De ster sales groeperen op product_id en geography_id levert 100 000 rijen op in plaats van 3 000 000



27-9-

Op de slide zie je een voorbeeld rapport query en een aantal veronderstellingen.

De gebruikers zijn niet meer tevreden over de performantie van de queries.

Het IT-team beslist een materialized view te maken die aggregeert op product_id en geography_id

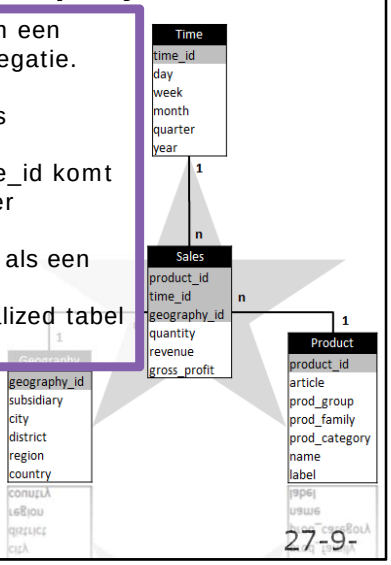
Ontwerp

Database optimalisatie voor datawarehouses

Vb. Materialized views en query rewrite

Voor betere performantie kan gekozen worden om een materialized view te maken op die geography aggregatie.

1. De query voor de materialized view zou er dan zoals hieronder kunnen uitzien
2. De query komt overeen met de ster sales maar time_id komt er niet in voor en de aantallen zijn gesommeerd over product_id en geography_id.
3. Het resultaat van de query wordt fysiek opgeslagen als een nieuwe tabel (materialized)
4. Aangezien het een aggregatie betreft zal de materialized tabel veel kleiner zijn dan de originele sales tabel




```
select product_id, geography_id,
sum(quantity), sum(revenu),
sum(gross_profit)
from sales s
group by product_id, geography_id
```

27-9-

We maken een "materialized view" aan die alle salesdata voor een product groepeert per regio. Zo verkleinen we al het aantal rijen uit de feitentabel wanneer we moeten gaan joinen om de gevraagde groepering te verkrijgen. De query van de view zie je op de slide hierboven. De database engine voert de view-query uit en slaat het resultaat op een afzonderlijke tabel. Deze tabel bevat 100 000 rijen. De originele Sales tabel bevat 3 000 000 rijen.

Ontwerp

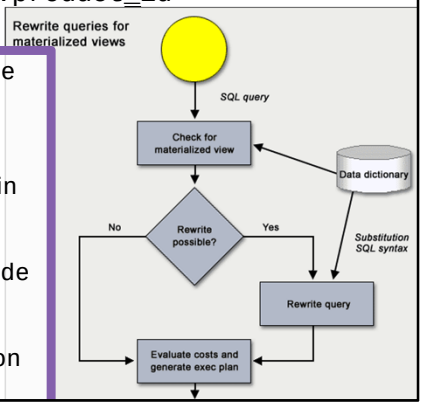
Database optimalisatie voor datawarehouses

Vb.Materialized views en query rewrite

How much gross_profit do we make for each prod_category and country

```
select country, prod_category, sum(gross_profit)
from sales s
inner join geography g on s.geography_id = g.geography_id
inner join product p on p.product_id = s.product_id
group by g.country, p.prod_group
```

1. Bij het uitvoeren van de query wordt gekeken of de query past binnen de materialized view.
2. Omdat groeperen op country kan binnen de groepering op geography_id en product_id (zoals in de mat. View) is een query rewrite mogelijk
3. Omdat de materialized view veel compacter is zal de query veel sneller worden uitgevoerd
4. Als een query rewrite niet mogelijk is wordt gewoon de originele sales tabel gebruikt



Als de materialized view gemaakt is en we voeren de query uit voor het rapport zullen de stappen in de rode kader doorlopen.
In dit concrete geval wil dit zeggen dat de query kan gedraaid worden op een tabel met 100 000 rijen in plaats 3 000 000 rijen. Als dit zo is voor 90% van onze queries kan hiermee uiteraard een immense performantie verbetering gerealiseerd worden.

Ontwerp

Database optimalisatie voor datawarehouses

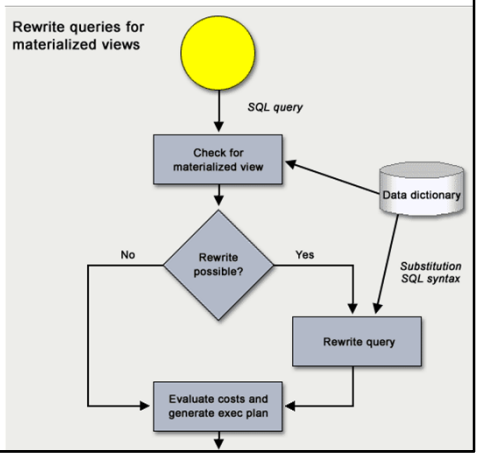
Vb Materialized views en query rewrite

How much gross_profit do we make for each prod_category and country

To good to be true?

Bij een wijziging in de originele sales tabel is de materialized view 'out of sync' en is hij onbruikbaar tot de materialized up to date wordt gebracht.

Als de specifieke technologie de materialized view onmiddelijk voorziet van een update creëert dit natuurlijk overhead bij CRUD operaties



Het probleem bij een materialized view is dat het vaak vrij veel resources vraagt om hem te maken en dat dat bij elke wijziging in de originele tabellen moet gebeuren.
Als een materialized view tijdelijk onbeschikbaar is werken de analysequeries nog, maar ze zijn natuurlijk minder performant omdat ze op de originele tabellen gebeuren.

Database optimalisatie voor datawarehouses

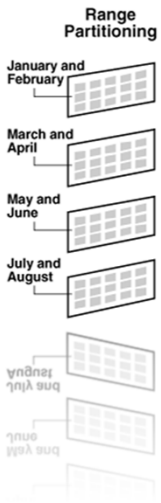
Materialized views en query rewrite bij SQL Server

- Om views te materializen plaats je een clustered index op de view
- De view wordt opgeslagen als een tabel
- De query optimizer past query rewrites toe
- De indexed view wordt bij elke CRUD operatie onmiddellijk geupdate en is dus steeds in sync.

Ontwerp

Database optimalisatie voor datawarehouses

Partitionering



- Techniek waarbij een tabel wordt opgesplitst in sub-tabellen op basis van een bepaalde voorwaarde.
- Vb. als er vaak gefilterd of gegroepeerd wordt per kwartaal, dan kan je opteren om per kwartaal een partitie te voorzien.
- Bij onderstaande query zal enkel de sub-tabel voor kwartaal 1 geraadpleegd worden.
- Omdat de sub-tabellen kleiner zijn zal de performantie van de queries sterk toenemen

```
select t.year, sum(gross_profit)
from sales s,
inner joing time t on s.time_id = t.time_id
where t.quarter = 1
group by t.year
```

27-9-

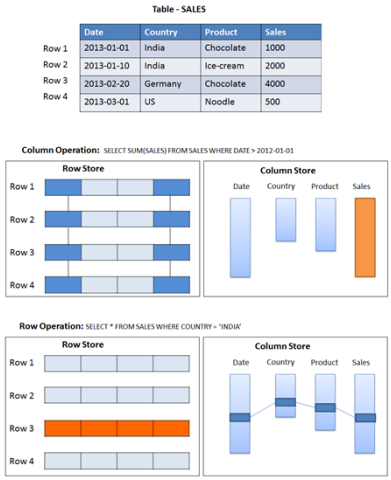
Bij partitionering maakt de databank verschillende fysieke tabellen voor één tabel. Hij splitst de inhoud op volgens een gedefinieerde voorwaarde. Je kan hiermee tabellen eenvoudig verkleinen.
De voordelen zijn duidelijk als de database engine een query kan parallelliseren over de verschillende partities en als hij zich tot één partitie kan beperken. Hierdoor moet hij beduidend minder gegevens doorzoeken.

Een goed artikel over hoe je dit doet in SQL Server 2016, vind je hier: https://docs.microsoft.com/en-us/sql/relational-databases/partitions/partitioned-tables-and-indexes en hier: https://docs.microsoft.com/en-us/sql/relational-databases/partitions/create-partitioned-tables-and-indexes

Ontwerp

Column Storage

- Opslag op kolomniveau i.p.v. rijniveau
- Betere compressie omdat die per kolom kan vastgelegd worden.
- Bij query op beperkt aantal kolommen moeten enkel die kolommen geraadpleegd worden.
- Binnen SQL Server kan je hiervoor een columnstore index gebruiken.



Je bent ondertussen zeer vertrouwd met rij gebaseerde databanken. In dergelijke systemen wordt de data opgeslagen per rij. Hoewel dit voor heel veel use cases goed werkt is dit toch niet altijd het geval. Soms is het aan te raden om de data op kolomniveau op te slaan.
In de afbeelding zie je dat de waarden voor de verschillende rijen per kolom worden opgeslagen. Het datatype bepaalt uiteraard hoe veel plaats een bepaalde kolom inneemt.

Omdat binnen een kolom vaak dezelfde waarde terugkomt (of er duidelijke andere patronen te vinden zijn) kan er zeer goede compressie op een kolom worden geplaatst. Zeer hoge compressie wil zeggen dat de data sneller uit de databank gehaald kan worden. Uiteraard zal de CPU wat harder moeten werken maar de IO vormt meestal een grotere bottleneck.

Vb. In bovenstaande afbeelding worden de dag totalen per land per product bijgehouden. Het dateveld bevat een datum waarbij elk veld 3 bytes lang is. Stel dat we 100 rijen in de database hebben zitten is dat in totaal 300bytes. De database engine weet echter dat die datums geordend zijn. In de column store kunnen we die kolom daarom comprimeren door met incrementen te werken. De eerste datum wordt als datum opgeslagen (3bytes). Voor de daaropvolgende datums wordt het increment opgeslagen in een tinyint van 1byte (voor row 2 wordt 9 opgeslagen omdat Date Row 2= Date Row1 + 9). In plaats van 300bytes hebben we nu 3bytes + 99bytes = 102bytes nodig. Hoewel de opslag van een date veld dus al vrij efficiënt is, slagen we er in de column store erin om alsnog met een factor 3 te comprimeren.

Columnar storage is vooral zeer performant bij queries waar maar een beperkt aantal kolommen gelezen moet worden voor veel rijen. Als alle rijen overlopen moeten worden moet de query engine enkel de kolombestanden lezen die in de query worden gebruikt. Laat dit nu typisch het geval zijn voor datawarehouse queries. Colomnar storage is daarom een techniek die zeer nuttig kan zijn op fact tables en grotere dimensietables.

Ook SQL Server maakt het mogelijk data op te slaan op kolomniveau. Je dient hiervoor gebruik te maken van zogenaamde Columnstore indexes.

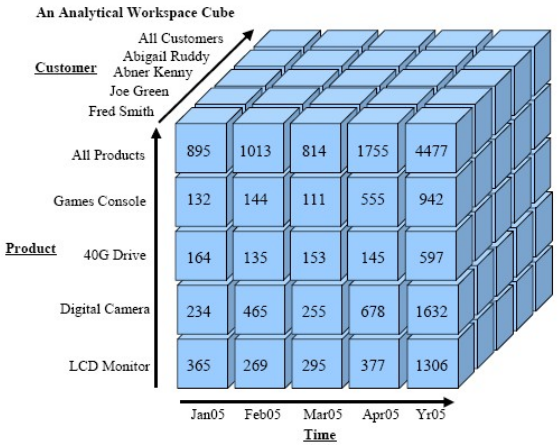
Bovenstaande afbeelding illustreert dat een query die beperkt blijft tot 1 of enkele kolommen maar van toepassing is op veel rijen.

Een goed artikel waarin dit type storage ook toegelicht wordt kan je vinden op https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview en http://www.databasejournal.com/features/mssql/understanding-new-column-store-index-of-sql-server-2012.html

Ontwerp

Database optimalisatie voor datawarehouses

OLAP Cubes



- Verschillende databases bieden ondersteuning voor Cubes
- Centraal staan aggregatie en meetwaarden
- Kolom georiënteerde opslag die aggregatie optimaliseert

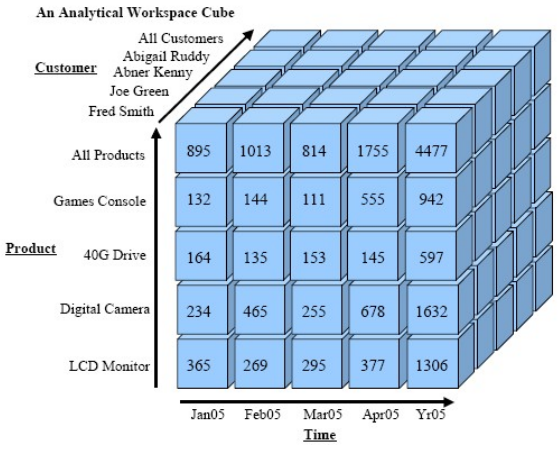
OLAP cubes zijn specifiek voor analytische doeleinden ontworpen. De gegevens worden ook hier op kolom niveau (en dus niet rijniveau) opgeslagen. In een OLAP Cube worden de hiërarchieën binnen dimensietabellen geformaliseerd (Product heeft meerdere Productgroepen die uit meerdere Productcategoriëen bestaan).

Als we het voorbeeld op de slide bekijken zien we dat elk vierkantje in de cube een getal bevat. Van product "LCD Monitor" zijn er door "Fred Smith" in "Jan05" 365 gekocht. Die 365 is eigenlijk al een aggregatie van alle afzonderlijke verkoopslijnen in het datawarehouse. Door deze aggregaties geoptimaliseerd in een databank op te slagen kan de databank ook veel sneller een antwoord geven op bepaalde vragen. Cubes moet je dan ook bouwen in functie van de analysevragen die vaak gebeuren.

Ontwerp

Database optimalisatie voor datawarehouses

OLAP Cubes



- Worden bij voorkeur gegenereerd op basis van stermodellen
- SQL Server: Analysis services > Multidimensional mode.

27-9-

Ook SQL Server heeft de mogelijkheid om data in CUBE structuren op te slaan. Hiervoor dien je gebruik te maken van het afzonderlijke SQL SERVER ANALYSIS SERVICES. Er zijn binnen dit onderdeel 3 verschillende manieren om data op te slaan waarvan de “Multidimensional mode” het concept van cubes toepast. Jammer genoeg kan je geen SQL gebruiken om hiervoor queries uit te voeren

Ontwerp

In-memory databases

- Opslag van data in memory
- Beschikbare technieken voor OLTP en OLAP
- Kan voldoen aan Durability vereiste (ACID) -> achterliggend opslag op schijf
- SQL Server:
  - OLTP: In-memory OLTP
  - OLAP: Analysis Services Tabular Mode

Nu beschikbaar geheugen in servers minder en minder een probleem is, vormt er zich de mogelijkheid gehele tabellen of databases volledig in memory beschikbaar te maken.
Bij SQL Server kan dit voor OLTP systemen met In-memory OLTP. De traditionele relationele tabellen worden dan (zei het in een afwijkende fysieke structuur) in memory opgeslagen.
Dit systeem biedt vooral voordelen bij transactionele queries (zeer gerichte queries) en is minder goed inzetbaar voor een analytische systemen. Omdat het om een RDBMS gaat blijft het natuurlijk belangrijk dat aan de ACID voorwaarde wordt voldaan. Aangezien memory niet Durable is wordt de data achterliggend nog op disk geplaatst zodat alles recoverable is.

Voor OLAP systemen kan je in SQL Server gebruik maken van Ananalys Services in Tabular Mode. Ook hier wordt data grotendeels in memory opgeslagen. Dit wordt echter gecombineerd met een columnar storage. De combinatie hiervan kan een serieuze performance boost betekenen.

Ontwerp

SQL Server Analysis Services

- Database engine specifiek voor analyse taken op data (lezen van groot aantal queries)
- Afzonderlijk compontent
- 2 prominente modes
  - Tabular Mode: Columnar in memory
  - Multidimensional: Cubes
- Querytalen: MDX, DMX, DAX, XML/A, ASSL, and TMSL (dus geen SQL)

Zoals reeds aangegeven bij Cubes en In memory databases heeft SQL Server een aantal services die zich toespitsen op analytische queries. Deze worden gehost door analytical data engine van Analysis Services. Bij de instalatieinstructies hebben jullie dit deel niet geïnstalleerd omdat de meesten onder jullie het ook niet zullen gebruiken.

De engine kan in meerdere modes werken waarvan de belangrijkste tabular en multidemensional zijn (zie voorgaande slides).
Het is niet mogelijk via gewone sql queries informatie uit de database te halen maar er worden wel talen zoals MDX, DMX enz. ondersteund. Dit hoeft geen probleem te zijn omdat rapporteringstools een visuele user interface gebruiken zodat je een rapport bij elkaar klikt. De queries worden door dergelijke tools achterliggend gegenereerd.

Ontwerp

Database optimalisatie voor datawarehouses

Insert/Update optimalisaties: Foreing keys uitschakelen

- Foreign keys hebben grote invloed op performantie bij insert en update statements
  - Elke insert/update →elke foreing key wordt geverifieerd in foreign tables
  - Operationeel systeem: Beperkt aantal insert/updates →Geen probleem
  - Datawarehouse: Verwerking van massa's insert/updats → Enorme invloed op performantie
  - Oplossing: Foreign keys uitschakelen
  - Voorwaarde: De verwijzingen worden gecontroleerd op applicatieniveau

27-9-

Ter illustratie: Een feit verwijst naar 20 dimensies. Als deze allen een FK-constraint hebben dan moeten er voor elke rij 20 tabellen gescreend worden. Als er een miljoen zijn wil dat zeggen dat er 20 000 000 checks moeten gebeuren. Hoewel de database engine hiervoor wel wat optimalisaties kan toepassen heeft dit een enorme invloed op de doorlooptijd van het laden van de database.
Als we die FK-constraints niet toepassen is de controle niet nodig. Er moet dan wel voor gezorgd worden dat er geen foute linken gelegd worden in de software. In een ETL tool wordt hier bij het bouwen van transformaties rekening mee gehouden.

Database optimalisatie voor datawarehouses

Insert/Update optimalisaties: Insert specifieke optimalisaties

- Groepeer rijen in 1 insert statement met meerdere value-lijsten
- Gebruik alternatieve insert technieken: bulk loading, load_data_infile (20 * sneller dan insert statements)
- Beperk het aantal transacties

27-9-

Database optimalisatie voor datawarehouses

Insert/Update optimalisaties: Update specifieke optimalisaties

- Beperk het aantal kolommen in het statement tot het hoogst noodzakelijke
  - Dit is vaak niet evident in een ETL tool (zie verder)

Inhoud

1. **Inleiding**
2. **Ontwerp**
   1. OLTP vs OLAP
   2. Stermodel
   3. Database optimalisaties
   4. **ETL Ontwerp overwegingen**
3. **Architectuur**

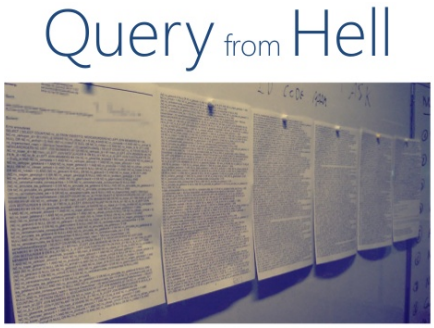
27-9-

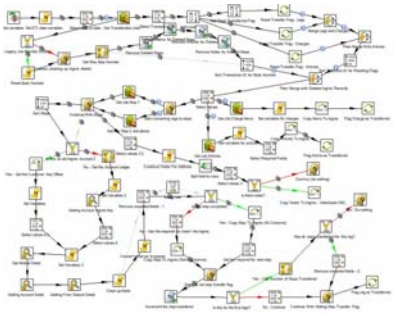
Ontwerp

ETL Ontwerp overwegingen

- Tranfsormatielogica in SQL of Transformatielogica in ETL Tool?
  - Complexe SQL queries maken de performantie afhankelijk van de bron waaruit je leest.
  - Bronnen zijn vaak niet geoptimaliseerd voor dergelijke queries.

→ Plaats zoveel mogelijk transformatielogica in de ETL Tool!





Vaak heeft men de neiging om de functionaliteit van een ETL tool te omzeilen door bij het inlezen van de data complexe queries te schrijven die al het merendeel van de ETL handelingen voor zich neemt. Dit is echter in veel gevallen geen optimale keuze.

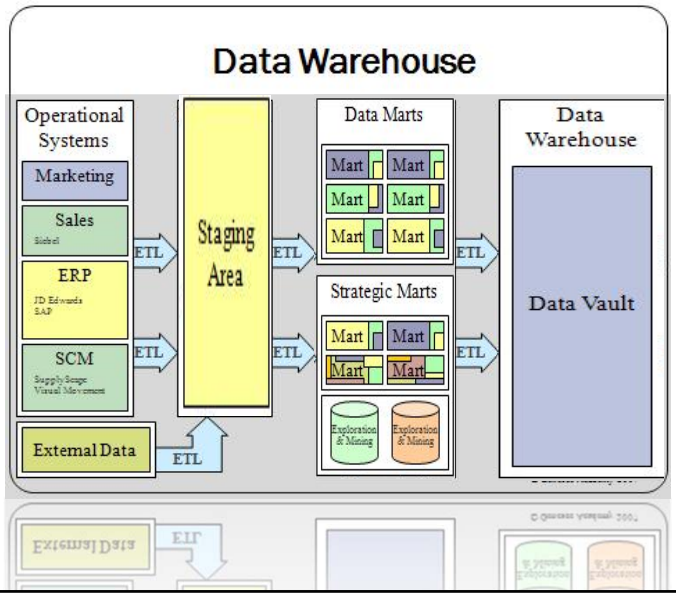
Ten eerste wordt er vaak gelezen uit bronnen die helemaal niet zijn geoptimaliseerd voor dergelijke queries. Je zou ze hiervoor kunnen optimaliseren maar dat kan dan weer een probleem vormen als die bronnen voor andere systemen gebruikt worden.
Ten tweede ben je op deze manier voor de performantie helemaal afhankelijk van de brondatabase. Als je kiest voor eenvoudige queries op de brontabellen die je daarna met ETL stappen transformeert in de ETL Tool zorg je ervoor dat je de optimalisaties zelf in handen hebt

Inhoud

1. Inleiding
2. Ontwerp van een datawarehouse
3. **Architectuur**

Architecturen

Klassieke architecturen

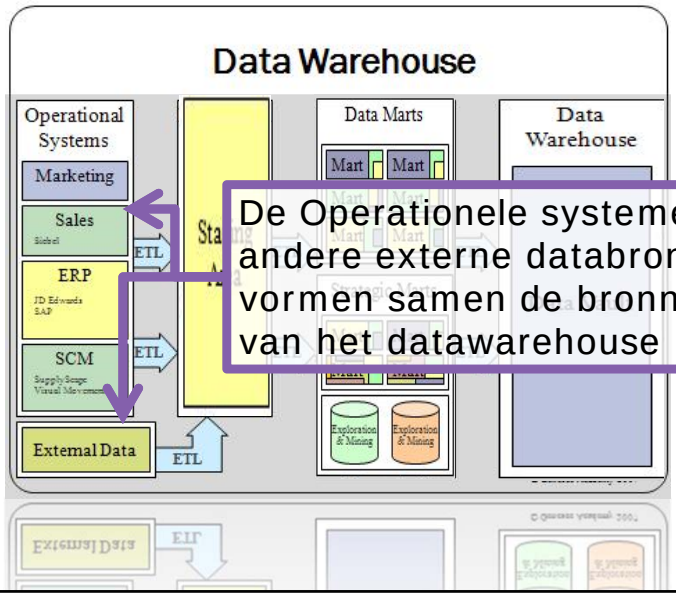


27-9-

We bekijken hier een mogelijke architectuur voor een datawarehouse omgeving en gaan daarvoor dieper in op de visie van Kimball. Bijvoorbeeld Inmon stelt een iets andere visie voor en met de komst van BigData en NoSQL oplossingen zien we ook dat de architectuur wordt aangepast. Hier verder op ingaan zou ons echter te ver leiden in functie van Databanken 3.

Architecturen

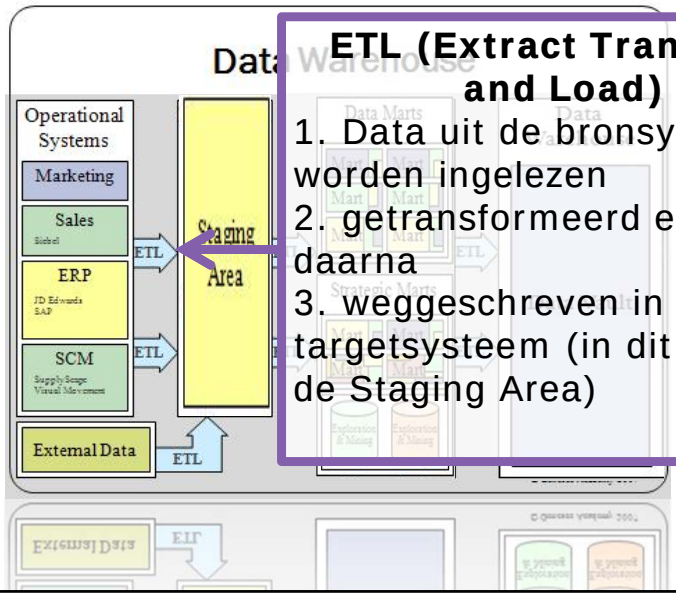
Klassieke architecturen



27-9-

Architecturen

Klassieke architecturen



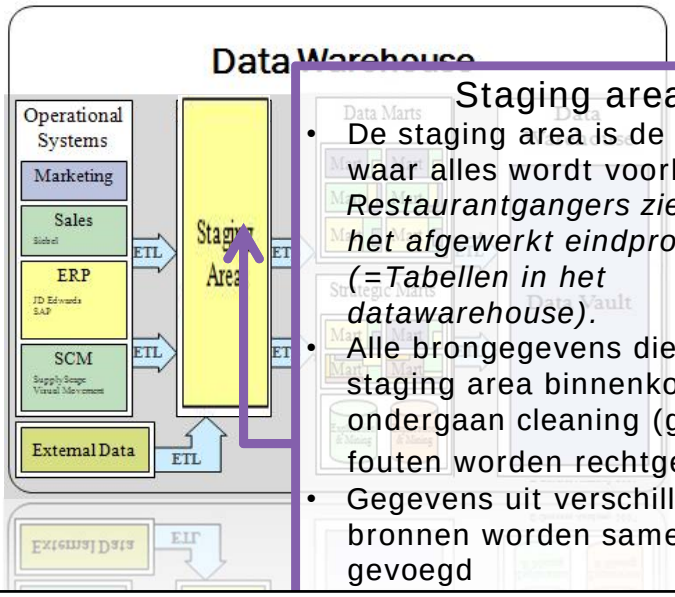
ETL (Extract Transform and Load)

1. Data uit de bronsystemen worden ingelezen
2. getransformeerd en daarna
3. weggeschreven in het targetsysteem (in dit geval in de Staging Area)

27-9-

Architecturen

Klassieke architecturen



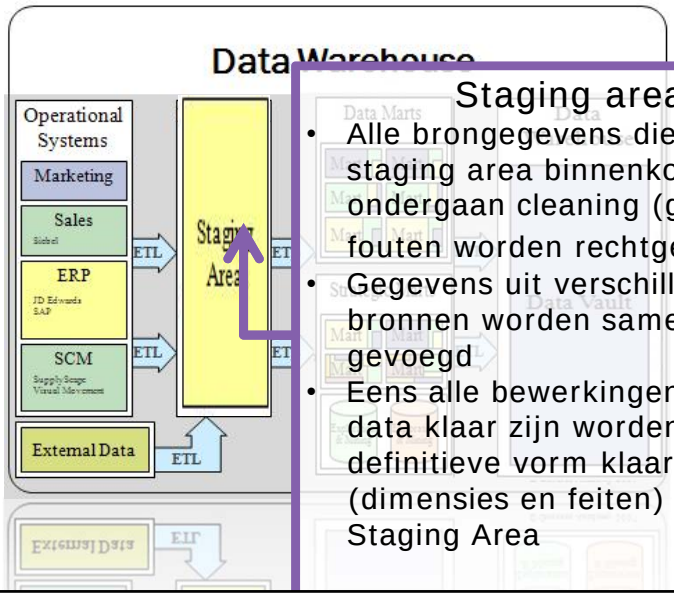
Staging area

- De staging area is de keuken waar alles wordt voorbereid. *Restaurantgangers zien enkel het afgewerkt eindproduct (=Tabellen in het datawarehouse).*
- Alle brongegevens die in de staging area binnenkomen ondergaan cleaning (gekende fouten worden rechtgezet)
- Gegevens uit verschillende bronnen worden samen gevoegd

27-9-

Architecturen

Klassieke architecturen

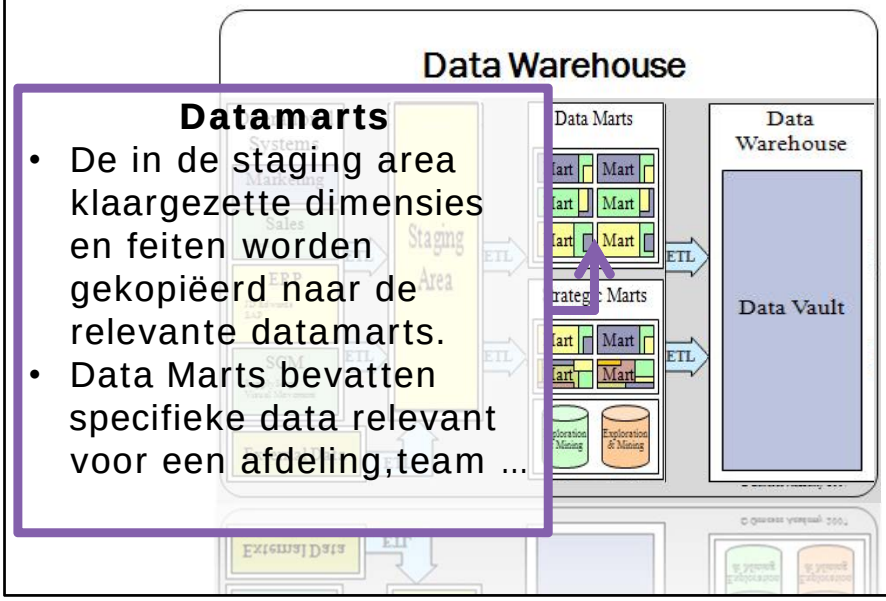


Staging area

- Alle brongegevens die in de staging area binnenkomen ondergaan cleaning (gekende fouten worden rechtgezet)
- Gegevens uit verschillende bronnen worden samen gevoegd
- Eens alle bewerkingen op de data klaar zijn worden ze in hun definitieve vorm klaargezet (dimensies en feiten) in de Staging Area

Architecturen

Klassieke architecturen



Datamarts

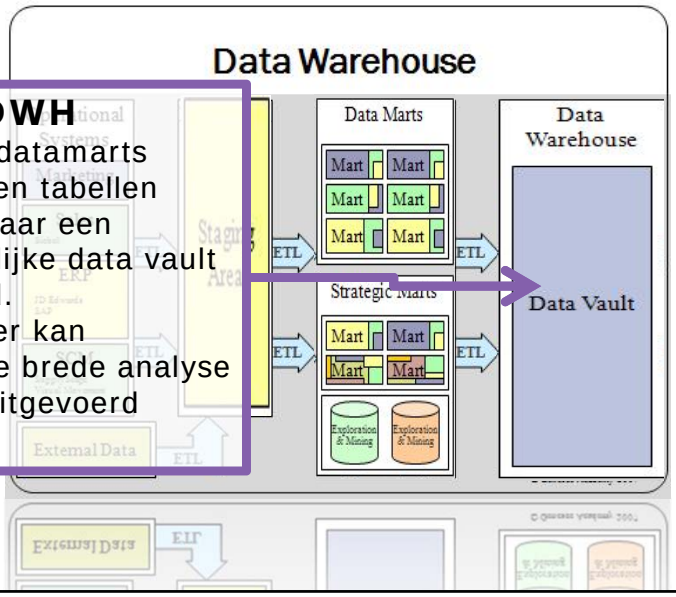
- De in de staging area klaargezette dimensies en feiten worden gekopiëerd naar de relevante datamarts.
- Data Marts bevatten specifieke data relevant voor een afdeling,team …

27-9-

- Datamarts zijn subsets van de data uit een datawarehouse, meestal gegroepeerd per business afdeling

Architecturen

Klassieke architecturen



DWH

- De in de datamarts opgeslagen tabellen worden naar een gezamenlijke data vault verstuurd.
- Vanuit hier kan enterprise brede analyse worden uitgevoerd

27-9-