Leertraject BI

## **BI in vogelvlucht**

Business Intelligence (BI) omvat alles wat te maken heeft met het verzamelen, opslaan, en verwerken (= analyseren en rapporteren) van data. Tegenwoordig hebben de meeste bedrijven hun data opgeslagen in de operationele ERP-systemen (Enterprise Resource Planning). De databases van deze systemen kunnen bestaan uit duizenden tabellen die soms miljoenen rows aan data hebben. Daarnaast hebben sommige bedrijven ook nog CRM (Customer relationship management) systemen, waarin data staat opgeslagen over klanten, maar bijvoorbeeld ook over de miljoenen mails die het bedrijf heeft verstuurd naar potentiële klanten. Doordat de opslag van data steeds goedkoper wordt, computers steeds sneller en krachtiger worden, en de software om deze data effectief op te slaan en te analyseren steeds beter wordt, worden bedrijven zich steeds bewuster van de kansen van goed datamanagement. Het wordt geschat dat bedrijven die hun data effectief gebruiken daar een output en productiviteitsverhoging van rond de 5% aan overhouden. Dit kan bij grote bedrijven dus om miljarden gaan. Een BI-er moet dus op een effectieve manier met gigantische hoeveelheden data om weten te gaan en deze data om weten te zetten in informatie.

**Datawarehousing**

Het is mogelijk om bovenop ERP-systemen analyses uit te voeren en rapportages te maken, maar dit heeft nadelen. Aangezien tabellen miljoenen rows kunnen hebben, er constant binnen het systeem transacties worden verwerkt, en voor rapportages vaak data uit tientallen tabellen moet worden gecombineerd, kan data uit deze databases halen erg belastend zijn voor het systeem. Ook hebben sommige bedrijven meerdere databases, zodat data uit verschillende systemen moet worden gecombineerd om analyses te kunnen maken. Bedrijven kiezen dan ook vaak voor een datawarehouse, een database met alle data binnen het bedrijf die nodig is voor analyses en rapportages. Binnen het datawarehouse wordt data op een intuitieve en efficiënte manier opgeslagen, zodat het analyseren van de juiste data makkelijker is. Verder kan binnen een datawarehouse de datakwaliteit beter worden gecontroleerd, en kan onnodige data eruit worden gefilterd.

**Meestgebruikte programma’s** : SSMS, SSIS

**Rapportages**

Om rapportages te maken heb je data nodig uit een ERP systeem of datawarehouse. Deze data vraag je op met queries: stukjes codes die de engine vertellen welke data moet worden opgehaald en hoe deze data getransformeerd moet worden. Als je bijvoorbeeld je totale omzet per land wil weten, staat dat niet zomaar in de database. Wat er wel in staat, zijn miljoenen rijen aan facturen waar ook een totaal bij zit. Door die totalen bij elkaar op te sommen en te groeperen op land krijg je dan het gewenste resultaat. Afhankelijk van de complexiteit van de business logica van het bedrijf en de behoeftes van de klant kunnen queries een paar regels tot een paar honderd regels lang zijn. De query voer je in bij Rapportagesoftware zoals SSRS of Power BI. Binnen de rapportagesoftware heb je veel soorten visualisaties, van tabellen tot wereldkaarten. Om data goed te rapporteren is soms licht programmeerwerk nodig, maar als de query goed in elkaar zit moeten vaak alleen nog aan de juiste elementen de juiste data meegegeven worden en parameters ingesteld worden.

**Meestgebruikte programma’s** : SSMS, SSRS, Power BI

**Analyse en Data mining**

<Yet to come>

**Meestgebruikte programma’s: SSAS, Disco, ProM, R, etc**

**Facts, dimensions en relationele theorie**

Stel je hebt een tabel in je database met sales data. Elke rij in het tabel is een transactie waarbij een product is verkocht. Een tabel met sales data is een **fact table**: Elke rij in een fact tablebeschrijft **een gebeurtenis.** Elke kolom beschrijft iets over deze transactie. Centraal staat de kwantitieve data: de **facts, of measures**. Voorbeelden van facts zijn: kwantiteit verkocht, TotaalBedrag (excl btw), Totaalbedrag (incl btw), kostprijs, etc.

**Dimensions**

Dimensions beantwoorden het wie, wat, waar, wanneer, waarom, en hoe van de gebeurtenissen. Een transactie kan honderden dimensies met zich meebrengen: het product dat verkocht is, in welke winkel, datum en tijdstip van de transactie, maar ook dingen als de kleur van het product, de dag in de week van de transactie, hoe er betaald is, of de klant man of vrouw is (of niet bekend), hoeveel het product weegt, etc. Uiteindelijk is het analyseren van data vaak zo simpel als een of een paar dimensies uitkiezen en de facts over deze dimensies te laten zien.

**Relationele theorie: het combineren van facts en dimensions**

Als elke fact table al deze informatie zou moeten tonen, zou je gigantische tabellen krijgen. Opslag wordt tegenwoordig steeds goedkoper, maar een tabel van miljoenen rijen en honderden kolommen is nog steeds te groot. Verder zou zo’n tabel ook enorm onoverzichtelijk zijn. Ook heb je sommige kolommen helemaal niet nodig voor bepaalde rapportages. In plaats van alles in een tabel te stoppen, kiezen we ervoor om dimensions die ‘bij elkaar horen’ samen in een dimensietabel te stoppen. Zo heb je bijvoorbeeld een date dimension, een customer dimension, een product/item dimension, etc. Omdat je dimensies in aparte tabellen opslaat, hoef je bijvoorbeeld ook maar één row in de dimension table te updaten als een aspect van een item verandert, in plaats van miljoenen rows in een fact table.

### Voorbeeld van een Item dimension



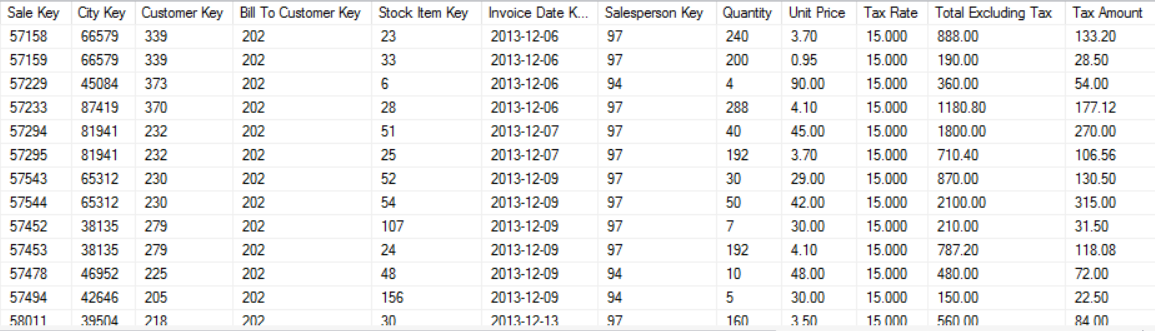
Elke unieke combinatie van kolommen in een dimensietabel krijgt zijn eigen rij, en ook zijn eigen Primary Key. Een Primary Key is er om de rij uniek te identificeren. De key van de dimensietabel geeft dus toegang tot elke dimensie binnen dat tabel. Door een Foreign Key in de fact table te plaatsen die verwijst naar de Primary Key van de dimension, krijgt de fact table ook toegang tot de dimensions. Het is belangrijk dat de Key voor de rest geen enkele betekenis heeft: een naam van een product zal dus ook nooit een key zijn. Dit komt omdat namen en andere aspecten kunnen veranderen, wat lastig kan zijn in sommige situaties.

### City Dimension



**Foreign Keys**

Fact tables gebruiken foreign keys die verwijzen naar de primary keys van dimensietabellen om de dimensies aan de facts te linken. Het onderstaande plaatje laat een deel van een Fact\_Sale tabel zien. De Sale Key is geen foreign key, maar wordt hier gebruikt om elke transactie in de fact tabel uniek te identificeren (dit is niet strict noodzakelijk). Daarna komen de verschillende foreign keys: City, Customer, Bill-To Customer, Item, Invoice Date, en Salesperson. Vervolgens komen de facts, zoals Quantity en Total Excluding Tax. Als we voor een analyse nu dus de totale verkopen per land willen zien, moeten we de tabel **joinen** met de City dimension om uit die tabel Country te halen. Op joins komen we terug bij de SQL syntax.



SQL Server Management Studio (SSMS)

SQL Server Management Studio wordt gebruikt om servers te onderhouden, nieuwe databases, tabellen, en andere objecten te creëren, te doorzoeken, en queries uit te voeren/testen op de database. Om met SSMS om te kunnen gaan moet je iets weten van de interface, maar vooral goed zijn in Structured Query Language (SQL), de taal om data uit databases te halen en te transformeren, en ook nieuwe objecten in een database aan te maken.

## Interface

Voor nu is het plaatje op de volgende pagina genoeg om de belangrijkste dingen over de interface te weten.

## **SQL**

Voor deze sectie heb je de database van WideWorld Importers (OLAP) nodig. Deze kun je downloaden met de volgende link:

<https://github.com/Microsoft/sql-server-samples/releases/tag/wide-world-importers-v1.0>

Ook kun je deze introductiecursussen volgen:

Pluralsight / edx

NOTES:

Settings: Line numbers instellen + prevent saving changes that require table re-creation uitzetten

**Bij Keys beginnen bij -0**

### Select statements

Centraal in SQL zijn de **select statements**. Hiermee haal je data op. Het format van een simpele select statement zit er zo uit:

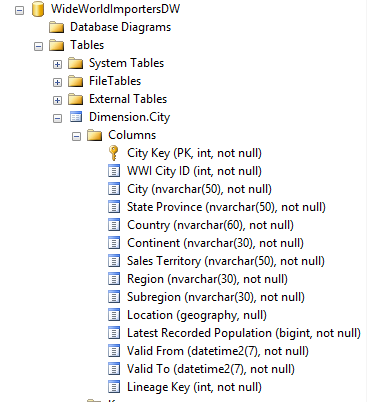
SELECT <Top x > <distinct> <tablename.>columnname1

, <tablename.>columnname2 ..

FROM databasename.schema.tablename <;>

(alles tussen < > is optioneel)

Ingevuld ziet dat er bijvoorbeeld zo uit:

FROM WideWorldImportersDW.Dimension.City

In de resultset van deze query zie je dat veel plaatsen, zoals Carrolton, meer dan een keer in de VS voorkomen. We hebben dus nog een stukje meer informatie nodig. In de object explorer kun je de tabel opzoeken en zie je dat State Province ook als kolom voorkomt. Om deze te selecteren moet je wel rekening houden met de spatie. Door die spatie ziet de engine de naam niet direct als één geheel. Door blokhaken [ ] eromheen te zetten los je dit op. Blokhaken zijn ook vaak nodig als de naam verward kan worden met sql syntax (FROM, WHERE, etc)

SELECT City.City

,City.[State Province]

,City.Country

FROM WideWorldImportersDW.Dimension.City

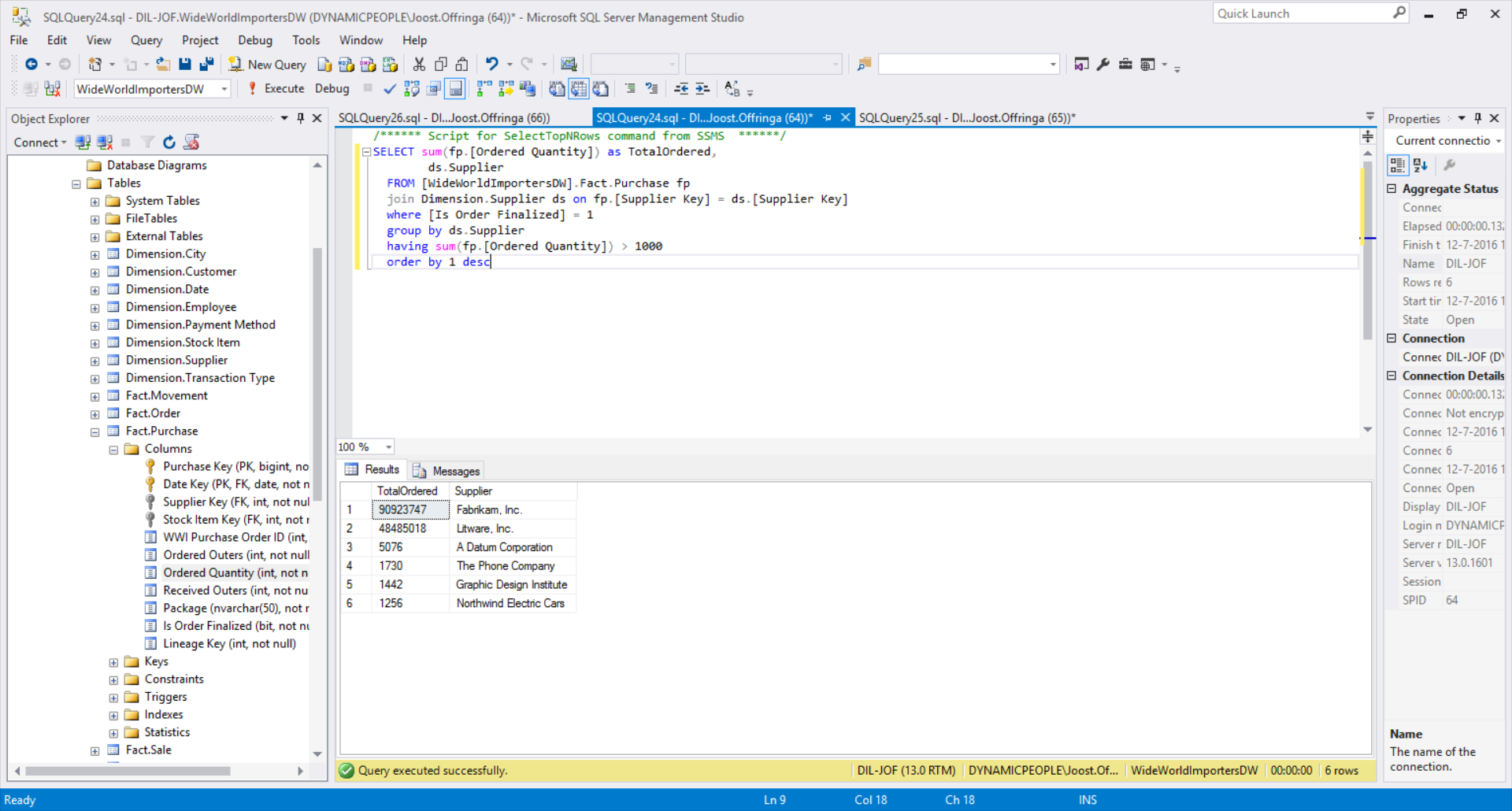
Met **distinct** vind je alleen alle unieke combinaties van de kolommen die je selecteert. Met **Top x** selecteer je alleen de eerste x rijen in plaats van alles.

Voer de huidige query uit

De Query editor gebruik je om queries te bouwen

Nieuwe query

De database waar je in werkt



De query results komen hier, en in het gele balkje eronder zie je info als execution time, en aantal rows die de query heeft opgeleverd.

Met de object explorer kun je verbinding maken met servers en de inrichting ervan verkennen.

Om alle kolommen in een database te selecteren gebruik je een sterretje ‘\*’ . Verder is het goed om op te merken dat het meestal niet per se nodig is om de tablename bij de columnname te zetten. Dit is alleen strict nodig als data uit meerdere tabellen wordt geselecteerd, en dezelfde kolomnaam in meerdere tabellen voorkomt. Sowieso is het overzichtelijker om de tabelnaam erbij te zetten als er uit meerdere tabellen wordt geselecteerd (zie ‘joins’). Ook de de databasenaam is niet nodig zolang de juiste database is geselecteerd in het vakje linksboven (zie vorige pagina).

### Where clause

Om data te zoeken met bepaalde eigenschappen gebruik je de where clause. Kolommen die je in de where clause gebruikt hoeven niet per se in de select statement te zitten. In de where clause vergelijk je dus een kolom met een bepaalde eigenschap. Hiervoor heb je de volgende **comparison operators** (zie data types voor een verdere uitleg van Used for) :

|  |  |  |
| --- | --- | --- |
| Comparison Operators | | Used for |
| = | Equal | Numeric, date and string |
| > | Greater than | Mostly numeric and date |
| < | Less than | Mostly numeric and date |
| >= | Greater than or equal to | Mostly numeric and date |
| <= | Less than or equal to | Mostly numeric and date |
| <> or != | Not equal to | Numeric, date and string |
| (NOT) LIKE | String comparison test | String only |
| (NOT) IN | Multiple equals | Numeric, date and string |
| (NOT) BETWEEN | Between | Mostly numeric and date |

Bijvoorbeeld:

SELECT \* FROM [WideWorldImportersDW].[Fact].[Stock Holding]

where [Quantity On Hand] > 300000

**Bij vergelijkingen met strings (tekst) en datums moet je enkele quotes om de string zetten** ('')

SELECT [Supplier]

,[Category]

,[Supplier Reference]

FROM [WideWorldImportersDW].[Dimension].[Supplier]

where Category = 'Novelty Goods Supplier'

SELECT \* FROM [WideWorldImportersDW].[Dimension].[Supplier]

where [Valid To] > '2016-08-30'

Als je meerdere where clauses hebt, onderscheid je ze met ‘and’. Als maar een van de clauses waar hoeft te zijn, gebruik je ‘or’. PAS OP: zet clausules met ‘or’ erin altijd tussen haakjes, of er kunnen onverwachte resultaten komen.

SELECT [Supplier]

,[Category]

,[Supplier Reference]

FROM [WideWorldImportersDW].[Dimension].[Supplier]

where ( Category = 'Novelty Goods Supplier'

or [Payment Days] > 14 )

Met LIKE kun je een deel van de hele string terugvinden. Daarbij kun je wildcards (%) toevoegen die aangeven of er karakters voor of na het zoekdeel kunnen komen. Er is ook een tweede wilcard (\_) die middenin een string aangeeft dat het welke character dan ook kan zijn. NOT LIKE doet precies wat je denkt dat het doet.

where Employee LIKE 'han%' vindt namen van employees die beginnen met han (Hanne)

where Employee NOT LIKE '%han' vindt namen van employees die niet eindigen op han (Chan)

where Employee LIKE '%han%' vindt employees met een naam met ergens ‘han’ erin (Johanna, maar ook Hanne en Chan)

where Employee LIKE '%h\_n%' vindt employees met een naam met ergens ‘h\_n’ erin (Johanna, Henk, Attila de Hun, maar niet Hein)

IN en BETWEEN zijn minder belangrijk dan de rest, maar worden voor de volledigheid hier ook genoemd. IN laat je meerdere Equals-vergelijkingen met OR ertussen efficiënter opschrijven.

Vergelijk de resultaten van deze twee queries:

SELECT \* FROM [WideWorldImportersDW].[Dimension].[Supplier]

where ( [Primary Contact] = 'Bill Lawson'

OR [Primary Contact] = 'Hai Dam'

OR [Primary Contact] = 'Hubert Helms' )

SELECT \* FROM [WideWorldImportersDW].[Dimension].[Supplier]

where [Primary Contact] IN ( 'Bill Lawson' , 'Hai Dam' , 'Hubert Helms' )

BETWEEN doet precies wat je denkt dat het doet. Wel is belangrijk dat het de bounds meeneemt (het is dus een >= en <= vergelijking, niet > en < ). Vergelijk de volgende twee queries:

SELECT \* FROM [WideWorldImportersDW].[Dimension].[Supplier]

where [Supplier Key] => 5 and [Supplier Key] <= 10

SELECT \* FROM [WideWorldImportersDW].[Dimension].[Supplier]

where [Supplier Key] BETWEEN 5 and 10

### Comments en highlighten

Met twee streepjes (--) commentarieer je een regel uit: deze hele regel wordt dus niet meegenomen bij het uitvoeren van de query. Een comment van meerdere lines kun je ook tussen /\* en \*/ zetten.

-- Dit is een comment

/\* Dit is een comment

van meerdere lines \*/

Of

CTRL + K + C (comment)

CTRL + K + U (uncomment)

Door een deel van de query te highlighten voer je alleen de gehighlighte text uit

### Opdrachten

1) Selecteer de supplier key, supplier naam, en categorie van elke supplier (in dimension.supplier) die meer dan 14 dagen betaaltermijn geeft. Highlight daarna alles behalve de where clause en zie wat er gebeurt.

2) Voeg een where clause aan de vorige query toe zodat alleen Suppliers worden getoond die eindigen op “Inc.” Zet er commentaar bij dat je alleen bedrijven wil zien die Incorporated zijn.

3) De vorige twee queries leveren dubbele records op. Selecteer eerst alle kolommen (laat de where clause staan) en bekijk hoe dit komt. Los het probleem op met een of twee extra where clauses.

4) Los het probleem van vraag 3 op zonder extra where clauses.

5) Je collega heeft net een lelijke aanvaring gehad met een contactpersoon van een supplier, maar is helaas zijn naam vergeten. Hij weet alleen nog dat de voor- **of** achternaam met een M begon. Probeer met een query de personen die aan het profiel voldoen te traceren. (Hint: een achternaam begint na een spatie)

6) Je collega weet ook nog dat het over de betaaltermijn van 30 dagen ging. Vul de vorige query aan en identificeer de juiste persoon.

7) Check extra goed of je query van vraag 6 klopt. Zo ja, dan staan er haakjes om de vergelijkingen met or. Haal dan de haakjes weg en zie wat er gebeurt. Waarom zijn de haakjes zo belangrijk?

### Aggregates

Ga naar de fact.Sale en selecteer alles ervan. Je zult zien dat deze table meer dan 200.000 rows heeft, die allemaal een verkoopgebeurtenis beschrijven. En dit is nog een relatief kleine fact table. Met aggregates zorgen we ervoor dat we de data kunnen samenvatten. De belangrijkste aggregates:

|  |  |
| --- | --- |
| Aggregate | Description |
| AVG() | Returns the average value |
| COUNT() | Returns the number of rows |
| MAX() | Returns the largest value |
| MIN() | Returns the smallest value |
| SUM() | Returns the sum |

Om de waarde van alle transacties (excl. tax) bij elkaar te zien doe je dus:

SELECT SUM

FROM [WideWorldImportersDW].[Fact].[Sale]

En voor de waarde van de gemiddelde verkoop:

SELECT AVG([Total Excluding Tax])

FROM [WideWorldImportersDW].[Fact].[Sale]

#### Group by

Om dimensies toe te voegen aan aggregates moeten we meer doen dan alleen die dimensies toe te voegen aan de select statement. De volgende query geeft namelijk een error:

SELECT SUM([Total Excluding Tax])

, Package

FROM [WideWorldImportersDW].[Fact].[Sale]

Om de totale omzet per verpakkingsmethode te zien moeten we de computer ook laten weten waar we op willen groeperen. Normaal zijn dat **alle kolommen die we selecteren behalve de aggregates zelf**. Om te groeperen gebruiken we ‘GROUP BY’. Deze komt na de WHERE clause.

SELECT SUM([Total Excluding Tax])

, Package

FROM [WideWorldImportersDW].[Fact].[Sale]

Group By Package

Bij het volgende voorbeeld berekenen we de gemiddelde grootte van een plaats (binnen de city dimension) per State in de VS. Als je in de table kijkt zul je zien dat bij sommige plaatsen een population van 0 staat; waarschijnlijk omdat de population niet bekend is. Deze records willen we uitsluiten:

SELECT avg([Latest Recorded Population])

, Country

, [State Province]

FROM [WideWorldImportersDW].[Dimension].[City]

where [Latest Recorded Population] > 0

group by Country, [State Province]

#### Having

Stel je wil alleen de staten zien met een gemiddeld inwoneraantal per plaats van meer dan 10.000 . Aggregates kunnen helaas niet in de where clause: door where clauses te evalueren voordat berekeningen worden geëvalueerd kunnen bij voorbaat al veel rows worden uitgesloten, waardoor queries sneller worden. In plaats van gebruiken we HAVING, direct na de group by:

SELECT avg([Latest Recorded Population])

, Country

, [State Province]

FROM [WideWorldImportersDW].[Dimension].[City]

where [Latest Recorded Population] > 0

group by Country, [State Province]

having avg([Latest Recorded Population]) > 10000

### Order by

Je resultaten sorteren is niet alleen handig, maar kan ook de resultaten van je query beïnvloeden (bij top x bijvoorbeeld, of windowing functions). Voor sorteren gebruiken we ORDER BY. Je kunt ook strings en datums sorteren. By default sorteert de computer ascending (van laag naar hoog). Door ‘desc’ toe te voegen sorteer je de andere kant op. Het is wel netjes om ook ‘asc’ toe te voegen waar het hoort.

Verder kun je sorteren op:

* De kolomnaam ( order by [State Province] asc )
* Een expressie of aggregate: (order by avg([Latest Recorded Population]) desc )
* Het kolomnummer: hoeveelste kolom in de select statement het is ( order by 1 desc)

Voorbeeld:

SELECT avg([Latest Recorded Population])

, Country

, [State Province]

FROM [WideWorldImportersDW].[Dimension].[City]

where [Latest Recorded Population] > 0

group by Country, [State Province]

having avg([Latest Recorded Population]) > 10000

order by 1 desc

### Opdrachten

8) Vind het gemiddelde aantal dagen dat je hebt om te betalen per supplier category.

9) Filter de query van vraag 8 zodat je alleen Categorieën krijgt met “Services” erin.

10) Vind in de fact.Purchase hoeveel transacties er zijn geweest per Package soort.

11) Vind in de fact.[Stock Holding] hoeveel items er op dit moment in totaal op voorraad zijn.

12) Vind in de fact.Movement alle dagen tussen 1 januari en 1 juni dit jaar waarop er in totaal meer spullen uitgingen dan binnenkwamen. Sorteer de resultaten van het meest eruit tot het minst eruit.

13) Vind in de fact.Sale de totaal verkochte hoeveelheid en totale winst per item (gebruik description). Laat alleen DBA joke mug items zien. Toon de meest verkochte items bovenaan.

14) Vind de totale winst per product per jaar (= year([Invoice Date Key])) . Sorteer op jaar ( oplopend) en op winst (meeste winst bovenaan).

### Joins

Zoals je hebt gelezen in de intro over facts, dimensions, en relational theory, is het efficiënter om dimensies apart in dimensietabellen op te slaan en dan met een key de fact tables toegang te geven tot deze dimensions. Dit doen we met joins. Joins komen tussen de from en de where clause. Na de join kunnen we alle kolommen van beide tabellen gebruiken alsof het één grote tabel is.

General format van een join:

select [column a], [column b], ..

FROM table a

<LEFT/RIGHT> <INNER/OUTER/CROSS> JOIN Table b

on [table a].[column a] = [table b].[column a]

< and [table a].[column c] = [table b].[column c] >

Voorbeeld van een join:

SELECT \*

FROM [WideWorldImportersDW].[Fact].[Sale]

inner join dimension.city

ON city.[City Key] = sale.[City Key]

Na een join kunnen we dus ook kolommen van de city dimension in de where clause gebruiken (of waar dan ook)

SELECT \*

FROM [WideWorldImportersDW].[Fact].[Sale]

inner join dimension.city

ON city.[City Key] = sale.[City Key]

where city.[State Province] = 'Alabama'

**De ON geeft aan welke kolommen met elkaar moeten worden gematched om rows binnen de tabellen met elkaar te joinen.** On is als de WHERE clause voor joins: er zit een vergelijking in (meestal =, maar zie NON EQUI JOIN) en er kunnen ook meerdere on clauses zijn, waar je AND tussen zet.

Overigens kun je ook joinen met een where clause. Normaal is een expliciete join met on beter, maar in sommige gevallen kan dit niet gebruikt worden.

SELECT \*

FROM [WideWorldImportersDW].[Fact].[Sale], Dimension.City

where fact.sale.[City Key] = Dimension.City.[City Key]

Voor het begrijpen van joins zijn de volgende dingen belangrijk:

* Relations
  + Many to one
  + One to many
  + Many to many
* Joins
  + INNER JOIN
  + LEFT OUTER JOIN
  + RIGHT OUTER JOIN
  + FULL OUTER JOIN
  + CROSS JOIN
  + NON EQUI JOIN

**Relations**

**Many to one**

De fact.Sale table heeft een foreign key naar dimension.Customer. Een foreign key kan meerdere keren in de fact.Sale voorkomen: elke keer als een klant wat koopt komt er een row bij met de Customer Key van die klant. In de Dimension.Customer komt dezelfde Key echter maar één keer voor als Primary Key: de Customer Key is daar de manier om een rij uniek te identificeren. Als je de fact.Sale dus joint met een dimension is het altijd duidelijk naar welke rij in de dimension de foreign key van de fact table verwijst. Het aantal rijen blijft ook gelijk na een join. Dit is de meest gebruikte en ook meest veilige soort join.

*(in alle voorbeeldtekeningen wordt gejoind op de nummers)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | a |  | 1 |  | 1 | a | 1 |
| 1 | b | JOIN | 2 | = | 1 | b | 1 |
| 2 | a |  |  |  | 2 | a | 2 |

Vergelijk het aantal rows van:

select \* from fact.sale

met

select \* from fact.sale

inner join Dimension.Customer

on Customer.[Customer Key] = sale.[Customer Key]

**One to many**

Bekijk het nu van de andere kant: je hebt een dimensietabel (dimension.Customer) en je joint daarop een fact table (fact.Sale) . Waar je eerst één rij per Customer Key had, wordt nu bij elke customer key alle transacties getoond. Je oorspronkelijke tabel dijt uit van een paar honderd rows tot een paar honderdduizend rows. Deze **opdubbeling** van data is niet altijd de bedoeling, en het is dan ook belangrijk dat hier voorzichtig mee om wordt gegaan.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  | 1 | a |  | 1 | 1 | a |
| 2 | JOIN | 1 | b | = | 1 | 1 | b |
|  |  | 2 | a |  | 2 | 2 | a |

**Many to many**

In beide tabellen is de kolom (of de combinatie van kolommen) waarop wordt gejoind niet uniek. Dit kan leiden tot een explosieve groei van het aantal rows. Many to many kan erg gevaarlijk zijn. Probeer voor de grap maar eens deze query (en cancel ‘m na niet te lang met de rode vierkante stopknop, want hij zal erg lang duren en uiteindelijk honderden miljoenen rows produceren). Kijk dus altijd extra uit als je joins niet many-to-one (of one-to-one) zijn, zodat het aantal rows met een join niet groter wordt.

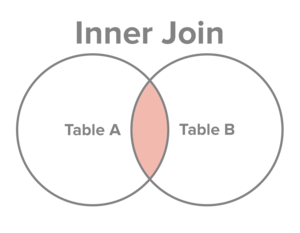
select \* from fact.sale

inner join fact.Purchase

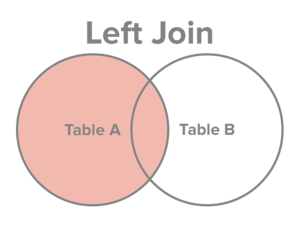
on sale.[Stock Item Key] = purchase.[Stock Item Key]

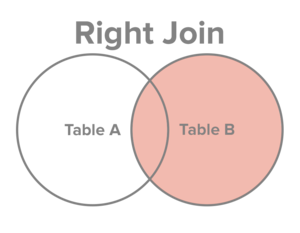
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | c |  | 1 | a |  | 1 | c | 1 | a |
| 1 | d | JOIN | 1 | b | = | 1 | c | 1 | b |
| 2 | c |  | 2 | a |  | 1 | d | 1 | a |
|  |  |  | 2 | b |  | 1 | d | 1 | b |
|  |  |  |  |  |  | 2 | c | 2 | a |
|  |  |  |  |  |  | 2 | c | 2 | b |

**Soort Joins**

**<Inner> Join (over het algmeen het snelst)**

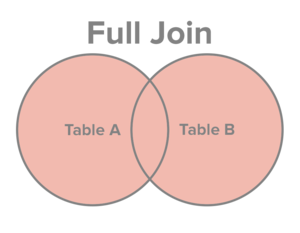
Bij een inner join worden alleen de rows meegenomen waarbij de kolommen van de ON-condition gematched zijn. Als iemand die op een of andere manier in de customer dimension staat, maar nog nooit iets gekocht heeft, zal deze rij dus verdwijnen als er een join is met de fact.Sale.

**Left <outer> join**

Hierbij wordt de gehele inhoud van de linker tabel (degene na de FROM) geheel getoond, of er nou een match is of niet. In het geval een row geen match heeft met de rechtertabel worden de kolommen van de rechtertabel op NULL gezet (meer over NULL bij data types en constraints).

**Right <outer> join**

Het tegenovergestelde van een left outer join: de rechter tabel wordt altijd geheel getoond.

**Full <outer> join**

Alle rows worden hoe dan ook getoond, of er nou een match is met de andere tabel of niet. Net zoals bij de andere outer joins worden kolommen die hierdoor geen waarde hebben (vanwege geen match) op NULL gezet.

**Cross join**

Voor deze join heb je geen ON nodig: elke row van de ene tabel wordt namelijk gecombineerd met alle rows van de andere tabel. Het totale aantal rows van het resultaat is dus de aantal rows van table 1 vermenigvuldigd met het aantal rows van table 2. Deze join wordt wel eens gebruikt voor het opbouwen van dimension tables.

**Non-equi join**

Dit is geen aparte join op zichzelf, maar een normale (meestal inner) join waarbij er geen = maar een ongelijke operator wordt gebruikt ( < , > , <= , >= ) . Vaak wordt deze gecombineerd met nog een join condition, maar dit hoeft niet.

### Aliases

**Column aliases**

Als je een query met een aggregate hebt uitgevoerd is het je vast opgevallen dat de resulterende kolom de naam (no column name) had. Om dit aan te passen kun je na de definitie van de kolom AS zetten met een naam erachter. Dit kan met elke soort kolom, ook als ze al een naam hebben. Let wel op dat je met WHERE en HAVING clauses nog steeds naar de originele definitie van de kolom moet refereren (dus niet de alias). In de ORDER BY kun je wel naar de alias verwijzen. Overigens is het niet per se nodig om het woordje AS te gebruiken: een alias meegeven kan ook door de alias direct achter de definitie van de kolom te zetten. AS neerzetten maakt je code wel overzichtelijker.

SELECT SUM([Total Excluding Tax]) as TotalSales

, Package

FROM [WideWorldImportersDW].[Fact].[Sale]

Group By Package

having SUM([Total Excluding Tax]) > 2000000

order by TotalSales desc

**Table aliases**

Het is ook mogelijk om tables aliassen mee te geven. Als je dit doet kun je wel meteen in alles naar de alias verwijzen. Sterker nog: de originele table name zal niet meer werken! Table aliases zijn niet alleen handig, maar soms ook nodig (zie subqueries, self-joins, en het voorbeeld hierna)

SELECT fs.[Total Excluding Tax], dc.Customer from fact.sale as fs

inner join Dimension.Customer as dc

on fs.[Customer Key] = dc.[Customer Key]

Soms heb je in een een tabel meerdere kolommen die verwijzen naar dezelfde dimension. Al deze kolommen wil je dus kunnen joinen met dezelfde dimension. In fact.Sale heb je bijvoorbeeld een Customer Key en een Bill To Customer Key, die iets anders betekenen maar wel allebei naar Dimension.Customer verwijzen. Als we beide kolommen met dezelfde Dimension.Customer joinen ontstaat er echter een probleem: als we een kolom van die Dimension selecteren, kan de engine niet bepalen vanuit welke join de kolom geselecteerd moet worden. Aliassen bieden hierbij uitkomst:

SELECT

fs.[Customer Key]

,dc.Customer as [Customer Name]

,dc.[Primary Contact] as [Customer Contact Person]

,fs.[Bill to Customer Key]

,dcBill.Customer as [Bill to Customer Name]

,dcBill.[Primary Contact] as [Bill to Customer Contact Person]

,fs.[Total Excluding Tax]

FROM [WideWorldImportersDW].[Fact].[Sale] as fs

inner join Dimension.Customer as dc on fs.[Customer Key] = dc.[Customer Key]

inner join Dimension.Customer as dcBill on fs.[Bill To Customer Key] = dcBill.[Customer Key]

WHERE fs.[Customer Key] <> 0

### Opdrachten

Omdat het moeilijk is om joins goed te begrijpen met grote tables, moet je voor deze opdracht eerst de twee queries in Appendix 1 uitvoeren. Deze creëren een kleine fact\_Sales en fact\_Customer table. Refresh de tables map in de object explorer (right click -> refresh) om de nieuwe tabellen te zien. Ook wil je misschien ctrl+shift+r drukken zodat de computer ook in de query editor direct de tables herkent (zie Intellisense).

15) Selecteer alles van fact\_Sales, dan van dim\_Customer en bestudeer de inhoud. Bedenk bij jezelf wat je verwacht bij een full outer join, left join, right join, en inner join. Maak daarna een query waarbij je alles van beide tabellen selecteert. Voer een voor een de verschillende joins uit en bekijk de resultaten.

16) Selecteer in plaats van alles alleen de Customer\_Key, FirstName, TransactionDate\_Key, en Total. Vul bij de kolomnamen niet de table name in. Wat voor error krijg je? Hoe komt dit?

17) Probeer een cross join. Hoeveel rows verwacht je als resultaat?

18) Je wil zien hoeveel aan spullen al je klanten in je systeem in totaal besteed hebben. Je wilt dus ook de mensen zien die niks gekocht hebben. Bedenk de juiste soort join en laat alleen FirstName, LastName, en het totaal zien (met naam: Total Bought). Sorteer verder op het totaal.

19) Waarschijnlijk heb je op dit moment gegroepeerd op FirstName en LastName. Dit gaat nu OK, maar er kunnen meerdere Piet Zwarts op deze wereld zijn. Hoe kun je voorkomen dat deze personen worden samengevoegd in de query?

Voer de volgende query uit:

insert into dbo.Dim\_Customer (FirstName , LastName, Birthday, CustomerCategory, DistanceFromStore, Gender)

values( 'Piet' , 'Zwart' , '1979-12-12' , 'None' , 5, 'M')

20) Voer je query van vraag 19 nogmaals uit en check of je oplossing heeft gewerkt (er moeten dus twee Piet Zwarts zijn)

21) Je wil nu alleen de mensen zien die in totaal meer dan 10 spullen hebben gekocht. Pas de query aan.

22) We willen nu weten hoeveel er per maand in totaal het verkocht is. Join op Dimension.Date en sekecteer daarvan de Calendar Month Label. Bedenk weer wat de juiste soort join is gezien de informatie die je wil krijgen. Voeg daarna ook de totale Quantity aan de spullen die zijn gekocht toe. Noem deze kolom “Qty Sold” en hernoem de totaalkolom “Total Sold”. Sorteer eerst op Qty Sold, en daarna op Total sold (allebei van meest naar minst). Inner Join of Right Join?

23) Vergelijk:

SELECT \*

FROM [WideWorldImportersDW].[dbo].[Dim\_Customer]

inner join fact\_Sales on 1=1

met

SELECT \*

FROM [WideWorldImportersDW].[dbo].[Dim\_Customer]

cross join fact\_Sales

en verder:

SELECT \*

FROM [WideWorldImportersDW].[dbo].[Dim\_Customer]

cross join fact\_Sales

where fact\_Sales.Customer\_Key = Dim\_Customer.Customer\_Key

met

SELECT \*

FROM [WideWorldImportersDW].[dbo].[Dim\_Customer]

inner join fact\_Sales

on fact\_Sales.Customer\_Key = Dim\_Customer.Customer\_Key

Wat kun je concluderen over inner joins en cross joins?

Als de key een complete match heeft dan is het resultaat van beide joins hetzelfde.

### Belangrijke Functies

#### Wiskundige functies

Je kunt in SQL natuurlijk ook optellen, aftrekken, vermenigvuldigen, en delen.

|  |  |
| --- | --- |
| Operator | Omschrijving |
| + | addition / **concatenation** |
| - | subtraction |
| \* | multiplication |
| / | division |
| % | modulo |
| SQRT(x) | Square root of x |
| Power(x,y) | xy |

Alle wiskundige functies worden per row apart geëvalueerd. Kijk uit met divisions: de query zal failen als er door 0 wordt gedeeld. Voorkom dit met case when of met safedivide functie (zie hieronder).

SELECT \* , Total / Quantity as AveragePrice

FROM [WideWorldImportersDW].[dbo].[fact\_Sales]

#### Concatenation

Het + teken gebruiken we ook om Strings aan elkaar te plakken. Dit heet concatenation. Het volgende voorbeeld plakt FirstName en LastName aan elkaar (met een spatie ertussen).

SELECT FirstName + ' ' + LastName as FullName

FROM [WideWorldImportersDW].[dbo].Dim\_Customer

#### Case when .. end

De “if” –functie van SQL. Case when geeft een waarde terug gebaseerd op een conditie. Case when kan in de SELECT statement worden gebruikt, maar ook in de WHERE, GROUP BY, ORDER BY, en HAVING. De syntax:

case when (condition) then (result)

when (2nd condition) then (result) ..

else (result)

end < as (Name) >

De functie evalueert de eerste condition voor elke row. Als die waar is geeft het een resultaat. Als de condition niet waar is gaat de functie naar de volgende when tot het een resultaat heeft. Als er een resultaat is gevonen gaat de functie door naar de volgende row in de table.

SELECT \*,

case when Quantity < 3 then 'Small Quantity'

when Quantity >=3 AND Quantity < 6 then 'Medium Quantity'

else 'Huge Quantity' end as QuantitySize

FROM [WideWorldImportersDW].[dbo].[fact\_Sales]

Na de when of de then kun je ook formules gebruiken. Verder kun je case when ook zelf binnen andere formules gebruiken, of in aggregates.

SELECT sum(case when dc.Gender = 'F' then fs.Total else 0 end) as TotalFemaleSales

,sum(case when dc.Gender = 'M' then fs.Total else 0 end) as TotalMaleSales

from fact\_Sales as fs

join Dim\_Customer as dc on dc.Customer\_Key = fs.Customer\_Key

#### dbo.safedivide

dbo.safedivide(Numerator, Divisor) is een zelfgemaakte functie die ervoor zorgt dat je veilig kan delen. Om deze functie te gebruiken moet je eerst op de database de functie creëren (zie appendix).

#### Date functions

Voor date functions heb je vaak dateparts nodig. De belangrijkste staan in het tabel hieronder. In formules kun je zowel de volledige naam van de datepart of de afkorting gebruiken.

|  |  |
| --- | --- |
| datepart | Abbreviations |
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |

**GETDATE()** geeft de datum van vandaag.

**DATEPART(datepart, date)** geeft de datepart van een bepaalde datum.

select datepart(yy, GetDate())

select datepart(DayOfYear, '2016-12-31')

Formules als **Year(date)**, **Month(date)** en **Day(date)** geven hetzelfde resultaat als de DATEPART formule met de betreffende datepart.

Zie ook: <https://msdn.microsoft.com/en-us/library/ms174420.aspx>

**DATEADD (datepart , number , date )** laat je dateparts optellen bij een datum.

select Dateadd(year, 5, Getdate() ) as TodayInFiveYears

select Dateadd(mm, -20, Getdate() ) as TodayTwentyMonthsAgo

Zie ook: <https://msdn.microsoft.com/en-us/library/ms186819.aspx>

**DATEDIFF ( datepart , startdate , enddate )** geeft het verschil tussen twee datums in de datepart die je opgeeft.

select Datediff(dd, birthday, Getdate() ) as DaysOld

, FirstName + ' ' + LastName as Name

from Dim\_Customer

order by DaysLived desc

Zie ook: <https://msdn.microsoft.com/en-us/library/ms189794.aspx>

#### String functions

**LTRIM(string)** en **RTRIM (string)** halen extra spaties aan het begin en eind van strings weg.

select Ltrim(' teveel spies ') geeft: 'teveel spaties '

select Rtrim(' teveel spaties ') geeft: ' teveel spaties'

**Upper(string)** en **Lower(string)** geven een string in respectievelijk ALL CAPS en kleine letters.

**SUBSTRING (string ,start , length )** geeft een deel van een string terug gebaseerd op de beginplaats en de lengte die je opgeeft. Het 1e karakter van een string krijgt 1, enz. Ook spaties tellen mee voor de lengte.

select Substring('Een voorbeeld' , 2 , 7) geeft: 'en voor'

zie ook: <https://msdn.microsoft.com/en-us/library/ms187748.aspx>

**LEFT(string, length)** is een SUBSTRING met start = 1

**RIGHT(string, length)** is het tegenovergestelde van left, en begint dus vanaf de andere kant.

**LEN(String)** geeft de lengte van een string

**CHARINDEX ( text to find , string )** vindt de positie van een stukje text binnen een string. Als de string niet gevonden wordt, evalueert de expressie naar 0.

select CHARINDEX('beeld', 'Een voorbeeld') geeft: 9

zie ook: <https://msdn.microsoft.com/en-us/library/ms186323.aspx>

**PATINDEX (text to find, string)** werkt als charindex maar gebruikt wildcards zoals LIKE (% en \_)

zie ook: <https://msdn.microsoft.com/en-us/library/ms188395.aspx>

### Opdrachten

24) Probeer SUBSTRING() een paar keer uit op de kolom Employee van Dimension.Employee .

25) De kolom Employee laat de voor- en achternaam van de employee zien. Er is ook een kolom die de voornaam laat zien (Preferred Name), maar achternaam staat er niet apart in. We gaan dit maken. Vind eerst de plek van de eerste spatie in Employee met CHARINDEX().

26) Gebruik nu het resultaat van de CHARINDEX om de starting position van de SUBSTRING te bepalen om de achternaam uit de Employee kolom te halen. Gebruik als lengte 50, dat moet genoeg zijn om alle achternamen volledig mee te nemen. Noem de kolom LastName.

27) Het resultaat klopt bijna, maar het lijkt alsof er een spatie voor achternaam zit. Je zou die eventueel met LTRIM() kunnen oplossen, maar het is eleganter om in de substring de starting position te veranderen. Die begint nu met de spatie. Je SUBSTRING moet dus 1 plek later beginnen. Doe dit met een simpele wiskundige functie.

28) Selecteer de SUM en de AVG van de [Total Excluding Tax], een derde kolom waarin je de SUM door de AVG deelt, en een 4e kolom die alle rijen telt in fact.Sale. Kijk of de 3e en 4e kolom gelijk zijn.

29) Selecteer City, [State Province], en Country van Dimension.City. Maak ook een kolom aan die die drie velden achter elkaar laat zien met een komma en spatie ertussen. Geef de kolom ook een naam.

30) Selecteer alles in Dim\_Customer en maak een extra kolom gebaseerd op DistanceFromStore:

* Als dit 5 of minder is, is het Loopafstand
* Van 6 tot 20 is het Fietsafstand
* Meer dan 20 betekent Lange afstand.

31) Je wilt een kortingsactie gebaseerd op de Customercategory:

* Voor 'None' of mensen die niet in het systeem staan is er geen korting.
* Voor Bronze is er 5% korting
* Voor Silver 10%
* Voor Gold 15%

Bereken hoeveel korting de klanten bij elke transactie hadden gekregen als de regeling al in plaats was geweest. Noem de kolom TotalDiscount. Maak daarna ook een kolom TotalWithDiscount aan. (Begin met de join van Fact\_Sales naar Dim\_Customer om de boel overzichtelijk te houden)

32) Je geeft klanten twee maanden de tijd na de verkoop om te bepalen. Bepaal in de fact\_Sales de uiterste betaaldatum.

33) Voor elk ding dat een klant koopt in een transactie krijgt hij of zij nog 2 dagen uitstel. Herbereken de uiterste betaaldatum.

34) Schrijf voor jezelf uit in stappen hoe je iemands leeftijd bepaalt. Probeer daarna met GETDATE(), DATEDIFF, DATEPART en een case when op de dag nauwkeurig de leeftijden van klanten in Dim\_Customer te berekenen.

35) Maak drie leeftijdsgroepen: 0 tot 30, 31 tot 50, en 50+ers. Laat daarna de totale verkopen zien per leeftijdscategorie én geslacht.

### Data types

Tot nu toe zijn we van drie soorten data types uitgegaan: strings, numbers, en dates. Voor de meeste situaties hoef je niet veel meer te weten, maar in bepaalde situaties (tables creeëren, data van verschillende types vergelijken en converten) kun je niet zonder gedetailleerde kennis. Data type is een eigenschap van een kolom: elke rij binnen de kolom heeft dus hetzelfde datatype. Verder kunnen kolommen eigenschappen en constraint hebben, die soms ook afhankelijk zijn van het data type. De dikgedrukte types hieronder zijn het belangrijkst om te kennen.

#### Numerieke data types

Voor nummers heb je verschillende opties, beschreven in het tabel hieronder

|  |  |
| --- | --- |
| Data type | Omschrijving |
| **int** | integer (heel nummer). Voor ints die heel groot kunnen worden (meer dan 2 miljard) moet je bigint gebruiken. Eventueel kun je voor kleine integers smallint of tinyint gebruiken om ruimte te besparen. |
| **decimal(n,m)**  **numeric(n,m)** | decimaal nummer met maximaal n tekens voor de komma en m tekens na de komma [decimal(10, 2)]. Decimal en numeric doen precies hetzelfde. |
| money | voor geldwaarden, al werkt decimal of numeric meestal ook. |
| bit | 0 of 1, kan gebruikt worden voor True/False waarden. |
| float | Een vreemd datatype: de computer bepaalt hierbij hoe de ruimte voor het nummer wordt verdeeld voor en na de komma. Is vaak niet precies. Je kunt dit type tegenkomen, maar het is zo goed als altijd beter om decimal of numeric te gebruiken. |

#### Character string data types

|  |  |
| --- | --- |
| Data type | Omschrijving |
| nchar(n) | Strings met lengte van precies n. |
| **nvarchar(n)** | Strings met variabele lengte van maximaal n. Voor hele lange strings kun je varchar(MAX) gebruiken. |
| ntext | een varchar met veel nadelen en weinig voordelen. Zo kun je bijvoorbeeld geen distinct selectie op text doen. Net zoals float zul je m af en toe tegenkomen maar zo goed als nooit zelf gebruiken. |

Naast nchar (en nvarchar, etc) heb je ook char (zonder n ervoor). Het verschil tussen een nchar en char is dat nchar een unicode string is: hiermee kun je tekens van vreemde talen er ook in krijgen. Nchars kosten ook meer ruimte, dus gebruik het alleen als het bedrijf internationaal georienteerd is.

#### Date data types

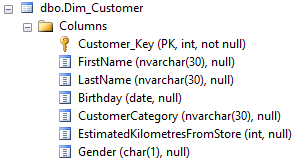
|  |  |
| --- | --- |
| Data type | Omschrijving |
| **date** | datum in YYYY-MM-DD format |
| time | tijd in hh:mm:ss:[ms] format |
| **datetime** | datum+tijd in één kolom. De GETDATE() functie geeft een datetime terug. |

Er zijn verder ook nog data types zoals image, hierarchyid, en uniqueidentifier, maar deze zijn niet belangrijk genoeg om hier uitgebreid te behandelen. Het enige wat we hier noemen is dat uniqueidentifiers soms worden gebruikt als primary key, maar veel meer ruimte innemen dan simpele int, waardoor deze laatste bijna altijd liever wordt gebruikt.

#### Implicit and explicit conversions

Normaal gesproken kunnen alleen kolommen van dezelfde data type met elkaar worden vergeleken. SQL maakt het ons echter makkelijk en kijkt bij een vergelijking met verschillende data types altijd eerst of het een van de twee kolommen kan converten naar een andere data type. Probeer maar eens de volgende query: select \* from Dim\_Customer where 1 = '1'

Deze query runt gewoon terwijl het ene nummer een int (of numeric) is en de ander een (var)char.

Als je select getdate() runt zie je dat de computer een datetime teruggeeft. De kolom Birthday in Dim\_Customer is een date, zoals je in de object explorer kan checken. Toch runt de volgende query zonder problemen:

select \* from dim\_customer where Birthday <Getdate()

Soms lukt het de computer echter niet om waardes te converten, of gebruikt het een implicit convert met onjuiste resultaten, wat vervelende gevolgen kan hebben. Daarom is het aan te raden om zo vaak mogelijk explicit converts uit te voeren. Daarmee geef je aan naar welke data type je een kolom wil converten.

Met **CONVERT(data type, expression)** maak je expliciete converts. Bijvoorbeeld:

select convert(nvarchar(2), [DistanceFromStore]) as KMFromStore from dim\_customer

Soms moet je uitkijken met converts. Met een conversion van een decimal naar een int gaat bijvoorbeeld data verloren:

select convert(int, 32.18) geeft: 32

Met convert kun je natuurlijk geen conversion doen die onmogelijk zijn.

select convert(int, FirstName) from dim\_customer

levert bijvoorbeeld een error op: Msg 245, Level 16, State 1, Line 2

Conversion failed when converting the nvarchar value 'Piet' to data type int.

Zo goed als alle data types kun je converten naar een nvarchar, ook als je niet weet wat

**CAST( expression AS data type)** doet hetzelfde als convert.

zie ook: <https://msdn.microsoft.com/en-us/library/ms187928.aspx>

### Tables

#### Create table

Om een table te creëren kun je twee dingen doen: een sql command of right-click op de map in de object explorer waar de table in moet komen -> new -> table. In het laatste geval kun je bij de column properties onderaan en table properties rechts alle extra properties en constraints meegeven die je zo gaat leren. Om tables makkelijker te begrijpen is het beter om te beginnen met de sql commands. Zie Appendix 1 voor een voorbeeld van een create table statement.

Create table <schema name.>[table name] (

[column1 name] [data type] <constraints>,

[column2 name] [data type] <constraints>, ...

<Constraint [contraint name] [constraint type] ([column 1], [column 2]> )

Probeer zelf maar eens eens een table te maken. Eventueel kun je als schema dbo neerzetten, maar dit is niet nodig. Vergeet bij data types niet de lengte op te geven (zoals bij decimals of nvarchars). Negeer voor nu even de constraints (die zijn sowieso altijd optioneel).

#### Kolomeigenschappen en constraints

* Not Null
* Primary Key
* Unique
* Identity
* Check
* Default

**NULL values**

Als er voor rij binnen een kolom geen data beschikbaar is, wordt de value in dat veld Null. Null values zijn niet altijd gewenst en kunnen ook voor problemen zorgen. Kijk bijvoorbeeld naar het case when voorbeeld van het vorige hoofdstuk:

case when Quantity < 3 then 'Small Quantity'

when Quantity >=3 AND Quantity < 6 then 'Medium Quantity'

else 'Huge Quantity' end as QuantitySize

Een null value zal hier onder de ‘else’ vallen en dus Huge Quantity als QuantitySize meekrijgen. We kunnen testen of iets Null is met where [column] is <not> null

We kunnen dan de case when van hierboven aanvullen met :

when Quantity is null then 'No data available'

We kunnen ook een kolom de constraint NOT NULL meegeven, waardoor inserts van data in de table zullen falen als er een null value in die kolom wordt

FirstName nvarchar(30) not null

**Primary Key**

Zoals al genoemd zorgt een primary key ervoor dat een rij in een table uniek identificeerbaar is. Naast één kolom kan ook een combinatie van kolommen een primary key zijn. Binnen een primary key kunnen velden niet NULL zijn.

[Customer\_Key] int primary key

Als je een primary key over meerdere kolommen wil aanmaken, moet je in de onderste line met constraints neerzetten: constraint PKName PRIMARY KEY ([column 1], [column 2]

**Unique**

Net zoals primary keys kunnen er geen duplicates voorkomen in een unique column, maar velden kunnen wel NULL zijn. Verder horen foreign keys niet naar unique columns te verwijzen.

**Identity**

Een identity is een eigenschap van een nummer (normaal een int, maar decimal kan eventueel ook) die het makkelijker maakt om primary keys te maken. Voor elke nieuwe row die binnen komt genereert de identity column automatisch een nieuw nummer dat het vorige nummer +1 is. Normaal begint een identity op 1 en doet het steeds +1 per nieuwe row, maar je kunt de “seed” en “increment” ook zelf instellen: identity(5,2) begint bijvoorbeeld op 5 en gaat steeds 2 omhoog.

[Customer\_Key] int identity primary key ,

**Check constraint**

Een check constraint beperkt de waarden van een kolom. Zo kun je bijvoorbeeld een aparte customer table aanmaken voor jonge mensen, en dan bij de definitie van kolom Age neerzetten:

Age int CHECK(Age <= 30) ,

**Default**

Geeft een default waarde mee als de kolom niet gevuld wordt.

Birthday date NOT NULL default('1900-01-01')

**Een voorbeeld van een create table met constraints:**

CREATE TABLE [dbo].[Dim\_YoungCloseCustomers](

[YoungCloseCustomer\_Key] int identity primary key,

[FirstName] [nvarchar](30) not null,

[LastName] [nvarchar](30) ,

[Birthday] [date] not null default('1900-01-01'),

[CustomerCategory] nvarchar(30) ,

[DistanceFromStore] [int] CHECK( [DistanceFromStore] < 50),

[Gender] [char](1),

[Age] int CHECK(Age <= 30),

constraint UniquePeople Unique (FirstName , LastName, Birthday) ,

constraint Genders CHECK(Gender IN ('F', 'M' , 'U'))

)

#### Inserting into tables

Natuurlijk is het ook mogelijk om data in tables in te voeren. Dit doen we met insert. General format:

insert into tablename (<column1> , <column2>, ..)

values ( row1value1 , row1value2, ..),

( row2value1 , row2value2, ..),

..

In de appendix kun je een ingevuld voorbeeld vinden, die je al gebruikt hebt om de kleine fact en dimensietables voor de vorige opdrachten te populeren. Als je waarden in een tabel wil invullen moet je na de tablename tussen haakjes de kolommen zetten waarin je data wil invoeren. Als een kolom wel in het tabel staat maar niet wordt genoemd bij de insert, komt er automatisch null in te staan. Als die kolom een not null constraint heeft krijg je dus een error. Voor de rest is het belangrijk dat je identity columns nooit in een insert statement noemt: door de identity krijgt deze kolom vanzelf een waarde.

Ook is het mogelijk om data van een andere table naar een nieuwe table te kopiëren.

insert into tablename (<column1> , <column2>, ..)

select ( <column1> , <column2>, ..)

FROM <tablename2>

#### Truncate

Om een table leeg te maken gebruik je truncate. Truancate reset ook de seed van de identity columns binnen de table.

truncate table tablename

#### Delete from

Om records te verwijderen gebaseerd op een bepaald criterium gebruik je delete from:

delete from tablename where (condition)

delete from Dim\_Customer2 where CustomerCategory is null

Als jouw table een primary key heeft is het makkelijker om een row uniek te identificeren. Keys zijn dus ook handig voor deletes (en updates, zie volgende alinea). LET OP: als je de where clause vergeet bij een delete from doet het hetzelfde als truncate!

#### Update

Om data te updaten gebruik je update, en net zoals bij delete kun je met de where clause aangeven welke records moeten worden aangepast.

update tablename set column = (expression) < where (condition) >

update Dim\_Customer2 set DistanceFromStore = DistanceFromStore + 5 where Customer\_Key IN ( 4, 7)

#### Drop table

Om een table inclusief alle data erin te verwijderen gebruik je drop:

drop table tablename

set

#### Opdrachten

36) Maak een leeg kopietje van de dbo.dim\_Customer op de volgende manier: rightclick op dim\_Customer in de object explorer -> script table as -> create to -> New query editor window. Bekijk of je de code snapt, en verander dan de naam naar dim\_Customer2 en voer het script uit. Refresh de tables folder waar de nieuwe table in zit en check dat er geen data in zit.

37) Maak nu een insert statement waarbij je alle data van Dim\_Customer in Dim\_Customer2 gooit, behalve DistanceFromStore. (Denk eraan dat je identities niet moet inserten). Bekijk wat voor waarde DistanceFromStore krijgt in Dim\_Customer2

38) Voer nu de ontbrekende data in met een update (TIP: na de’ =’ komt een select statement die in de where clause de Dim\_Custoupdatemer en Dim\_Customer2 joint. Deze hele select moet tussen haakjes staan om de where clause van de select niet te verwarren met die van de update)

39) Kopieer van de vorige pagina de create table van YoungCloseCustomers en voer die uit. Insert nu de data van Dim\_Customer erin, en zorg er ook voor dat Age er goed inkomt (gebruik dezelfde berekening als bij opdracht 29). Zorg er verder voor dat alleen de data erin komt die aan de check constraints voldoet (vooral die van Age en DistanceFromStore), omdat er anders een error wordt gegeven.

40) Voer ook manueel een nieuwe regel in voor deze table. Probeer daarbij ook een regel in te voeren die niet aan de constraints voldoet om te zien wat er gebeurt.

41) In een gekke bui heb je bedacht dat je een kortingsactie gaat opzetten voor de YoungCloseCustomers. De procentuele korting is gelijk aan DistanceFromStore / Age \* 100 . Bedenk wat voor soort data types je door elkaar moet delen en berekent het antwoord op 2 decimalen nauwkeurig.

42) De korting lijkt wel erg hoog te worden zo. Verander de berekening zodat kortingen van boven de 35% worden verlaagd tot 35%.

43) Maak een query die totale quantity sold en totale revenue per sales territory laat zien. Het resultaat van deze query willen we graag in een nieuw tabel zetten, maar het kost redelijk wat tijd om deze te maken. In plaats daarvan vullen we de volgende regel in vóór de FROM:

into fact.[SalesByTerritory]

Voer nu deze query uit en refresh de tables map in de object explorer. Bekijk de nieuwe fact table en zie wat voor data types en constaints die heeft meegegeven. Waarom heeft Sales Territory de constraint not null meegekregen?

### Overige belangrijke technieken

#### Subqueries

Subqueries zijn queries die binnen een andere query worden gebruikt. Normaal gaat het om select statements. Subqueries kunnen bijvoorbeeld handig zijn als je data van tables wil bewerken voordat je ze joint, of als je maar een paar kolommen nodig hebt van een table en de query sneller wil maken. Ook zijn subqueries soms de enige manier om een query te laten werken (zie bijvoorbeeld de where clause bij Windowing Functions)

Normaal gebruik je subqueries in de from, maar het kan ook in de select zelf (al is dit meestal onhandig). Subqueries zet je tussen haakjes en, als je ze gebruikt in de select of from, geef je een naam zoals je een alias geeft aan een table. Alle kolommen die je selecteert binnen de subquery krijgen de table name van de naam van de subquery. Voorbeeld:

select DiCu.FirstName, DiCu.LastName , SumSales.QuantitySold

from

(select FirstName , LastName , Customer\_Key from Dim\_Customer ) as DiCu

inner join

(select Customer\_Key, sum(Quantity) as QuantitySold from fact\_Sales group by Customer\_Key having sum(Quantity) > 10 ) as SumSales

on DiCu.Customer\_Key = SumSales.Customer\_Key

Subqueries kun je ook gebruiken in de where clause. Subqueries zijn vooral handig in exists of not exists clauses. Deze geven aan of een row met de aangegeven waarden bestaat in een table. Op basis daarvan kun je dan data in een andere table filteren. Zo kun je bijvoorbeeld je YoungCloseCustomers uitsluiten door:

SELECT \*

from Dim\_Customer dcc

where not exists ( select \* from Dim\_YoungCloseCustomers dycc

where dycc.FirstName = dcc.FirstName

and dycc.LastName = dcc.LastName

and dycc.Birthday = dcc.Birthday )

In de where clause van de subquery wordt aangegeven hoe de records van de subquery moeten worden gematched met de records van de main query (zoals een join). Op basis hiervan worden de records uit Dim\_Customer gefilterd. Bedenk hierbij dat Dim\_YoungCloseCustomers slecht in elkaar zit: in deze table zou dezelfde Customer\_Key als in Dim\_Customer moeten voorkomen.

#### Windowing functions

Windowing functions gebruik je om rankings te geven aan je resultaten. Daarbij heb je drie opties:

* Row\_Number() telt altijd door, ook als er records met gelijke waarde tussen zitten.
* Rank() geeft records met gelijke waardes dezelfde rank, en slaat daarna de ranks over die de records bij row\_number zouden hebben gekregen.
* Dense\_Rank() geeft records met gelijke waardes dezelfde rank, maar telt daarna gewoon door.

Waar de ranks op gebaseerd zijn zet je in een order by, zoals in het format hieronder:

ROW\_NUMBER() over (<partition by [column] > order by [Column] <desc/asc>)

Voer de volgende query uit om de syntax van windowing functions en het verschil tussen de drie duidelijker te maken:

SELECT DistanceFromStore,

ROW\_NUMBER() over (order by [DistanceFromStore] desc) as DistanceRowNumber,

RANK() over (order by [DistanceFromStore] desc) as DistanceRank,

DENSE\_RANK() over (order by [DistanceFromStore] desc) as DistanceDenseRank

FROM [WideWorldImportersDW].[dbo].[Dim\_Customer]

Het resultaat:

|  |  |  |  |
| --- | --- | --- | --- |
| KMFromStore | DistanceRowNumber() | DistanceRank() | DistanceDenseRank() |
| 98 | 1 | 1 | 1 |
| 40 | 2 | 2 | 2 |
| 40 | 3 | 2 | 2 |
| 40 | 4 | 2 | 2 |
| 22 | 5 | 5 | 3 |
| 13 | 6 | 6 | 4 |
| 8 | 7 | 7 | 5 |
| 6 | 8 | 8 | 6 |
| 5 | 9 | 9 | 7 |
| 5 | 10 | 9 | 7 |
| 3 | 11 | 11 | 8 |

Met partitions geef je aan dat je je binnen groepen rankings wil maken, zodat bij elke groep de ranking opnieuw (bij 1) begint.

SELECT Gender,

DistanceFromStore,

ROW\_NUMBER() over (partition by gender order by [DistanceFromStore] desc) as DistanceRowNumber,

RANK() over (partition by gender order by [DistanceFromStore] desc) as DistanceRank,

DENSE\_RANK() over (partition by gender order by [DistanceFromStore] desc) as DistanceDenseRank

FROM [WideWorldImportersDW].[dbo].[Dim\_Customer]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Gender | KMFromStore | DistanceRowNumber | DistanceRank | DistanceDenseRank |
| F | 98 | 1 | 1 | 1 |
| F | 40 | 2 | 2 | 2 |
| F | 6 | 3 | 3 | 3 |
| F | 3 | 4 | 4 | 4 |
| M | 40 | 1 | 1 | 1 |
| M | 40 | 2 | 1 | 1 |
| M | 22 | 3 | 3 | 2 |
| M | 13 | 4 | 4 | 3 |
| M | 8 | 5 | 5 | 4 |
| M | 5 | 6 | 6 | 5 |
| M | 5 | 7 | 6 | 5 |

##### Where clause

Je kunt windowing functions niet in de where clause gebruiken. Als je dat doet krijg je namelijk een error. Dit is onhandig, maar gelukkig is er een workaround. Door een subquery te gebruiken kun je alsnog bepaalde ranks eruit filteren:

SELECT \*

from (select Gender,

DistanceFromStore,

ROW\_NUMBER() over (partition by gender order by [DistanceFromStore] desc) as DistanceRowNumber

FROM [WideWorldImportersDW].[dbo].[Dim\_Customer]) KMRank

where KMRank.DistanceRowNumber <=3

#### Variables

Variables kun je gebruiken om waardes in op te slaan. Deze kun je dan later in een query gebruiken. Hiermee kun je ook dynamische queries maken: queries die reageren op bepaalde omstandigheden door middel van variables. Een variable herken je door een apenstaartje (@). Het is belangrijk om te weten dat variables tijdelijk zijn: zodra de query heeft gerund is de variable al weer verdwenen. General format:

set @[variablename] [type] < = [Defaultvalue] >

Voorbeelden:

declare @Age int

declare @FirstName varchar(30) = 'Harry'

Met set kun je de waarde van een variable aanpassen. Daarbij kun je ook een query gebruiken om een waarde op te halen. Bedenk wel dat die query dan maar één waarde moet ophalen. Ook is het goed om te bedenken dat je in dezelfde query als dat je de variable declaret ook je set moet gebruiken (zonder declare kan de set niks doen omdat de variable alweer verdwenen is).

set @Age = 40

set @FirstName = (select FirstName from Dim\_Customer where Customer\_Key = 5)

Met select @FirstName of print(@FirstName) kun je dan de waarde van je variable controleren (al is dit bijna nooit nodig).

##### Gebruik in queries

Je kunt variables overal binnen queries gebruiken. Normaal gebeurt dat vooral binnen de where clause, maar je kunt een variable dus ook bijvoorbeeld in de select gebruiken.

declare @MinDistance int

, @MaxDistance int

, @Gender nchar(1)

set @MinDistance = 8

set @MaxDistance = 30

set @Gender = 'M'

select \*

from Dim\_Customer

where DistanceFromStore between @MinDistance and @MaxDistance

and Gender = @Gender

##### Table variables

Normale variables kunnen maar één waarde in zich hebben, maar je hebt bijvoorbeeld ook table variables: een geheel tabel dat in een variable is opgeslagen. Deze variabelen gedragen zich echter niet echt als variablen, maar meet als tijdelijke tabellen. Zo werkt set bijvoorbeeld niet, maar moet je gewoon inserten:

declare @CustomerTableNames table( FirstName varchar(50), LastName varchar(50));

insert into @CustomerTableNames select FirstName, LastName from Dim\_Customer

select \* from @CustomerTableNames

##### Executing SQL code within variables

Je kunt ook SQL statements in variables opslaan. Hiervoor gebruiken we normaal varchar(max). Je kunt bijvoorbeeld een select statement in een variabele opslaan. Met exec(@[variablename]) voeren we daarna de code binnen de variabele uit:

declare @sql varchar(max)

set @sql = 'select \* from Dim\_Customer'

exec(@sql)

Door middel van concatenation kunnen we ook andere variables binnen de sql variabele gebruiken. Soms is dit echter een heel gedoe, vanwege data types en vanwege single quotes:

Stel je hebt een integer variabele @MinKM die aangeeft hoe ver klanten op zijn minst van de winkel moeten wonen om geselecteerd te worden. Je wil deze variabele concatenaten met de rest van de select statement als volgt:

declare @sql nvarchar(MAX)

, @MinKM int = 10

set @sql = 'select \* from Dim\_Customer where DistanceFromStore >= ' + @MinKM

exec(@sql)

Het uitvoeren van de variabele geeft echter de volgende error omdat het eerste deel van de variabele een nvarchar is en het tweede deel een int (en de engine probeert de nvarchar naar int te converten):

Msg 245, Level 16, State 1, Line 24

Conversion failed when converting the varchar value 'select \* from Dim\_Customer where DistanceFromStore >= ' to data type int.

Om dit op te lossen kunnen we beter de @MinKM variable als nvarchar invoeren. Als deze later wordt vergeleken met de kolom DistanceFromStore gaat de implicit convert naar int wel goed:

exec(@sql)

declare @sql nvarchar(MAX)

, @MinKM nv

printarchar(10) = '10'

set @sql = 'select \* from Dim\_Customer where DistanceFromStore >= ' + @MinKM

exec(@sql)

Als je de @sql string bekijkt (met print(@sql) of select @sql ) zie je dat de select statement klopt, en ook dat er om de 10 geen single quotes staan. Deze verdwijnen namelijk als de variable geevalueerd wordt. In een geval van een int is dit wenselijk, maar bedenk dat nvarchars etc. tussen single quotes moeten blijven staan. Hoe doen we dat? De oplossing blijkt om aan de beide kanten van de variable drie single quotes en een + te zetten: ''' + @[variable]+''' . Ook moet dit hele verhaal binnen de single quotes van de rest van de @sql string staan!

declare @Gender nchar(1) = 'F'

declare @sql varchar(max)

set @sql = 'select \* from Dim\_Customer where Gender = ''' + @Gender+''''

print(@sql)

exec(@sql)

Bekijk naast de resultaten ook het resultaat van de print(@sql) in de messages tab.

##### Global variables

Global variables heb je niet vaak nodig, maar als je ze wel nodig hebt is het vaak in een stored procedure. Global variables herken je aan de dubbele apenstaartjes (@@) . Eigenlijk zul je maar één global variable af en toe gebruiken (@@ROWCOUNT in stored procedures). Global variables kunnen snel van waarde veranderen en kun je dus vaak beter opslaan in normale variables als dat nodig is.

@@ROWCOUNT geeft het aantal rows dat gegenereerd is door de laatst uitgevoerde query. Voorbeeld:

select \* from fact.Sale

select @@ROWCOUNT

--of:

declare @NumberofRows int

select \* from fact.Sale

set @NumberofRows = @@ROWCOUNT

select @NumberofRows

#### Stored procedures

Stored procedures zijn een speciaal soort opgeslagen queries. Je vindt stored procedures onder Programmability -> Stored procedures. Stored procedures kunnen bijvoorbeeld een select statement bevatten. Ook kun je variabeles meegeven aan stored procedures. Deze noemen we dan parameters omdat ze een vaste waarden hebben.

##### Creating and executing procedures

Het general format voor het creëren van een stored procedure ziet er zo uit:

create procedure <procedurename>

AS

BEGIN

*CODE HIER*

END

Een simpele procedure die alle data van Dim\_Customer ophaalt ziet er dan bijvoorbeeld zo uit:

create procedure sp\_GetCustomers

AS

BEGIN

select \* from dbo.Dim\_Customer

END

Voer deze code uit en voer daarna de procedure uit met de volgende code:

execute sp\_getCustomers

Je kunt de procedure ook uitvoeren door in de map van stored procedures (refresh hem eerst als je net een nieuwe procedure hebt gemaakt) rechts te klikken op de juiste procedure -> Execute Stored Procedure -> OK.

##### Parameters in Stored Procedures

Om parameters (=variables) mee te geven aan stored procedure moet je dit direct na de naam van de parameters die gebruikt worden aangeven. Daarna kun je deze in de query erna gebruiken. General format:

create procedure [ProcedureName]

@[Parameter1] [datatype],

@[Parameter2] [datatype],

..

AS

BEGIN

..

END

Ingevuld:

create procedure sp\_GetCustomersByDistanceAndGender

@MinDistance int,

@MaxDistance int,

@Gender nchar(1)

AS

BEGIN

select \* from dbo.Dim\_Customer

where DistanceFromStore between @MinDistance and @MaxDistance

and Gender = @Gender

END

Bij het uitvoeren van zo’n procedure moet je dan ook die parameters invullen. Dit kan op drie manieren.

1. Je geeft de waardes mee in de volgorde dat de parameters waren opgegeven:

exec sp\_GetCustomersByDistanceAndGender 2, 20, 'F'

1. Je geeft expliciet aan welke parameters welke waarden moeten krijgen:

exec sp\_GetCustomersByDistanceAndGender @Gender = 'M' , @MinDistance = 18, @MaxDistance = 40

1. Je klikt rechts op de stored procedure, klikt op execute, en vult de parameters in de dialog box in. Daarna genereert de engine de juiste code vanzelf. Gebruik deze optie alleen als de andere twee op een of andere manier niet handig zijn.

##### While loops

Loops zijn onmisbaar bij het maken van dynamische SQL. In SQL heb je alleen de while loop. Bij de while loop zet je een condition neer. Als de condition waar is (bijvoorbeeld 5 < 10 ), voert de engine de code tussen BEGIN en END uit. Als de engine bij END is gekomen kijkt ie of de condition nog steeds waar is. Als dat zo is, dan begint ie weer bij BEGIN enzovoorts. General format:

while (condition)

BEGIN

(SOME CODE / QUERY)

(SOMETHING that changes the condition)

END

Als je condition in het begin waar is en tussen BEGIN en END niet iets staat wat de condition verandert, dan blijft de loop eeuwig doorgaan. Kijk dus uit met while loops!

Voorbeeld van eeuwige loop (je kunt hem afbreken met de rode vierkante stopknop) :

declare @loopvariable int = 1;

while @loopvariable < 10

BEGIN

select @loopvariable as [MOOI NUMMER]

END

Om uit een loop te komen moet je in dit geval dus zorgen dat de @loopvariable verandert, bijvoorbeeld zo:

declare @loopvariable int = 1;

while @loopvariable < 10

BEGIN

select @loopvariable as [MOOI NUMMER]

set @loopvariable = @loopvariable +1

END

Na een loop kun je gewoon doorgaan met verdere sql code te schrijven:

declare @loopvariable int = 1;

while @loopvariable < 10

BEGIN

select @loopvariable as [MOOI NUMMER]

set @loopvariable = @loopvariable +1

END

select 'DE LOOP IS AFGELOPEN!' as MELDING

##### Sys views

Elke database heeft zijn eigen sys views. Deze views (= een soort table) geven informatie over objecten, instellingen, en nog veel meer binnen de server. De sys views die je het meest gaat gebruiken zijn:

* sys.columns laat alle kolommen binnen de database zien. sys.columns.name geeft de naam van de kolom.

sys.tables laat alle tables binnen de database zien. Je kunt sys.tables joinen met sys.columns op object\_id: select \* from sys.columns a

join sys.tables b

on a.object\_id = b.object\_id

sys.tables.name geeft de naam van de table.

* sys.schemas heb je af en toe nodig om de volledige naam van een table te bouwen. sys.schemas join je met sys.tables op schema\_id

Door binnen stored procedures gebruik te maken van de data binnen sys views kun je makkelijker de juiste informatie bij elkaar krijgen om je database te onderhouden. Een voorbeeld hiervan maken we in de opdracht.

#### Opdrachten

Van opdracht 44 tot 49 maken we een stored procedure die een overzicht maakt van elke table van de database en hoeveel rows er in zitten.

44) Maak een nieuwe table (dbo.TableData) aan met twee kolommen: TableName (nvarchar) en NumberOfRows (int) .

45) Bepaal het aantal rijen in sys.tables en stop het resultaat in de variable maxrows. Maak ook een variable genaamd tablename (varchar(100)), een variable genaamd loopvariable (int, default = 1) en een variable genaamd sql (varchar(MAX)) aan.

46) Maak een while loop aan met als condition dat de loopvariable kleiner dan of gelijk aan maxrows is. Binnen deze loop moet het volgende gebeuren:

46.1) Maak uit sys.tables en sys.schemas een kolom die schema name en table name combineert (denk daarbij aan hoe je normaal een table name inclusief schema in de FROM zet). Extra hint: zet blokhaken om de table name aangezien sommige tables een spatie in de naam hebben. Voeg ook een ranking toe. De ranking mag gebaseerd zijn op elke willekeurige kolom in de sys.tables, maar zorg er wel voor dat je een windowing function gebruikt die geen dubbele rankings kan geven of ranks kan overslaan.

46.2) Zet de variable tablename gelijk aan het resultaat van deze query, met als where clause dat de ranking gelijk is aan de variable loopvariable.

46.3) Nu het meest tricky gedeelte: Vul de variable @sql met een select statement, waarbij je twee kolommen selecteert: de eerste is table name (= @tablename), en de volgende is een total row count van deze @tablename. Let goed op de single quotes!

46.4) stop in dbo.TableData de execute van de @sql variable.

46.5) Doe de loopvariable +1 zodat de volgende table binnen sys.tables aan de beurt is en eindig daarna de loop.

47) Selecteer alles van dbo.TableData en sorteer van meeste rows naar minste rows.

48) Probeer de hele query uit en zie of die werkt (dbo.TableData moet gevuld zijn met meerdere tables waarvan de grootste honderdduizenden rows heeft en de kleine 0). Als dit zo is, zetten we de create procedure eromheen. Noem de procedure [sp\_Refresh\_dbo.TableData]. De stored procedure heeft geen parameters: alle variabelen worden binnen de query zelf gevuld. Voordat je deze procedure aanmaakt, moeten we nog een regel aan het begin toevoegen (meteen na de BEGIN van de create proc): truncate table TableData. We willen namelijk dat de procedure de data ververst, en niet oude data in de table houdt.

49) Test de procedure uit door wat nieuwe tabelletjes aan te maken en / of wat rows te verwijderen of aan te maken in de Dim\_Customer en dan de procedure opnieuw te draaien.

50) WideWorldImporters wil meer inzicht in de turnover van hun voorraad. Daarvoor willen ze een overzicht krijgen waarin per product de volgende dingen zien:

* Item naam
* Totaal verkochte hoeveelheid in de laatste maand
* Het aantal stuks op voorraad.
* Het turnover percentage: het aantal verkochte stuks (in de laatste maand) als percentage van het aantal stuks op voorraad.

Als je database outdated is, zoek dan de datum van de laatste transactie op met:

select max([invoice date key]) from fact.sale

en gebruik data als einddatum van je laatste maand.

Als je in de Dimension.Stock Item kijkt zie je dat items meerdere keren in de table voorkomen. Dit komt omdat sommige records als outdated woren gemarkeerd. Sommige regels in de Fact.Sale kunnen echter nog wel naar die outdated records verwijzen, en moeten dus wel worden meegeteld om totale hoeveelheid verkocht te berekenen. Kijk dus goed op welk veld je moet groeperen. Bedenk voor de rest wat voor risico’s dit en het feit dat je twee facttabellen met elkaar gaat joinen met zich meebrengt. Zorg ervoor dat je op meerdere punten controleert of je resultaten nog kloppen en bedenk goed welke query technieken van pas kunnen komen.

# Appendix

Simple customer dimension

create table dbo.Dim\_Customer (

Customer\_Key int identity primary key,

FirstName nvarchar(30),

LastName nvarchar(30),

Birthday date,

CustomerCategory nvarchar(30),

DistanceFromStore int,

Gender nchar(1)

)

insert into dbo.Dim\_Customer (FirstName , LastName, Birthday, CustomerCategory, DistanceFromStore, Gender)

values( 'Piet' , 'Zwart' , '1966-08-12' , 'Gold' , 40, 'M'),

( 'Henk' , 'Peters' , '1989-11-03' , 'None' , 5, 'M'),

( 'Anita' , 'van Veen' , '1976-01-23' , 'None' , 3, 'F'),

( 'Karin' , 'Zwart' , '1964-03-25' , 'Silver' , 40, 'F'),

( 'Mike' , 'Janssen' , '1990-01-08' , 'Bronze' , 40, 'M'),

( 'Harry' , 'Jansen' , '1946-06-06' , 'Bronze' , 22, 'M'),

( 'Jan' , 'Verhoeven' , '2000-09-30' , 'Silver' , 13, 'M'),

( 'Anouk' , 'Visser' , '1979-02-22' , 'Bronze' , 98, 'F'),

( 'Henk' , 'Hoogendoorn' , '1988-04-03' , 'Bronze' , 8, 'M'),

( 'Marit' , 'Markering' , '1992-05-29' , 'Silver' , 6, 'F')

Simple fact table

create table dbo.fact\_Sales (

Customer\_Key int,

TransactionDate\_Key date,

Quantity int,

Total decimal(10,2),

)

insert into dbo.fact\_Sales(Customer\_Key, TransactionDate\_Key, Quantity, Total )

values( 1 , '2016-01-01' , 5, 100.00),

( 8 , '2016-02-01' , 6, 120.12),

( 2 , '2016-03-01' , 2, 41.20),

( 4 , '2016-04-02' , 8, 162.23),

( 5 , '2016-05-02' , 1, 19.75),

( -1 , '2016-06-04' , 5, 98.40),

( 6 , '2016-01-05' , 4, 117.89),

( 7 , '2016-02-05' , 3, 56.09),

( 8 , '2016-03-05' , 5, 85.01),

( 9 , '2016-04-05' , 7, 138.65),

( 1 , '2016-05-06' , 7, 199.90),

( 4 , '2016-06-06' , 2, 21.20),

( 5 , '2016-01-06' , 1, 8.70),

( 8 , '2016-02-06' , 1, 19.30),

( 7 , '2016-03-07' , 10, 221.99),

( 4 , '2016-04-07' , 5, 100.90)

dbo.safedivide

CREATE FUNCTION [dbo].[SafeDivide](

@Numerator decimal(38,19)

,@divisor decimal(38,19) )

RETURNS decimal(38,6)

BEGIN

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- Description:

-- This function divides the first argument by the second argument after

-- checking for NULL or 0 divisors to avoid "divide by zero" errors.

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DECLARE @quotient decimal(38,6);

SET @quotient = null;

IF ( @divisor is not null

AND @divisor <> 0

AND @Numerator is not null )

SET @quotient = @Numerator / @divisor;

RETURN(@quotient)

END