



De GO-database

HAAGSE HOGESCHOOL | ADVANCED DATABASES

SEMESTER 5 | PROJECTGROEP 4

THIERRY PIKET (21159963)

EDWIN ROS (19137052)

LESLEY-ANN DE WIT (21151482)

SANDER IN 'T HOUT (15126463)

Versies

- 1.0 Eerste versie.
- 2.0 Versie ter herkansing. Aangepast zijn de volgende zaken:

Gezamelijke aanpassingen

- Changelog toegevoegd aan het document.
- Complexe query 1 aangepast:
 - Usecase uitgebreider beschreven.
 - Conclusie uitgebreider beschreven.
 - Jaartaal toegevoegd in de query om de resultaten te beperken.
 - Uitgebreid met een CTE.
 - De sortering is nu op kolomnaam.
 - Fout in CASE statements opgelost.
 - Layout aangepast voor leesbaarheid.
- Complexe query 2 aangepast:
 - Usecase uitgebreider beschreven.
 - Conclusie beschreven.
 - Fout in CASE statements opgelost.

Individuele aanpassingen

- Edwin: Voor onderdeel E heb ik een extra [functie](#) (GetRetailerCityCount) toegevoegd met bijbehorende test. De test voor deze functie is een geautomatiseerde test en is [hier](#) te vinden.
- Sander: Gedeelte over testen geschreven (welke geheel absent was in versie 1.0), fout in stuk over functions verbeterd (total_order_value was als CTE geschreven ipv als function).
- Thierry: Voor onderdeel G, nieuw objectsoort, heb ik een extra VIEW toegevoegd, uitgelegd wat het nut hiervan is voor het bedrijf Great Outdoor en getest. Deze is te vinden op pagina 76. SQL script is te vinden op onze [GitHub repository](#).

Inhoudsopgave

Inleiding	4
Technische Analyse	5
COUNTRY	5
SALES_BRANCH	5
ORDER_HEADER	6
RETAILER_SITE	7
RETAILER_TYPE	7
RETAILER	8
ORDER_DETAILS	9
RETURNED_ITEM	10
RETURN_REASON	10
PRODUCT	11
INVENTORY LEVELS	12
PRODUCT_FORECAST	12
CAMPAIGN	13
PROMOTION	14
SALES_STAFF	14
SALES_TARGET	15
ORDER_METHOD	15
PRODUCT_LINE	16
PRODUCT_TYPE	16
Nieuw ERD ontwerp	18
SALES_BRANCH	18
RETAILER_SITE	18
COUNTRY	19
ORDER_HEADER	19
ORDER_DETAILS	19
RETAILER	20
SALES_TARGET	20
Overige opmerkingen en adviezen	20
Nieuwe ERD	21
Queries	22

Complexe Query 1 – Categorieën per retourreden	22
Complexe Query 2 – Hoeveelheid werk per ordermethode	24
System Catalog.....	27
Stored Procedures & Functions	31
Thierry.....	31
Edwin	37
Lesley-Ann.....	43
Sander	47
Tests	52
Thierry.....	52
Edwin	55
Lesley-Ann.....	62
Sander	69
Uitbreiding objectsoort.....	75
Thierry.....	75
Edwin	80
Lesley-Ann.....	84
Sander	87
Conclusie	89
Bibliography	90
Bijlagen.....	93

Inleiding

Dit verslag beschrijft de analyse van de database van Great Outdoor (hierna: GO). Great Outdoor is een wereldwijd actieve groothandel op het gebied van artikelen ten behoeve van buitenrecreatie. Om de activiteiten van GO in goede banen te leiden maken zij gebruik van een SQL-database in Microsoft SQL Server.

Ons team is gevraagd om de SQL-database te onderzoeken op het gebied van efficiëntie en mogelijkheden. We hebben de database op verschillende niveaus bekeken voor dit onderzoek.

Allereerst hebben wij de individuele tabellen onder de loep genomen. We hebben daarbij onderzocht of de tabellen juist genormaliseerd zijn volgens de regels van de 3e normaalvorm. Daarnaast hebben we bekeken of de tabellen wellicht inconsistenties bevatten of andersoortige potentiële problemen. Waar mogelijk hebben wij de normalisatie toegepast, de inconsistenties verbeterd en andere kleine verbeteringen toegepast in een voorstel voor een nieuw fysiek model. De voorgestelde wijzigingen lichten we tevens toe.

Het tweede deel van ons onderzoek bestaat uit het bedenken van nuttige query-mogelijkheden op de oorspronkelijke database. Het uitgangspunt is dat de query's potentieel kunnen bijdragen aan de bedrijfsvoering van GO. Binnen deze query's tonen wij enkele geavanceerde mogelijkheden van SQL.

Het derde deel van ons onderzoek bestaat uit het bepalen van de mogelijkheden van de System Catalog van Microsoft SQL Server en welk voordeel het gebruik van de System Catalog zou kunnen bieden aan GO.

Het vierde deel van het onderzoek bestaat uit het bepalen van de mogelijkheden die Stored Procedures (hierna: SP's) en Functions kunnen bieden. Middels voorbeelden laten we zien welke voordelen SP's en Functions kunnen bieden binnen de bedrijfsvoering. Daarnaast laten we door middel van handmatige tests de werking van de SP's en Functions zien.

Als laatste stellen we vast welke voordelen Views en Triggers binnen de SQL-database kunnen bieden om (respectievelijk) het queryen te vergemakkelijken en enkele soorten wijzigingen in de database te automatiseren om overbodig hand- en/of programmeerwerk te vermijden.

Technische Analyse

Er is ons een database met een ERD-diagram hiervan aangereikt om te herzien. Onze taak ik om te bepalen of alle tabellen de juiste relaties hebben, attributen bezitten die consistent zijn met de SQL-conventies en of de normalisatie van de tabellen in orde zijn. In dit deel van ons onderzoek voeren wij een technisch analyse op de zogenaamde Greatoutdoor (GO) database. Wij zullen onze constateringen zo goed als mogelijk onderbouwen wanneer deze afwijkt van de SQL-standaardisatie.

Om te toetsen of een tabel voldoet aan de regels van de 3^e normalisatievorm is voor een behouden van het overzicht en consistentie in dit document gekozen om een checklist, in tabelvorm, op te nemen. Deze checklist geeft overzichtelijk het antwoord in welke normaalvorm de tabel zich bevindt. Daarna zal bij afwijkingen een onderbouwing worden gegeven. Hiermee geven we aan tot welke normalisatievorm de normalisering correct is uitgevoerd. Per tabel analyseren we ook de relaties met andere tabellen en onderbouwen deze.

COUNTRY

Tabel: COUNTRY		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Ieder niet key attribuut in de tabel is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel	Y

De tabel COUNTRY is correct genormaliseerd volgens de 3^e normaalvorm.

Onderbouwing

Tabel COUNTRY heeft een kolom met een PK te weten COUNTRY_CODE. Kolom COUNTRY om de naam in op te slaan en kolom LANGUAGE voor de taal. De naam van het land maakt de kolom COUNTRY uniek echter de kolom LANGUAGE heeft op elke regel 'EN' staan is dit een fout in de data? Verder vraag ik mij af wat de functionele betekenis is van de kolom LANGUAGE. Als dit de officiële taal van het land is dan is dit niet juist, en hoe zit het met tweetalige landen? De officiële taal kun je dan niet kwijt in deze opzet.

SALES_BRANCH

Tabel: SALES_BRANCH		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y

	Er zijn geen herhalende groepen aanwezig	N
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	N
3	Ieder niet key attribuut in de tabel is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel	N

De tabel SALES_BRANCH_CODE is niet correct genormaliseerd volgens de 3^e normaalvorm.

Onderbouwing

Bij tabel SALES_BRANCH valt op het eerste gezicht op dat de kolommen ADDRESS1 en ADDRESS2 herhalende kolommen zijn. Het opslaan van een herhalende groep op een regel is een overtreding van de 1^{ste} normaalvorm. Deze tabel staat eigenlijk volgens de normalisatieregels in de 0^{de} (0 NF) normaalvorm. Bij nader onderzoek van de data blijkt dat ADDRESS2 een toevoeging is van het adres. De naam is allen dan wat onhandig. ADDRESS en ADDRESS_ADDITION is misschien een betere benaming.

Er valt nog wat op er is nog een tabel (RETAILER_SITE) met veel overeenkomstige kolommen. Dit is niet direct een overtreding van één van de normalisatie regels maar er zijn vijf kolommen met dezelfde naam.

Wat als er een Sales branch bijkomt in dezelfde plaats? Dan krijgen we voor iedere regio en POSTAL_ZONE een herhaling. Deze tabel voldoet niet aan de 1e normalvorm en staat daarom in de 0 de normaalvorm.

ORDER_HEADER

Tabel: ORDER_HEADER		
Normaalvorm	Omschrijving	Voltoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	N
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	N
3	Ieder attribuut in de tabel is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel	N

Onderbouwing

De tabel ORDER_HEADER heeft als key de kolom ORDER_NUMBER wat iedere regel uniek maakt.

De kolom RETAILER_NAME zorgt echter voor herhaling wanneer een order is geplaatst door dezelfde RETAILER. Dit is een overtreding van de 1^{ste} normaalvorm. De kolom RETAILER_NAME heeft ook geen afhankelijkheid met de key kolom ORDER_NUMBER van deze tabel, wat een overtreding is van de 2^{de} normaalvorm. Verder zegt kolom RETAILER_NAME iets over de RETAILER en lijkt mij daarom thuis horen in de RETAILER tabel.

Deze tabel staat in de 0^{de} normaalvorm door de herhalende waarden in de RETAILER_NAME kolom, zie onderstaande afbeelding.

De tabel ORDER_HEADER is **niet** correct genormaliseerd volgens de 3^e normaalvorm.

RETAILER_SITE

Tabel: RETAILER_SITE		
Normaalvorm	Omschrijving	Voltoed (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	N
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Ieder niet key attribuut in de tabel is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel	N

Onderbouwing

Bij tabel RETAILER_SITE valt het opnieuw op dat de kolommen ADDRESS1 en ADDRESS2 herhalende kolommen zijn. Het opslaan van een herhalende groep op een regel is een overtreding van de 1^{ste} normaalvorm. Deze tabel staat eigenlijk volgens de normalisatieregels in de 0^{de} (0 NF) normaalvorm. Ook bij deze tabel blijkt uit de data dat ADDRESS2 een toevoeging is van het adres. De naam is ook wat onhandig gekozen, ADDRESS en ADDRESS_ADDITION is misschien een betere benaming.

Zoals ook bij de analyse van tabel SALES_BRANCH benoemd is, is dat er veel overeenkomstige kolommen zijn. Dit is niet direct een overtreding van één van de normalisatie regels maar er zijn dus vijf kolommen met dezelfde naam.

Als we kijken naar de data zien we herhaling in verschillende kolommen. Dit komt omdat er meerdere retailers zijn in dezelfde CITY en REGION, of zelfde CITY,REGION én POSTAL_ZONE. Ook in de kolom ACTIVE_INDICATOR is een herhaling te zien.

Door de herhaling in een aantal kolommen voltoed deze tabel niet aan de 1e normaalvorm.

RETAILER_TYPE

Tabel: RETAILER_TYPE		
Normaalvorm	Omschrijving	Voltoed (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y

3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y
---	--	---

Onderbouwing

Primary key: RETAILER_TYPE_CODE

De relatie one-to-many naar RETAILER. Deze is correct.

Wanneer we RETAILER_TYPE samen zouden voegen met de RETAILER tabel, krijgen we herhaling van de attributen van de TYPE_NAME_EN kolom. Dit is niet gewenst vanuit het eerste normaalvorm perspectief en overtreedt daarmee de eerste normaalvorm. De tabel staat correct in de derde normaalvorm. Dit omdat de TYPE_NAME_EN afhankelijk is van de primary key. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen aanwezig.

RETAILER

Tabel: RETAILER		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

Primary key: RETAILER_CODE

Foreign key: RETAILER_TYPE_CODE

Relatie one-to-many naar SALES_TARGET is correct. Een retailer kan, maar hoeft geen, sales targets te hebben. Een sales target is gekoppeld aan een retailer.

Relatie one-to-many naar RETAILER_SITE is correct. Een retailer kan één of meerder “sites” hebben. En een site is gekoppeld aan een retailer.

Onderbouwing

De RETAILER tabel is op zichzelf staand correct genormaliseerd. Echter is in de SALES_TARGET tabel gekozen om de RETAILER_NAME voluit uit te schrijven. Dit zou overbodig zijn omdat de RETAILER_CODE al informatie verschafft over de COMPANY_NAME vanuit de RETAILER tabel.

Wanneer we de RETAILER tabel zouden combineren met de RETAILER_SITE kan het zijn dat er herhalende retailer namen voor gaan komen omdat een retailer meerdere sites kan hebben. De splitsing is een juiste keuze. Hetzelfde geldt voor de SALES_TARGET tabel. Echter maakt de

SALES_TARGET tabel niet op de juiste wijze hier gebruik van. Ik verwijs dan ook graag naar de analyse van de SALES_TARGET kolom.

De naam RETAILE_CODEMR is een ongelukkig gekozen naam voor een kolom. Het volgt niet de naamgevingsconventie voor kolomnamen. Het is niet duidelijk wat hier mee bedoeld wordt. Dit kan een typo zijn maar ook bewust. Dit zou voor beter omschreven moeten worden. Voor de normalisering is het in dit geval onduidelijk of het de tweede of derde normaalvorm overtreedt. De sleutelafhankelijkheid is nu niet goed te controleren. De aanname is gedaan dat het wel een afhankelijkheid heeft met de key en geen transitieve afhankelijkheid heeft met de keys, en dus daarom ook in de 2-de en 3-de normaalvorm staat.

ORDER_DETAILS

Tabel: ORDER_DETAILS		
Normaalvorm	Omschrijving	Voltoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	N
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	N
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	N

PK - ORDER_DETAIL_CODE

Relatie many-to-one naar ORDER_HEADER de tabel is niet correct. Iedere ORDER_HEADER zou een ORDER_DETAIL moeten hebben. Met de zero or many relatie naar ORDER_DETAIL kan een ORDER_HEADER ook geen ORDER_DETAIL bevatten.

Relatie one-to-many naar RETURNED_ITEM table. Aan de notatie te lezen is het een one to zero or many. En dit is correct. Een ORDER hoeft geen returned item te hebben namelijk.

De tabel ORDER_DETAILS is **niet** correct genormaliseerd.

Onderbouwing

Tabel ORDER_DETAILS zou gekoppeld moeten worden met de PRODUCT tabel.

Hierin staat de primary key PRODUCT_NUMBER waarna ook direct de production_cost -> unit_cost afgeleid kan worden. Echter hebben we wel baat bij UNIT_COST in de tabel gezien het behouden van eventuele histories data. Maar voor de volledigheid, de PRODUCT_MARGIN zou gebruikt kunnen worden om de unit_price te berekenen.

De kolommen UNIT_COST, UNIT_PRICE en UNIT_SALES_PRICE hebben veel herhaling. Dit overtreedt de eerste normaalvorm.

De kolom UNIT_SALE_PRICE zou verwijderd kunnen worden en met een Stored Procedure (SP) berekend kunnen worden. Nu is het handmatig invoeren van een UNIT_SALE_PRICE erg foutgevoelig. Dit kan in het nadeel van de omzet van het bedrijf doorwerken wanneer er te hoge kortingen worden gegeven dan eigenlijk de bedoeling. Voor de normalisering heeft dit verder geen invloed.

RETURNED_ITEM

Tabel: RETURNED_ITEM		
Normaalvorm	Omschrijving	Voltoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutel attributen	Y

PK - RETURN_CODE

FK's - ORDER_DETAIL_CODE en RETURN_REASON_CODE

Relatie many-to-one naar RETURN_REASON. De relatie tot de RETURN_REASON is correct. Een returned item heeft maar één reason. En een reason kan aan meerdere returned items worden toegekend.

Onderbouwing

Wanneer deze twee tabellen zouden worden samengevoegd zal er herhaling plaats gaan vinden van de reden waarom een item is geretourneerd. Om aan de NF1 te voldoen is het zaak te voorkomen dat er geen herhalende groepen aanwezig zijn.

RETURN_REASON

Tabel: RETURN_REASON		
Normaalvorm	Omschrijving	Voltoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y

3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y
---	--	---

Primary key is RETURN_REASON_CODE.

De table RETURN_REASON_CODE is correct genormaliseerd:

Onderbouwing

De tabel RETURN_REASON heeft zelf geen relaties met andere tabellen anders dan de RETURNED_ITEM tabel. De RETURN_REASON tabel is een splitsing van de tabel RETURNED_ITEM. Dit is een logische splitsing. Wanneer we beide tabellen zouden combineren zou er een herhaling van de reden van retournering voor gaan komen. Dit is niet gewenst vanuit het eerste normaalvorm perspectief en overtreedt daarmee de eerste normaalvorm. De tabel staat correct in de derde normaalvorm. Dit omdat de REASON_DESCRIPTION_EN afhankelijk is van de primary key. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen aanwezig.

PRODUCT

Tabel: PRODUCT		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De one to many relatie naar CAMPAIGN is juist omdat product nummer vaker dan een maal gelinkt kunnen zijn aan de campagne.

De one to many relatie naar INVENTORY LEVELS is juist omdat een product in meerdere inventory levels voor kan komen omdat deze per maand worden bijgehouden en producten vaak langer dan 1 maand beschikbaar zijn.

De one to many relatie naar PRODUCT FORECAST is juist omdat een product in meerdere forecasts voor kan komen omdat deze forecasts maandelijks gemaakt lijken te worden.

De tabel PRODUCT is correct genormaliseerd volgens de 1^e, 2^e en 3^e normaalvorm.

Onderbouwing

Er zijn geen herhalende groepen van gegevens in de tabel. Alle niet sleutel attributen zijn niet functioneel afhankelijk van andere niet sleutel attributen maar onafhankelijk.

INVENTORY LEVELS

Tabel: INVENTORY_LEVELS		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De many to one relatie naar PRODUCT FORECAST is juist omdat een product in meerdere product forecasts voor kan komen omdat de forecasts maandelijks gemaakt worden.

De tabel INVENTORY LEVELS is correct genormaliseerd volgens de 1^e, 2^e en 3^e normaalvorm.

Onderbouwing

Elke cel in de tabel bevat een enkele waarde, er is geen spraken van herhalende groepen gegevens. Er is geen gedeeltelijke afhankelijkheid en de kolommen zijn functioneel afhankelijk van de volledige primaire sleutels. Er zijn twee primaire sleutels, die samen een samengestelde primaire sleutel vormen. Dit is in dit geval juist omdat er twee unieke waardes nodig zijn. Zowel de maand als het jaar waarin de inventory gecheckt zijn is belangrijk om een unieke waarde te krijgen.

PRODUCT_FORECAST

Tabel: PRODUCT_FORECAST		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y

	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De one to many relatie naar PRODUCT is juist omdat een product in meerdere forecasts voor kan komen omdat deze forecasts maandelijks gemaakt lijken te worden.

De tabel INVENTORY_FORECAST is correct genormaliseerd volgens de 1^e, 2^e en 3^e normaalvorm.

Onderbouwing

PRODUCT_FORECAST heeft geen gemixte datatypes en er is geen rijvolgorde om informatie over te brengen. Iedere non key attribute in de tabel is afhankelijk van de gehele primary key. Er zijn twee primary keys, die samen een samengestelde primary key vormen. Dit is in dit geval juist omdat er twee unieke waarden nodig zijn. Zowel de maand als het jaar waarin de inventory voorspeld is, is belangrijk om een unieke waarde te krijgen.

CAMPAIGN

Tabel: CAMPAIGN		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De tabel CAMPAIGN is correct genormaliseerd volgens de 1^e, 2^e en 3^e normaalvorm.

Onderbouwing

Als er geen primaire sleutel is gedefinieerd voor de tabel voldoet dit niet aan de regels voor een database. Het hebben van een primaire sleutel is een van de basisregels van relationele databases. Dit omdat een primaire sleutel de integriteit van de gegevens in de database waarborgt. Ook helpt het om relaties met andere tabellen te kunnen definiëren. Hoewel de tabel CAMPAIGN in eerste opzicht geen primary key lijkt te hebben, heeft deze wel een primary key door de twee foreign keys.

Twee foreign keys kunnen samen een samengestelde sleutel vormen uit meerdere velden. Een samengestelde sleutel fungeert ook als een primaire sleutel. Hierdoor zijn beide secundaire sleutels nodig om te bepalen wat de discount van een product is. Zie onderstaande afbeelding:

PROMOTION

Tabel: PROMOTION		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De tabel PROMOTION is correct volgens de 3^e normaalvorm.

One to many relatie naar CAMPAIGN is correct. Een promotie kan meerdere campaigns hebben omdat producten verschillende kortingen kunnen hebben. Dit omdat wij in de dataproduct nummers in meerdere promoties hebben zien terugkomen.

Onderbouwing

PROMOTION heeft een primaire sleutel die ervoor zorgt dat elke promotie een unieke identiteit heeft. Ook zijn de andere kolommen functioneel afhankelijk van de primaire sleutel. Iedere niet sleutel kolom is afhankelijk van enkel de sleutel. Er is simpel in te lezen wat de start en einddata is van de promoties. Bij deze tabel moet er wel van uitgaan worden dat er altijd een begin- en einddatum voor een promotie bekend is, dit is op het moment het geval omdat de data's niet nullabel zijn en ook in de database voor elke promotie een start en eind data genoteerd is.

SALES_STAFF

Tabel: SALES_STAFF		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y

2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De tabel SALES_STAFF is correct genormaliseerd volgens de 1^e, 2^e en 3^e normaalvorm.

Onderbouwing

De tabel heeft als sleutel SALES_STAFF_CODE. Dit is een ID-nummer per medewerker. Alle-nonsleutel attributen van de tabel SALES_STAFF bevatten allerlei soorten losse gegevens over een medewerker van GO. De tabel bevat geen dubbele gegevens, aangezien de naam en contactgegevens vrijwel allemaal verschillen. Verder bevat de tabel geen gegevens die in andere tabellen voorkomen of anderzijds niet in deze tabel horen. Deze tabel is daarom volledig in orde.

SALES_TARGET

Tabel: SALES_TARGET		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	N

De tabel SALES_TARGET is NIET correct genormaliseerd volgens de 3^e normaalvorm.

Onderbouwing

Deze tabel bevat een kolom RETAILER_NAME. Deze kolom lijkt te naam van de retailer te bevatten waardoor de target geldt. Echter bevat de tabel ook al een kolom RETAILER_CODE. Hiermee zou (middels een JOIN met de tabel RETAILER) direct al de RETAILER_NAME gezocht kunnen worden bij alle SALES_TARGET records. Daarom is de kolom RETAILER_NAME overbodig. Dit maakt dat de tabel niet correct genormaliseerd is.

ORDER_METHOD

Tabel: ORDER_METHOD		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y

	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De tabel ORDER_METHOD is correct genormaliseerd volgens de 3^e normaalvorm.

Onderbouwing

De tabel ORDER_METHOD is een erg simpele tabel die slechts uit een code en een naam bestaat. Er is een foreign key vanaf de tabel ORDER_HEADER. Het is volgende de normalisatieregels juist dat dit een aparte tabel is, doordat meerdere orders uit de ORDER_HEADER tabel dezelfde ORDER_METHOD delen. Aan deze tabel hoeft dan ook niets gewijzigd te worden.

PRODUCT_LINE

Tabel: PRODUCT_LINE		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y
	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De tabel PRODUCT_LINE is correct genormaliseerd volgens de 1^e, 2^e en 3^e normaalvorm.

Onderbouwing

De tabel PRODUCT_LINE is een erg simpele tabel die slechts uit een code en een naam bestaat. Er is een foreign key vanaf de tabel PRODUCT_TYPE. Het is volgende de normalisatieregels juist dat dit een aparte tabel is, doordat meerdere product types uit de PRODUCT_TYPE tabel dezelfde PRODUCT_LINE delen. Aan deze tabel hoeft dan ook niets gewijzigd te worden.

PRODUCT_TYPE

Tabel: PRODUCT_TYPE		
Normaalvorm	Omschrijving	Voldoet (Y/N)
1	De tabel heeft een primary key	Y

	Er zijn geen gemixte data types aanwezig in eenzelfde kolom	Y
	Er wordt geen rijvolgorde gehanteerd om informatie over te brengen	Y
	Er zijn geen herhalende groepen aanwezig	Y
2	Iedere non-key attribuut in de tabel is afhankelijk van de gehele primary key	Y
3	Elke niet-sleutel attribuut is afhankelijk van de sleutel, de hele sleutel en niets anders dan de sleutel. Ook zijn er geen transitieve afhankelijkheden tussen niet-sleutelattributen	Y

De tabel PRODUCT_TYPE is correct genormaliseerd volgens de 3^e normaalvorm.

Onderbouwing

De tabel PRODUCT_TYPE is een simpele tabel die slechts uit een code, een naam en een foreign key attribuut naar PRODUCT_LINE bestaat. Er is tevens een foreign key vanaf de tabel PRODUCT.

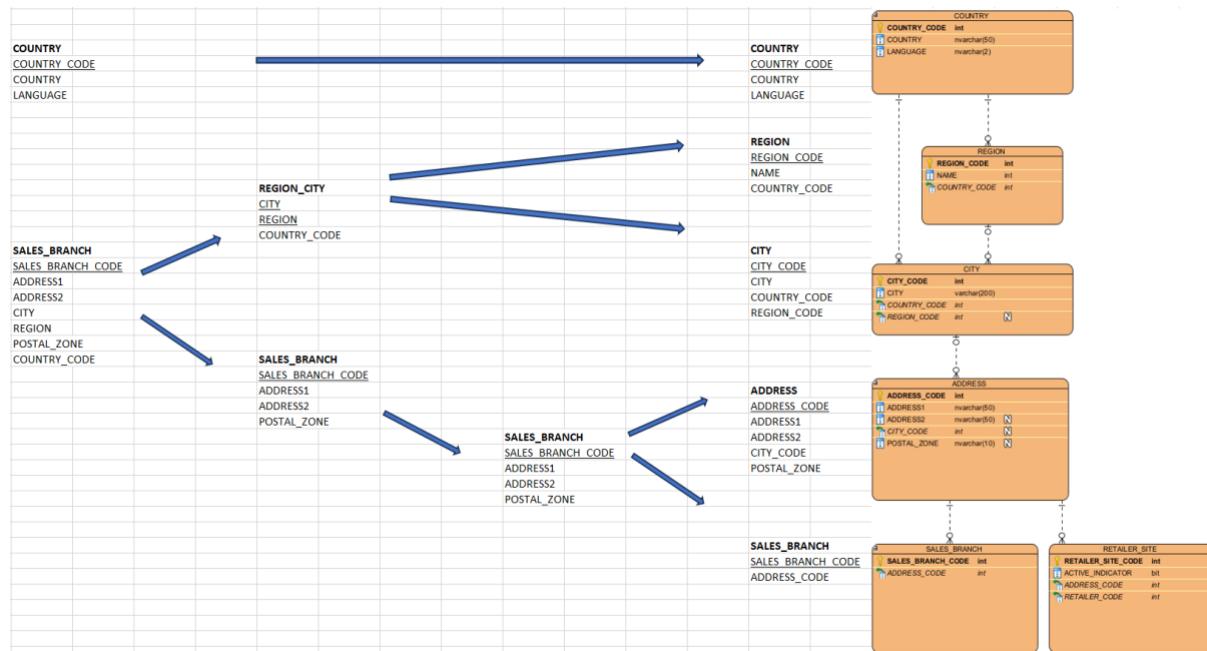
Het is volgende de normalisatieregels juist dat dit een aparte tabel is, doordat meerdere producten uit de PRODUCT tabel dezelfde PRODUCT_TYPE delen. Aan deze tabel hoeft dan ook niets gewijzigd te worden.

Nieuw ERD ontwerp

Het model van Great Outdoor hebben wij geanalyseerd. Enkele wijzigingen aan de hand van de normalisatiestappen hebben wij hieronder uitgewerkt. Ook enkele verbeteringen ten aanzien van het verminderen van redundantie leggen we hier uit.

SALES_BRANCH

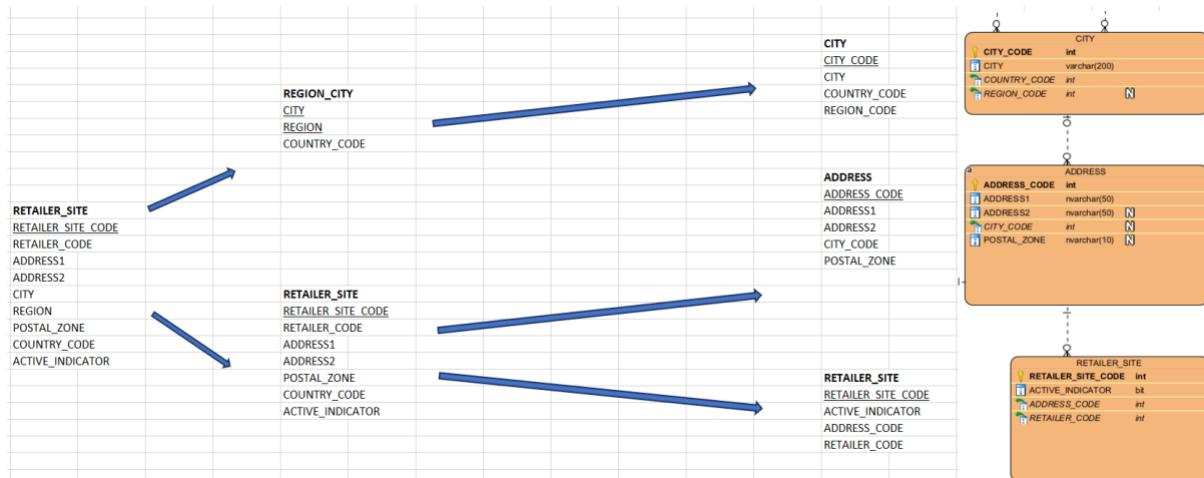
Uit de analyse van de tabellen SALES_BRANCH en RETAILER_SITE komt naar voren dat de kolommen ADDRESS1 en ADDRESS2 géén herhalende kolommen zijn. Wat we wel zien is dat er vijf kolommen hetzelfde zijn. Door dezelfde CITY, REGION en POSTAL zone op te voeren krijgen herhaling in een deel van de rijen. We lossen dit op door de ADDRESS, CITY en REGION attributen te verhuizen en een aparte entiteit te geven.



Afb. Normalisatie stappen naar 3^e normaalvorm.

RETAILER_SITE

Tabel RETAILER_SITE bevat herhalende attribuut waarden zoals we zagen in de analyse. We maken hiervoor een aparte entiteiten waarbij de adressen in een aparte tabel gaan net als de adressen in SALES_BRANCH. Steden (CITY) en regio, (REGION) krijgen ook een eigen entiteit. Zo is REGION, CITY en ADDRESS te gebruiken door RETAILER_SITE, zie onderstaande afbeelding.



Afb. Normalisatie stappen naar 3^e normaalvorm voor RETAILER_SITE.

COUNTRY

De tabel COUNTRY blijft ongewijzigd

ORDER_HEADER

De tabel ORDER_HEADER heeft zoals we zagen in de analyse herhalende attribuut waarden voor de kolom RETAILER_NAME. Door deze kolom te verwijderen kunnen we dit oplossen. De kolom verhuist naar de tabel RETAILER. De tabel heeft echter al een RETAILER kolom in de vorm van COMPANY_NAME. Hiermee vervalt de RETAILER_NAME kolom.

ORDER_HEADER		ORDER_HEADER		ORDER_HEADER
ORDER NUMBER		ORDER NUMBER		ORDER NUMBER
RETAILER_NAME		RETAILER_NAME		RETAILER_SITE_CODE
RETAILER_SITE_CODE	→	RETAILER_SITE_CODE	→	RETAILER_CONTACT_CODE
RETAILER_CONTACT_CODE		RETAILER_CONTACT_CODE		SALES_STAFF_CODE
SALES_STAFF_CODE		SALES_STAFF_CODE		SALES_BRANCH_CODE
SALES_BRANCH_CODE		SALES_BRANCH_CODE		ORDER_DATE
ORDER_DATE		ORDER_DATE		ORDER_METHOD_CODE
ORDER_METHOD_CODE		ORDER_METHOD_CODE		

Afb. Normalisatie stappen naar 3^e normaalvorm voor ORDER_HEADER.

ORDER_DETAILS

De tabel ORDER_DETAILS is niet op de juiste wijze genormaliseerd. De kolommen UNIT_COST, UNIT_PRICE en UNIT_SALES_PRICE hebben veel herhaling. Dit overtreedt de eerste normaalvorm. Aangezien de tabel nu in de nulde normaalvorm staat, hebben we de ORDER_DETAIL attributen nagelopen. Hier kwamen wij tot de conclusie dat deze veel gelijkenis heeft met de PRODUCT tabel.

De PRODUCT tabel heeft de attributen PRODUCTION_COST en MARGIN. Dit geeft al informatie over de kostprijs van een product. Aan de hand van de MARGIN kan afgeleid worden wat de verkoopprijs zal zijn.

De ORDER_DETAILS tabel heeft de attributen UNIT_COST, UNIT_PRICE en UNIT_SALE_PRICE. Deze attributen kunnen vervallen wanneer we de ORDER_DETAILS tabel koppelen aan PRODUCT. UNIT_COST is equivalent aan PRODUCTION_COST. De UNIT_PRICE is te berekenen door de PRODUCTION_COST te vermenigvuldigen met de MARGIN en het resultaat weer op te tellen bij de PRODUCTION_COST. De UNIT_SALE_PRICE is weer af te leiden door een korting die eventueel gegeven kan worden. Omdat we de ORDER_DETAILS tabel koppelen aan de PRODUCT tabel, kan via de PRODUCT tabel weer een DISCOUNT afgeleid worden vanuit de CAMPAIGN-tabel. Deze heeft op

zijn beurt weer een relatie met de PRODUCT tabel. Omdat we ook graag rekening willen blijven houden met behoud van historische data, hebben wij de keuze gemaakt om UNIT_PRICE in de ORDER_DETAILS tabel te laten bestaan. Het is weliswaar mogelijk om deze ook vanuit de PRODUCT tabel af te lijden, alleen verlies je er dan wel de vastlegging van de ooit bepaalde verkoop prijs van een order.

RETAILER

In de tabel RETAILER komt het attribuut RETAILER_CODEMR voor. Dit is een nikszeggende naamgeving voor de kolom. Ook aan de waardes is niet af te leiden wat het in zou kunnen houden. Wat dat betreft zou het normalisatieproces stagneren omdat eerst uitgezocht moet worden wat deze kolom nou precies inhoudt. Ook komen er in deze kolom geregeld NULL-waardes terug. Dit hoeft voor normalisatie verder geen probleem te zijn, maar vergt wel de aandacht. Het is dus eerst van belang te begrijpen wat de kolom nou precies inhoudt, zodat ook de NULL-waardes geëvalueerd kunnen worden of deze legitiem zijn.

SALES_TARGET

In de tabel SALES_TARGET zou het attribuut RETAILER_NAME kunnen vervallen. Door de koppeling met de RETAILER tabel heeft de SALES_TARGET ook een relatie met de COMPANY_NAME attribuut uit de RETAILER tabel.

Overige opmerkingen en adviezen

<Kolomnaam>_EN, waar de _EN staat voor de taal, hebben wij gekozen deze weg te laten. Om de database te optimaliseren zouden wij aanraden om voor een vaste taal te gaan. Mochten er informatie in een aparte taal nodig zijn dan kan hier afhankelijk van de informatie een aparte kolom of tabel voor aangemaakt worden. Wij raden aan dat als bedrijfsregel alle waardes opgeslagen worden in 1 taal. In dit geval dat de database Engelstalig wordt ingevoerd.

Wij hebben ook een aantal data types aangepast in het ERD.

Datetime:

Voor DATE_HIRED hebben wij date in plaats van datetime gebruikt. In ons opzicht is er geen meerwaarde voor het noteren van exacte tijd dat een nieuwe werknemer wordt aangenomen. De dag zal hierin volstaan.

PROMOTION en ORDERDATE hebben het datatype datetime. Wij hebben ervoor gekozen om deze te vervangen voor DateTimeOffset. Hiervoor is gekozen omdat wij denken dat de timezone relevant is voor deze data. Dit om een juiste retour periode vanuit orderdate vast te kunnen stellen en om de promotions op logische tijden te laten beginnen en eindigen in de verschillende timezones.

Decimal:

In dbo.DISCOUNT is de DISCOUNT te vinden met het datatype real. Wij denken dat decimal hier gepaster is. Hiermee kunnen exacte waardes worden opgegeven, hierbij kunnen de decimalen van bedragen bijvoorbeeld beter worden opgeslagen. Real is hier minder precies in door de aard van floating point getallen.

Float:

Float(53) is meerdere malen terug te vinden in het ERD, wij hebben besloten om deze allemaal te vervangen voor money omdat er alleen geldbedragen worden opgeslagen in deze kolommen. Geldbedragen hebben meestal exact 2 decimalen en float is hier minder voor geschikt. Ook dit door

de aard van floating point getallen. De reden dat wij niet voor small money hebben gekozen is omdat dit mogelijk niet genoeg ruimte geeft voor grote bedragen.

Nieuwe ERD

Het nieuwe ERD is te vinden als Bijlage 1.

Queries

In dit hoofdstuk laten we enkele voorbeelden zien van query's die geschreven kunnen worden op het oorspronkelijke databaseschema van GO. Binnen deze query's demo'en we enkele geavanceerde SQL-mogelijkheden die handig kunnen zijn binnen query's, zoals GROUP BY, CASE en CTE's.

Complexe Query 1 – Categorieën per retourreden

Great Outdoor heeft kwaliteit hoog in het vaandel staan en mikt op een doelgroep die hiervoor bereid is te betalen. Omdat het management te faciliteren in het bepalen welke producten voor het meeste problemen zorgen, zijn we aan de slag gegaan om te bepalen welke producten het vaakst worden teruggebracht wordt.

De lijst moet alle producten bevatten die zijn geretourneerd met het aantal per product en reden. Ook het totaal aantal per product moet worden getoond. De lijst moet gesorteerd worden op totaal aantal geretourneerd van hoog naar laag.

Verder willen we het management ondersteunen in het bepalen van de belangrijkste oorzaken van retouren door alle retourredenen te categoriseren in slechts drie categorieën:

- Product gerelateerd
 - Dit zijn de oorzaken waarvoor de leverancier of fabrikant van de producten mogelijk aangewezen kan worden als verantwoordelijke.
- Great Outdoor oorzaak
 - Dit zijn de oorzaken waarvoor Great Outdoor kan worden aangewezen als verantwoordelijke.
- Klant gerelateerd
 - Dit zijn de oorzaken waarvoor de klant kan worden aangewezen als verantwoordelijke.

De verdeling in categorie moet als volgt gebeuren:

Retourreden	Gecategoriseerd als
Defective product	Product issue
Incomplete product	Product issue
Unsatisfactory product	Product issue
Wrong product shipped	Great Outdoor related issue
Damaged during shipping	Great Outdoor related issue
Wrong product ordered	Customer related issue

In ons voorbeeld nemen we de data uit 2005 als voorbeeld. Dit jaartal hebben we gekozen omdat er in dat jaar veel data beschikbaar is. Het jaartal in de query kan eenvoudig aangepast worden op regel 11. Zo kan de data per jaar vergeleken worden voor BI doeleinden.

Query en resultaat

```
WITH RETURNED_ITEMS_BY_REASON_CTE AS (
    SELECT
        p.PRODUCT_NAME,
        SUM(r.RETURN_QUANTITY) NUMBER_OF RETURNS_BY_REASON,
        rr.REASON_DESCRIPTION_EN AS REASON_DESCRIPTION_EN
    FROM
        ORDER_DETAILS o
        JOIN RETURNED_ITEM r ON o.ORDER_DETAIL_CODE = r.ORDER_DETAIL_CODE
        JOIN PRODUCT p ON p.PRODUCT_NUMBER = o.PRODUCT_NUMBER
        JOIN RETURN_REASON rr ON rr.RETURN_REASON_CODE = r.RETURN_REASON_CODE
    WHERE YEAR(r.RETURN_DATE) = 2005
    GROUP BY
        o.PRODUCT_NUMBER,
        p.PRODUCT_NAME,
        rr.REASON_DESCRIPTION_EN
)
SELECT
    ri.PRODUCT_NAME,
    SUM(ri.NUMBER_OF RETURNS_BY_REASON) OVER (PARTITION BY ri.PRODUCT_NAME)
TOTAL_RETURN_AMOUNT,
    ri.NUMBER_OF RETURNS_BY_REASON,
    ri.REASON_DESCRIPTION_EN,
    CASE
        WHEN ri.REASON_DESCRIPTION_EN IN (
            'Defective product',
            'Incomplete product',
            'Unsatisfactory product'
        ) THEN 'Product issue'
        WHEN ri.REASON_DESCRIPTION_EN IN (
            'Wrong product shipped',
            'damaged during shipping'
        ) THEN 'Great Outdoor related issue'
        WHEN ri.REASON_DESCRIPTION_EN IN (
            'Wrong product ordered'
        ) THEN 'Customer related issue'
    END AS CATEGORY
FROM
    RETURNED_ITEMS_BY_REASON_CTE ri
ORDER BY
    TOTAL_RETURN_AMOUNT DESC,
    ri.NUMBER_OF RETURNS_BY_REASON DESC;
```

Dit geeft het volgende resultaat (waarvan de eerste 20 rijen):

	PRODUCT_NAME	TOTAL_RETURN_AMOUNT	NUMBER_OF RETURNS_BY_REASON	REASON_DESCRIPTION_EN	CATEGORY
1	TrailChef Cup	204	164	Unsatisfactory product	Product issue
2	TrailChef Cup	204	32	Incomplete product	Product issue
3	TrailChef Cup	204	8	Defective product	Product issue
4	EverGlow Lamp	180	172	Wrong product ordered	Customer related issue
5	EverGlow Lamp	180	8	Defective product	Product issue
6	Firefly 4	172	112	Unsatisfactory product	Product issue
7	Firefly 4	172	60	Wrong product ordered	Customer related issue
8	TrailChef Water Bag	168	154	Unsatisfactory product	Product issue
9	TrailChef Water Bag	168	14	Defective product	Product issue
10	Hibernator Self - Inflating Mat	160	160	Unsatisfactory product	Product issue
11	Firefly Lite	152	128	Wrong product shipped	Great Outdoor related issue
12	Firefly Lite	152	24	Unsatisfactory product	Product issue
13	Glacier Basic	144	92	Wrong product ordered	Customer related issue
14	Glacier Basic	144	40	Unsatisfactory product	Product issue
15	Glacier Basic	144	10	Incomplete product	Product issue
16	Glacier Basic	144	2	Defective product	Product issue
17	Firefly Rechargeable Battery	126	66	Wrong product shipped	Great Outdoor related issue
18	Firefly Rechargeable Battery	126	52	Unsatisfactory product	Product issue
19	Firefly Rechargeable Battery	126	8	Incomplete product	Product issue
20	TrailChef Kitchen Kit	120	112	Wrong product shipped	Great Outdoor related issue

Afb. Geretourneerde producten gesorteerd op totaal aantal geretourneerd van hoog naar laag.

De sortering die is toegepast is nu op kolomnaam op aflopende volgorde van aantal stuks geretourneerd per product, en daarbinnen aflopend gesorteerd op aantal geretourneerd per reden.

De query bevat een analytische functie (Microsoft, 2023) om het totaal per product te berekenen. Door na de OVER clause achter de PARTITION aan te geven over welke deel de som moet worden berekend, namelijk per PRODUCT_NAME krijgen we per product het totaal.

De som in de CTE zorgt voor de som per product per retour reden en wordt daarna aangeroepen en getoond naast het total per product.

Dit voegt toe dat er op twee manieren naar de geretourneerde producten gekeken wordt. Dus ook het aantal van de retouren per product zoals het manager heeft gevraagd.

Conclusie

We zien dat product *TrailChef Cup* het vaakst wordt geretourneerd met 164 stuks met als reden "Unsatisfactory product". Bovenaan zien we tevens drie oorzaken uit de categorie "Product issue". Gezien deze uitkomst, zou Great Outdoor contact kunnen opnemen met de leverancier of fabrikant van dit artikel om hen van de problemen op de hoogte te brengen.

Door de grootste oorzaken van retourneren te analyseren en aan te pakken kan Great Outdoor de kwaliteit van haar dienstverlening verbeteren en besparen op personeelskosten veroorzaakt door deze retouren.

Complexe Query 2 – Hoeveelheid werk per ordermethode

Voor een bedrijf als Great Outdoor kan het nuttig zijn inzicht te krijgen in welke order methodes er het minste orders opleveren. Iedere order methode kost het bedrijf geld. Het kan lonen om dit onder de loep te nemen, en misschien wel methodes schrappen. Ook kunnen deze gegevens het bedrijf een indruk geven welke order methode het meest opleveren. Hier kunnen zij intern op sturen door hier meer focus en inspanning aan te geven.

Hierbij hebben we de volgende aannames gemaakt.

Categorie	Omschrijving	Order methode(s)
1	Geen extra werk	5

2	Weinig extra werk	1, 3, 4
3	Veel extra werk	2, 7, 8

Geen extra werk houdt in dat het bedrijf er geen werk aan heeft. Een voorbeeld hiervan is dat een klant zelf de order in de webbrowser invoert waarna de order direct in de database wordt opgeslagen. Er is geen personeel nodig geweest om de order in de database toe te voegen.

Weinig extra werk houdt in dat een medewerker van het bedrijf orders binnenkrijgt via een online medium. Deze wordt gelezen en verwerkt door een medewerker en opgeslagen in de database.

Omdat hier toch handwerk bij is (maar het werk beperkt is en bijvoorbeeld met copy-paste versneld uitgevoerd kan worden) beschouwen we dit als “weinig” werk.

Veel extra werk houdt in dat het bedrijf een medewerker de klant moet laten nabellen, of zelf gebeld wordt om de order(s) telefonisch af te handelen. Ook het fysiek sturen van een medewerker naar een klant om zaken met de klant te bespreken en orders op te nemen, betekent veel extra werk.

Wij hebben de aannname gedaan om een invulling te geven voor ORDER_METHOD_CODE 8, “Special”, dat de klant een vaste accountmanager heeft die ondersteuning biedt en regelmatig contact heeft/houdt met de klant(en).

De volgende order methodes zijn nu bekend binnen het bedrijf.

<input type="checkbox"/> ORDER_METHOD_CODE	<input type="checkbox"/> ORDER_METHOD_EN
1	Fax
2	Telephone
3	Mail
4	E-mail
5	Web
7	Sales visit
8	Special

Figuur 1 - Order methodes (ORDER_METHODS tabel)

Query en resultaat

```

WITH ORDER_VALUE_CTE (ORDER_NUMBER, SUBTOTAL)
AS(SELECT o_details.ORDER_NUMBER,
      SUM(o_details.QUANTITY * o_details.UNIT_PRICE)
   FROM ORDER_HEADER o_header
    INNER JOIN ORDER_DETAILS o_details
      ON o_details.ORDER_NUMBER = o_header.ORDER_NUMBER
  GROUP BY o_details.ORDER_NUMBER),
AMOUNT_OF_WORK_CTE (ORDER_METHOD_CODE, AMOUNT_OF_WORK)
AS (SELECT method.ORDER_METHOD_CODE,
           CASE
             WHEN method.ORDER_METHOD_CODE = 1 THEN 'Least manual work'
             WHEN method.ORDER_METHOD_CODE = 2 THEN 'Most manual work'
             WHEN method.ORDER_METHOD_CODE = 3 THEN 'Average manual work'
             WHEN method.ORDER_METHOD_CODE = 4 THEN 'Average manual work'
             WHEN method.ORDER_METHOD_CODE = 5 THEN 'Least manual work'
             WHEN method.ORDER_METHOD_CODE = 7 THEN 'Most manual work'
             WHEN method.ORDER_METHOD_CODE = 8 THEN 'Most manual work'
             ELSE 'UNKNOWN'
           END AS AMOUNT_OF_WORK
      FROM ORDER_METHOD method)
SELECT work.AMOUNT_OF_WORK,
       Count(*)                               ORDER_COUNT,
       Cast(Sum(order_v.SUBTOTAL) AS DECIMAL(20, 2)) AS TOTAL_REVENUE
  FROM ORDER_METHOD method
   INNER JOIN ORDER_HEADER order_h
     ON order_h.ORDER_METHOD_CODE = method.ORDER_METHOD_CODE
   INNER JOIN ORDER_VALUE_CTE order_v
     ON order_v.ORDER_NUMBER = order_h.ORDER_NUMBER
   INNER JOIN AMOUNT_OF_WORK_CTE work
     ON work.ORDER_METHOD_CODE = order_h.ORDER_METHOD_CODE
 GROUP BY work.AMOUNT_OF_WORK
 ORDER BY ORDER_COUNT DESC

```

<input type="checkbox"/> AMOUNT_OF_WORK	<input type="checkbox"/> ORDER_COUNT	<input type="checkbox"/> TOTAL_REVENUE
Most manual work	2225	69317683.88
Least manual work	1199	38317462.88
Average manual work	937	28985566.70

Figuur 2 - Resultaat opbrengst per order methode category

Conclusie

We zien uit het resultaat dat het overgrote deel van de orders binnenkomt via media die veel extra werk kosten. Hierdoor heeft Great Outdoor mogelijk meer personeelskosten dan nodig. Dit doordat de methodes uit deze categorie veel handwerk vergen zoals handmatig productnummers overnemen, controleren en eventuele fouten corrigeren.

Great Outdoor zou op basis van dit resultaat kunnen overwegen meer in te zoomen op de "Most manual work" categorie. Wellicht kan Great Outdoor bijvoorbeeld actief proberen klanten op de hoogte te brengen van de webshop en/of feedback vragen over de webshop om deze interessanter te maken om te gebruiken voor klanten. Dit zal uiteindelijk de personeelskosten en kan het aantal fouten door het extra handwerk reduceren.

System Catalog

De System Catalog bestaat uit systeemtabellen en views die de structuur van de database beschrijven (Griffin, 2013). Met de structuur worden de objecten bedoeld zoals databases en de zaken die samen de database vormen zoals tabellen, views en indexes. Toegang tot de System Catalog gaat via de database engine.

Een andere beschrijving van de System Catalog is dat het een groep systeemtabellen en views zijn die essentiële details over een database bevatten (RouseTechnologie, 2014). Als voorbeeld is de data dictionary language (DDL) te benoemen voor elke tabel in de database die wordt opgeslagen in de System Catalog, zoals de CREATE TABLE-statements om de betreffende tabellen aan te maken.

Tabellen die in de System Catalog staan zijn de zogenaamde systeemtabellen en leven in de master-database. Doordat de tabellen dezelfde logische structuur als reguliere tabellen hebben, kan men met T-SQL instructies de systeemtabellen raadplegen.

De mogelijkheden die de database engine als interface kan bieden om toegang te krijgen tot de System Catalog zijn (Petkovic, 2020):

- Catalog views: Voor het opvragen en bekijken van metadata die opgeslagen ligt over kenmerken van objecten die het databasesysteem beschrijven.
- Dynamic management views en functies: Voor het observeren van actieve processen en de inhoud van het gebeuren.
- Information schema: Een gestandaardiseerde alleen lezen oplossing voor toegang tot de metadata.
- System and property functions: Ophalen van systeminformatie en eigenschappen. Eigenschapsfuncties kunnen over het algemeen meer informatie opleveren dan systeemfuncties.
- System stored procedures: Sommige systeem stored procedures kunnen gebruikt worden om toegang te verkrijgen en de inhoud aan te passen van de systeembasistabellen. Het wordt ten zeerste aangeraden om alleen system store procedures te gebruiken voor het wijzigen van systeminformatie.

Met de opgesomde interfaces kunnen onder andere de volgende taken worden uitgevoerd:

- Snel raadplegen van informatie over de database.
- Door inzicht in de structuur gemakkelijker wijzigingen aanbrengen en onderhoud plegen aan de databasestructuur.
- Het system catalog geeft ook informatie over de gebruikers, rollen en beveiligingsinstellingen. Hierdoor wordt men in staat gesteld het beleid en toegangsrechten te beheren.
- Het kan helpen bij het onderhouden en verbeteren van de prestaties van een database.

Voorbeeld 1

Omdat we bezig zijn met de analyse van ons model van Great Outdoor zijn we soms opzoek naar een bepaalde kolom van de aangemaakte tabellen in de database.

In dit voorbeeld gebruiken we de catalog views om te zien bij welke tabel een kolom hoort, maar ook is te zien hoe vaak een bepaalde kolomnaam eigenlijk voorkomt in het model van Great Outdoor. Ook potentieel repeterende kolommen zijn hierdoor snel zichtbaar.

Bij een hoog aantal is dit input voor een verdere analyse, bijvoorbeeld het uitzoeken of de tabellen verder genormaliseerd zouden kunnen worden.

In onderstaand voorbeeld zijn de drie catalog views gecombineerd. De PARTITION BY (Microsoft, Microsoft SQL Server, 2023) clause verdeelt een set rijen voordat de window functie wordt toegepast. Dus de OVER clausule definieert een set rijen binnen het resultaat van de query.

Na het uitvoeren van deze query verschijnt het volgende resultaat.

```
SELECT s.name AS schema_name
      , t.name AS table_name
      , c.name AS column_name
      , COUNT(c.name) OVER (PARTITION BY c.name) AS aantal
  FROM sys.schemas s
  JOIN sys.tables t ON s.schema_id = t.schema_id
  JOIN sys.columns c ON c.object_id = t.object_id
 WHERE s.name = 'dbo'
 ORDER BY 4 DESC,2,3
```

	schema_name	table_name	column_name	aantal
1	dbo	CAMPAIGN	PRODUCT_NUMBER	6
2	dbo	INVENTORY_LEVELS	PRODUCT_NUMBER	6
3	dbo	ORDER_DETAILS	PRODUCT_NUMBER	6
4	dbo	PRODUCT	PRODUCT_NUMBER	6
5	dbo	PRODUCT_FORECAST	PRODUCT_NUMBER	6
6	dbo	SALES_TARGET	PRODUCT_NUMBER	6
7	dbo	COUNTRY	COUNTRY_CODE	3
8	dbo	ORDER_HEADER	SALES_BRANCH_CODE	3
9	dbo	ORDER_HEADER	SALES_STAFF_CODE	3
10	dbo	RETAILER	RETAILER_CODE	3
11	dbo	RETAILER_SITE	COUNTRY_CODE	3
12	dbo	RETAILER_SITE	RETAILER_CODE	3
13	dbo	SALES_BRANCH	COUNTRY_CODE	3
14	dbo	SALES_BRANCH	SALES_BRANCH_CODE	3
15	dbo	SALES_STAFF	SALES_BRANCH_CODE	3
16	dbo	SALES_STAFF	SALES_STAFF_CODE	3
17	dbo	SALES_TARGET	RETAILER_CODE	3
18	dbo	SALES_TARGET	SALES_STAFF_CODE	3
19	dbo	CAMPAIGN	PR_NUMBER	2
20	dbo	ORDER_DETAILS	ORDER_DETAIL_CODE	2
21	dbo	ORDER_DETAILS	ORDER_NUMBER	2
22	dbo	ORDER_HEADER	ORDER_METHOD_CODE	2
23	dbo	ORDER_HEADER	ORDER_NUMBER	2
24	dbo	ORDER_HEADER	RETAILER_NAME	2
25	dbo	ORDER_HEADER	RETAILER_SITE_CODE	2
26	dbo	ORDER_METHOD	ORDER_METHOD_CODE	2
27	dbo	PRODUCT	PRODUCT_TYPE_CODE	2
28	dbo	PRODUCT_LINE	PRODUCT_LINE_CODE	2
29	dbo	PRODUCT_TYPE	PRODUCT_LINE_CODE	2
30	dbo	PRODUCT_TYPE	PRODUCT_TYPE_CODE	2
31	dbo	PROMOTION	PR_NUMBER	2

Afb. Aantal voorkomens van een kolom.

Voorbeeld 2

Als tweede voorbeeld kunnen we de volgende query op de System Catalog uitvoeren.

Om te weten welke kolommen er zijn in een bepaalde tabel, bijvoorbeeld de tabel PRODUCT van de GO-database en welke data types ze hebben, kun je de volgende query uitvoeren. Binnen deze query is in de laatste regel de aanname gemaakt dat de database de naam “Greatoutdoor” draagt.

De onderstaande query berekent het aantal rijen en de grootte van deze rijen in kilobytes en megabytes. Onze database komt niet snel aan de megabytes maar dit, en eventueel ook gigabytes kan interessant zijn voor grotere databases. In dit voorbeeld vragen wij deze informatie uit voor PRODUCT en CAMPAIGN.

```
SELECT c.name AS column_name, t.name AS data_type, c.max_length, c.is_nullable
FROM sys.columns c
JOIN sys.types t ON c.user_type_id = t.user_type_id
WHERE c.object_id = object_id('Greatoutdoor.PRODUCT')
```

column_name	data_type	max_length	is_nullable
PRODUCT_NUMBER	int	4	false
INTRODUCTION_DATE	datetime	8	false
PRODUCT_TYPE_CODE	int	4	false
PRODUCTION_COST	float	8	false
MARGIN	float	8	false
PRODUCT_IMAGE	nvarchar	300	false
PRODUCT_NAME	nvarchar	510	true

Figuur 3 - PRODUCT tabel, kolommen en data types

De query gebruikt de system catalog views sys.column en sys.type die informatie geven over alle kolommen in de PRODUCT tabel van de GO-database.

Bron

<https://github.com/MicrosoftDocs/sql-docs/blob/live/docs/relational-databases/system-catalog-views/sys-service-queues-transact-sql.md>

Voorbeeld 3

In het derde voorbeeld voeren wij een query op de System Catalog uit die inzicht geeft over de grootte van specifieke tabellen in de database. Dit kan gebruikt worden voor opslag optimalisatie en analyse. Ook kan het handig zijn als er gewerkt wordt met cloud databases, hier betaal je voor opslag. Het weten van de grootte van tabellen kan helpen met het inschatten van het benodigde budget en het kiezen van het juiste abonnement.

```
SELECT
    t.name AS TableName,
    SUM(p.rows) AS AantalRijen,
    SUM(a.total_pages) * 8 AS DatalengtheKB,
    (SUM(a.total_pages) * 8) / 1024 AS DatalengtheMB
FROM sys.tables t
INNER JOIN sys.partitions p ON t.object_id = p.object_id
INNER JOIN sys.allocation_units a ON p.partition_id = a.container_id
WHERE t.name IN ('PRODUCT', 'CAMPAIGN')
GROUP BY t.name
```

TableName	AantalRijen	DatalengtheKB	DatalengtheMB
CAMPAIGN	7404	264	0
PRODUCT	345	72	0

We gebruiken verschillende system views uit de System Catalog voor deze query. De sys.tables wordt aangeroepen omdat dit het deel van de System Catalog is die informatie bevat over de tabellen in de database. Sys.partitions wordt aangeroepen omdat dit het deel is van de System Catalog die informatie bevat over de segmenten in de tabellen. Sys.allocation_units wordt aangeroepen omdat dit het deel is van de System Catalog dat informatie bevat over de units in de tabellen. (Microsoft, 2023)

Wil je alle tabellen van de database zien uit de query dan kan de query ook uitgebreid worden met een join op sys.schemas zoals in het onderstaande voorbeeld

```

SELECT
    t.name AS TableName,
    SUM(p.rows) AS AantalRijen,
    SUM(a.total_pages) * 8 AS DatalengthKB,
    (SUM(a.total_pages) * 8) / 1024 AS DatalengthMB
    ,s.name AS SCHEMA_NAME
FROM sys.tables t
INNER JOIN sys.partitions p ON t.object_id = p.object_id
INNER JOIN sys.allocation_units a ON p.partition_id = a.container_id
INNER JOIN sys.schemas s ON s.schema_id = t.schema_id
WHERE s.name = ('dbo')
--AND t.name IN ('PRODUCT', 'CAMPAIGN')
GROUP BY t.name ,s.name
ORDER BY 2 DESC,1

```

	TableName	AantalRijen	DatalengthKB	DatalengthMB	SCHEMA_NAME
1	ORDER_DETAILS	249386	12744	12	dbo
2	ORDER_HEADER	92181	2568	2	dbo
3	SALES_TARGET	79060	2248	2	dbo
4	PRODUCT_FORECAST	24572	648	0	dbo
5	RETURNED_ITEM	10466	328	0	dbo
6	CAMPAIGN	7404	264	0	dbo
7	INVENTORY_LEVELS	3888	136	0	dbo
8	RETAILER_SITE	1173	72	0	dbo
9	PRODUCT	345	72	0	dbo
10	SALES_STAFF	306	72	0	dbo
11	RETAILER	218	72	0	dbo
12	PROMOTION	210	72	0	dbo
13	SALES_BRANCH	84	72	0	dbo
14	COUNTRY	63	72	0	dbo
15	PRODUCT_TYPE	42	72	0	dbo
16	RETAILER_TYPE	16	72	0	dbo
17	ORDER_METHOD	14	72	0	dbo
18	PRODUCT_LINE	12	72	0	dbo
19	RETURN_REASON	12	72	0	dbo

Stored Procedures & Functions

Thierry

Stored Procedures

Voor een onderzoek naar Stored Procedures (SP) heb ik informatie hierover ingewonnen op het internet. Er is veel te vinden op het internet dat uitleg geeft over wat SP's zijn. Een van de bronnen die ik voor het meest basale begrip hierover het beste beeld schetste was die van TechTarget.com (Hughes, 2019).

Ook is er in de les op school uitleg gegeven hierover. Het sterk vereenvoudigde tabel met de functionaliteiten en onderlinge verschillen met functies was heel verhelderend (Hogeschool, 2023).

SP heeft veel gelijkenis met functies. Alleen retourneert een SP geen waarde, wat een functie wel kan. Een SP zijn een reeks instructies die na elkaar worden uitgevoerd. Ze zijn ook opgeslagen in de database. SQL-statements die als taak worden geselecteerd om uit een tabel snel gegevens te kunnen halen.

De voordelen van een SP zijn:

- De snelle verwerking van data door het programma draait in de database zelf. Indien het rechtstreeks vanaf de SQL-server wordt uitgevoerd heeft het ook geen last van netwerk overhead. En wanneer er wel een client – server communicatie plaatsvindt heeft een SP het voordeel dat SP in een batch wordt uitgevoerd en over de lijn wordt verstuurd als resultaat en niet elke statement iedere keer apart.
- Behoudt van de gegevensintegriteit omdat informatie op een consistente manier wordt uitgevoerd. Een SP staat opgeslagen in de database. De normale gebruikers kunnen er gebruik van maken, maar kunnen een SP niet aanpassen.

Als onderzoek naar hoe de syntax van een SP eruitziet, ben ik eerst op zoek gegaan naar de basisbegrippen hierover. Hierover werd een goede eerste basis uitgelegd op W3Schools (W3Schools, n.d.). Om hierna de begripsvorming hierover te verruimen en complexere SP's te schrijven heb ik veel aan de informatie van Microsoft (Microsoft, Create a stored procedure, 2023) zelf gehad. Deze bron is dan ook direct geschikt voor een MSSQL-database.

Als voorbeeld van een SP op de GO-database heb ik een SP gebaseerd op het berekenen van de totale hoeveelheid geplaatste orders per retailer. Onderstaande code maakt een tijdelijke nieuwe tabel aan RETAILER_TOTAL_ORDERS om de waardes tijdelijk in te kunnen opslaan. Hierna volgt het daadwerkelijk vullen van de tabel met gegevens. De tijdelijke tabel wordt getoond en hierna weer verwijderd.

```

CREATE PROCEDURE CalculateTotalOrdersPerRetailer
AS
BEGIN
    /* Maak een tijdelijke tabel aan */
    CREATE TABLE #RETAILER_TOTAL_ORDERS
(
    RETAILER_NAME varchar(max),
    TOTAL_ORDERS int
);
    INSERT INTO #RETAILER_TOTAL_ORDERS (RETAILER_NAME, TOTAL_ORDERS)

    SELECT oh.RETAILER_NAME, count(oh.ORDER_NUMBER) AS total_order_count
    FROM Greatoutdoor.ORDER_HEADER oh
        JOIN Greatoutdoor.ORDER_DETAILS od ON oh.ORDER_NUMBER = od.ORDER_NUMBER
    GROUP BY oh.RETAILER_NAME

    SELECT * FROM #RETAILER_TOTAL_ORDERS

    /* Ruim de tijdelijke tabel weer op */
    DROP TABLE #RETAILER_TOTAL_ORDERS
END;

EXECUTE CalculateTotalOrdersPerRetailer

```

Deze SP start met CREATE PROCEDURE om aan te geven dat het hier gaat om een Stored Procedure. AS BEGIN geeft aan dat de feitelijke definitie van de procedure hiermee begint. De END op het eind geeft het einde aan. Hier tussenin staat feitelijk de “body” van de procedure. Met de CREATE TABLE RETAILER_TOTAL_ORDERS maak ik een tijdelijk tabel aan met de attributen RETAILER_NAME en TOTAL_ORDERS. Hierna met de INSERT INTO worden gegevens in de tijdelijke tabel ingevoegd. Deze gegevens worden opgehaald vanuit ORDER_HEADER en ORDER_DETAILS tabellen. De SELECT query haalt de RETAILER_NAME op en telt het aantal bestellingen voor elke retailer. Dit gebeurt met de JOIN-statement. Deze voegt de tabellen ORDER_HEADER en ORDER_DETAILS samen op basis van ORDER_NUMBER. Hierna worden deze gegroepeerd met de GROUP BY.

De SELECT in de SP toont het resultaat, waarna ik de tijdelijk tabel weer netjes opruim met de DROP methode.

Het resultaat van de EXECUTE “CalculateTotalOrdersPerRetailer” SP is zoals onderstaand screenshot. Deze wordt opgevraagd met een simpele SELECT query vanuit de SP.

<input type="checkbox"/> RETAILER_NAME	<input type="checkbox"/> TOTAL_ORDERS
Act'N'Up Fitness	240
Actiforme	43
Advanced Climbing Ltd	14
Altitudes extrêmes	258
American Home	383
Anapurna	262
Artículos de Campismo El Aquila, S.A. de C.V.	821
Ausrüstungshaus Globetrotter	451
Beach Beds Pty Ltd.	487
Bellini	119
Beter Buitenleven	1383
Blue Mountains Golfing Company	41
Brambilla	132
Buena Vista Eyeware	22
By A Thread	433
Camping Equipment Online	188
Camping Sauvage	722
Campingspecialisten	920
Chen Yu Enterprise Co.,	96
China Multi-channel Sports Co., LTD.	83
Chuei Hyakkaten	68
Connor Department Store	264
Consumer Club	867
Cordages Discount	244
Der Fitness-Doktor	206
Die Fitness-Experten	54
Die Zeltstadt	432

Functies

Voor het onderzoek naar SQL Functions heb ik op het internet diverse bronnen kunnen vinden die uitleg geven over SQL Functions

Tabel 1 - Geraadpleegde bronnen

Onderwerp	Bron
User-defined functions	(Factor, SQL Server User-Defined Functions, n.d.)
MS SQL functions	(Microsoft, What are the SQL database functions?, 2023)
SQL functions	(Papiernik, 2022)

Ook is er vanuit de les op school aandacht besteed aan wat functions zijn, ook in relatie tot Stored Procedures en Views. Deze vergelijkingen werden daar sterk vereenvoudigd toegelicht door het in een tabel te tonen. Ik vond dit zeer nuttig.

Er zijn grofweg twee type Functions. De scalar functions en de Table Valued functions.

- Scalar function: Een type functie dat één waarde retourneert. Een scalar functie accepteert een of meerder parameters en voeren een berekening uit om een enkel waarde terug te geven. Een Scalar functie kan dus handig zijn wanneer je op zoek bent naar een enkele waarde vanuit een SQL-query.
- Table-Valued function: Hierin zijn twee types te onderscheiden.
 - o Inline Table-Valued functions: Dit is een functietype dat een tabelresultaat retourneert. Dit type functie accepteert inputparameters en retourneert een tabel met rijen en kolommen.
 - o Multi-Statement Table-Valued functions: Dit type functie is vergelijkbaar met de Inline afgeleide. Echter kan dit type meer complexe logica bevatten omdat dit meerdere SQL-statements kan bevatten. Ook retourneert dit type een tabelresultaat.

Functie 1

Een functie die geschreven zou kunnen worden is antwoord krijgen op de vraag, geef per production_cost met een prijsrange als category welke producten in de opgegeven prijsrange liggen. Met andere woorden, geef alle producten die voorkomen in de opgegeven prijsrange. Deze prijsrange zou gecategoriseerd kunnen worden. Dit is nuttig wanneer klanten op zoek zijn naar producten in een bepaalde prijsklasse en hierop te filteren.

Ook kan een bedrijf deze functie gebruiken om producten te selecteren op basis van hun productiekosten om rapportages en analyses te genereren. Zo kan een bedrijf producten identificeren die binnen een bepaalde kostenmarge vallen.

De function ziet er als volgt uit. Het resultaat geeft alle producten aan in de prijsrange van 10 tot 100 eur.

```
CREATE FUNCTION dbo.GetProductsByProductionCostCategory(@minPrice FLOAT, @maxPrice FLOAT)
RETURNS TABLE
AS
RETURN
(
    SELECT *
    FROM Greatoutdoor.PRODUCT
    WHERE PRODUCTION_COST BETWEEN @minPrice AND @maxPrice
);

SELECT PRODUCT_NAME, PRODUCTION_COST
FROM GetProductsByProductionCostCategory(10, 100)
ORDER BY PRODUCTION_COST DESC
```

Deze functie accepteert 2 parameters, minPrice en maxPrice. Deze worden gebruikt om de minimum- en maximumproductiekosten op te geven. Met de RETURNS TABLE geef ik aan dat er een tabel met resultaten zal worden geretourneerd. De AS RETURN is het startpunt. Alles wat hierin wordt uitgevoerd is de “body” van de functie. De SELECT statement haalt producten op uit de tabel PRODUCT, waarvan de productiekosten (PRODUCTION_COST) binnen het opgegeven prijsrange vallen. De BETWEEN wordt gebruikt om te controleren de kosten binnen de grenzen van minPrice en maxPrice liggen.

<input type="checkbox"/> PRODUCT_NAME	<input type="checkbox"/> PRODUCTION_COST
Seeker Extreme	94.12
Seeker 50	92.58
Hibernator	86
TrailChef Deluxe Cook Set	85.11
Blue Steel Max Putter	81.8
Course Pro Golf Bag	80
Polar Sports	80
Edge Extreme	80
Seeker 35	79.19
Glacier GPS	78.55
Hibernator Self - Inflating Mat	78.55
TrailChef Double Flame	75
Polar Ice	73.33
Polar Extreme	72.5
Hibernator Camp Cot	65.33
Canyon Mule Climber Backpack	62.5

Functie 2

En ander voorbeeld van een functie die nuttig kan zijn voor een bedrijf als GO is een functie dat de totale kosten per order kan weergeven. Dit kan het bedrijf een indruk geven hoe vaak, en voor hoeveel geld een klant een order plaatst. Dit kan handig zijn om gericht kortingen te geven wanneer een klant een vaste klant wordt en veel geld besteed aan geplaatste orders.

Onderstaand is deze functie uitgewerkt. In deze functie kan je een specifiek ordernummer opgeven waarvan je de gegevens op wilt halen. In dit voorbeeld heb ik ORDER_NUMBER “1153” opgegeven, waarna het één regel retourneert met de totale kosten voor die specifieke order.

```

CREATE FUNCTION dbo.CalculateOrderTotalCost (
    @OrderNumber INT
)
RETURNS DECIMAL(18, 2)
AS
BEGIN
    DECLARE @TotalCost DECIMAL(18, 2);

    SELECT @TotalCost = SUM(QUANTITY * UNIT_COST)
    FROM Greatoutdoor.ORDER_DETAILS
    WHERE ORDER_NUMBER = @OrderNumber;

    RETURN @TotalCost;
END;

DECLARE @OrderNumber INT;
SET @OrderNumber = 1153;

SELECT
    @OrderNumber AS ORDER_NUMBER,
    dbo.CalculateOrderTotalCost( @OrderNumber: @OrderNumber) AS TotalCost;

SELECT OH.ORDER_NUMBER, dbo.CalculateOrderTotalCost( @OrderNumber: 1153) AS TotalCost
FROM Greatoutdoor.ORDER_HEADER OH

```

Hoe deze functie werkt is dat het de totale kosten berekent van een specifieke bestelling door alle relevante rijen in de tabel ORDER_DETAILS op te halen, de kosten per eenheid te vermenigvuldigen met de hoeveelheid van elk product in de bestelling en de resultaten op te tellen. Het resultaat is een decimaal getal met precies 18 cijfers en 2 cijfers achter de komma. De laatste SELECT query retourneert onderstaande tabel. Dit is het belangrijkste deel van de functie. De query haalt alle rijen op uit de tabel ORDER_DETAILS waar het ORDER_NUMBER overeenkomt met de opgegeven ordernummer @OrderNumber. Voor elke rij wordt het product van QUANTITY en UNIT_COST berekend en opgesteld met de reeds berekende kosten. Het resultaat van deze berekening wordt toegewezen aan de variabele TotalCost.

Onderstaand het resultaat na uitvoering van deze functie.

<input type="checkbox"/> ORDER_NUMBER	<input type="checkbox"/> TotalCost
1153	15556.32

Figuur 4 - Resultaat TotalCost voor ORDER_NUMBER "1153"

In relatie tot de eerder gemaakte [complex query's](#) kan afgeleid worden dat functies meer voordelen kunnen bieden. Zo kunnen functies bijdragen in de herbruikbaarheid. Om iedere keer de complexe query opnieuw te moeten maken kost veel tijd. Ook kunnen er veranderingen ontstaan tussen de eerder geschreven query en de nieuwe. Functies kunnen ook modulair opgebouwd worden en bevordert hierdoor de leesbaarheid. Door kleinere stukjes logica te schrijven en op te delen wordt de logica overzichtelijker. Het draagt dan niet alleen bij aan het overzicht maar ook het begrijpen van de code wordt hiermee bevorderd. Zoals eerder gezegd is het herschrijven van complexe query's

foutgevoelig, een functie is wat dat betreft consistentiever. Het ligt vast en zal iedere keer dezelfde logica uitvoeren. Ook zijn functies beter te onderhouden en kan men ervoor kiezen om versiebeheer hierop toe te passen.

Edwin

Stored Procedure

Door de toegenomen energieprijzen en hogere inkoopkosten van grondstoffen zoals metaal hebben enkele leveranciers de prijzen moeten verhogen. Hierdoor is Great Outdoor genoodzaakt een prijsverhoging door te voeren van 10%.

De afdeling inkoop van Great Outdoor (GO) wil deze aanpassing op de prijs doorvoeren en het betreft hier alle kookgerei artikelen van het product type 'Cooking Gear'

De afdeling inkoop heeft aan de IT afdeling gevraagd dit aan te passen op de database.

Om dit te realiseren heb ik een stored procedure bedacht die a.d.h.v een product type en gewenst percentage de prijs van het product aanpast. Het gaat hier om kolom PRODUCTION_COST uit de tabel PRODUCT.

De product typen zijn te vinden in kolom PRODUCTTYPE_EN uit tabel PRODUCT_TYPE met de lk contoleer of het product type bestaat en kom op de eerste regel het betreffende product type 'Cooking Gear' tegen op de eerste rij.

```
SQLQuery1.sql - DK...(ASRLAN\EDR (52))*
select * from dbo.PRODUCT_TYPE;
```

The screenshot shows a SQL query window with the following results:

	PRODUCT_TYPE_CODE	PRODUCT_LINE_CODE	PRODUCT_TYPE_EN
1	1	1	Cooking Gear
2	2	1	Tents
3	3	1	Sleeping Bags
4	4	1	Packs
5	5	1	Lanterns

Afb. tabel PRODUCT_TYPE.

De productieprijs is te vinden in kolom PRODUCTION_COST in tabel PRODUCT en via de Foreign key kolom is te zien dat het gaat om tien producten.

```
SQLQuery1.sql - DK...(ASRLAN\EDR (52))*
select * from dbo.PRODUCT;
```

The screenshot shows a SQL query window with the following results:

	PRODUCT_NUMBER	INTRODUCTION_DATE	PRODUCT_TYPE_CODE	PRODUCTION_COST	MARGIN	PRODUCT_IMAGE	PRODUCT_NAME
1	1	1995-02-15 00:00:00.0000000	1	4	33	P01CE1CG1.jpg	TrailChef Water Bag
2	2	1995-02-15 00:00:00.0000000	1	9,22	23	P02CE1CG1.jpg	TrailChef Canteen
3	3	1995-02-15 00:00:00.0000000	1	15,93	28	P03CE1CG1.jpg	TrailChef Kitchen Kit
4	4	1995-02-15 00:00:00.0000000	1	5	28	P04CE1CG1.jpg	TrailChef Cup
5	5	1995-02-15 00:00:00.0000000	1	34,97	3	P05CE1CG1.jpg	TrailChef Cook Set
6	6	1997-03-05 00:00:00.0000000	1	85,11	28	P06CE1CG1.jpg	TrailChef Deluxe Cook Set
7	7	1995-02-15 00:00:00.0000000	1	46,38	28	P07CE1CG1.jpg	TrailChef Single Flame
8	8	1997-03-05 00:00:00.0000000	1	75	41	P08CE1CG1.jpg	TrailChef Double Flame
9	9	1997-03-05 00:00:00.0000000	1	9	25	P09CE1CG1.jpg	TrailChef Kettle
10	10	1995-02-15 00:00:00.0000000	1	10	4	P10CE1CG1.jpg	TrailChef Utensils
11	11	1995-02-15 00:00:00.0000000	2	250	28	P11CE1TN2.jpg	Star Lite
12	12	1997-03-05 00:00:00.0000000	2	473,07	23	P12CF1TN2.jpg	Star Dining

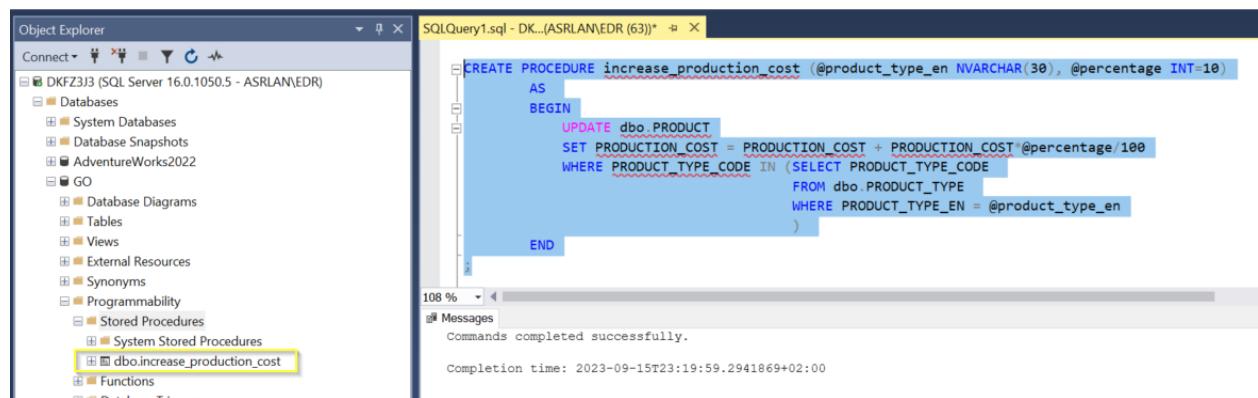
Afb. tabel PRODUCT_TYPE.

Omdat de tabel PRODUCT_TYPE en PRODUCT een relatie hebben in de database van Great Outdoor zijn deze te koppelen. Door het meegeven van een product type kan vervolgens de kolom PRODUCTION_COST worden bijgewerkt met een UPDATE statement in de stored procedure.

De stored procedure is al volgt gedefinieerd:

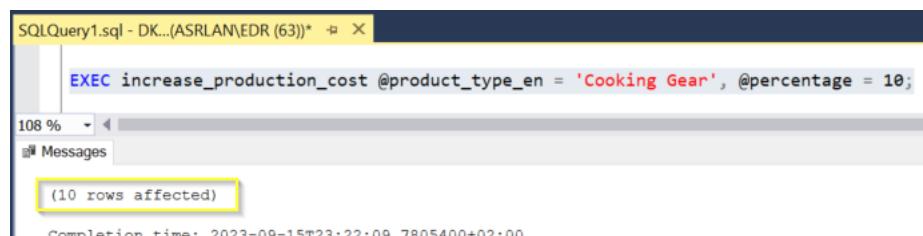
```
CREATE PROCEDURE increase_production_cost (@product_type_en NVARCHAR(30), @percentage INT=10)
AS
BEGIN
    UPDATE dbo.PRODUCT
    SET PRODUCTION_COST = PRODUCTION_COST +
PRODUCTION_COST*@percentage/100
        WHERE PRODUCT_TYPE_CODE IN (SELECT PRODUCT_TYPE_CODE
                                    FROM dbo.PRODUCT_TYPE
                                    WHERE PRODUCT_TYPE_EN = @product_type_en
                                )
END
;
```

Na het uitvoeren van de code is de stored procedure zichtbaar aan de linkerzijde in de Object Explorer via *Programmability -> Stored Procedures*.



Afb. Aangemaakte procedure 'increase_production_cost' op de database.

De stored procedure kunnen we gebruiken door de procedure uit te voeren d.m.v het EXEC statement en geven het product type en percentage mee.



Afb. Uitvoeren van stored procedure increase_production_cost.

De stored procedure heeft nu tien rijen bijgewerkt.

Voordelen Stored Procedure

Het voordeel van het gebruik van een stored procedure (Petković, 2020) is dat we deze kunnen hergebruiken. Als we opnieuw een prijs verhoging moeten doorvoeren kunnen we de procedure opnieuw gebruiken. Ook voor andere product typen is de procedure bruikbaar door een ander product type mee te geven.

Verder komt het gebruik van stored procedures de performance ten goede omdat deze precompiled wordt opgeslagen in de database. De stored procedure word in deze voorgecompileerde vorm opgeslagen, zodat bij executie ervan deze niet steeds opnieuw gecompileerd hoeft te worden.

Table Function Inventory Count

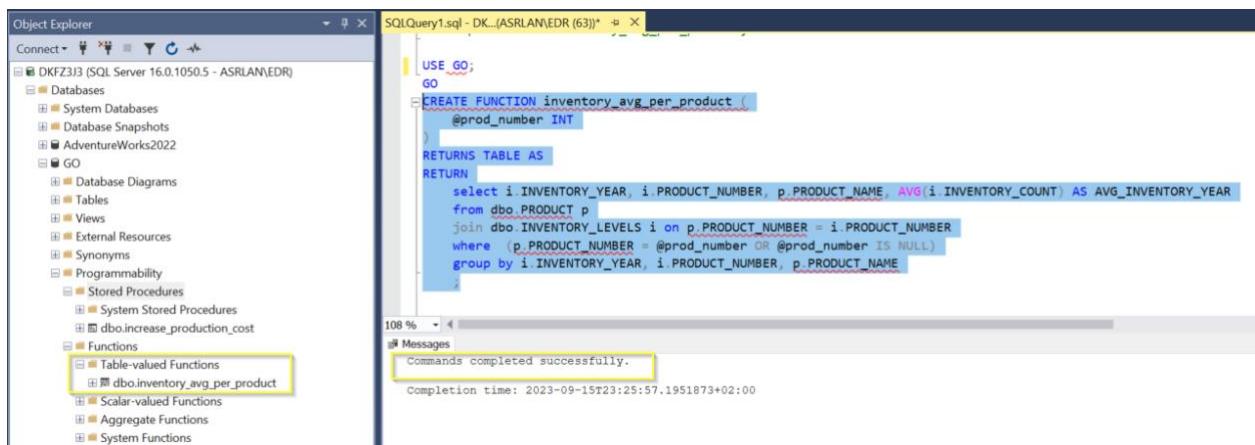
De magazijn manager van Great Outdoors ziet de beschikbare vrije ruimte in het magazijn steeds kleiner worden en wil graag weten of de voorraad is toegenomen t.o.v voorgaande jaren. Door deze informatie wil de manager bepalen of er over een tijdje wel voldoende ruimte is.

Om deze informatie uit de database te halen heb ik de volgende Table Function gemaakt. De functie haalt het gemiddelde aantal producten uit de voorraad op per product per jaar

```
CREATE FUNCTION inventory_avg_per_product (
    @prod_number INT
)
RETURNS TABLE AS
RETURN
    select i.INVENTORY_YEAR, i.PRODUCT_NUMBER, p.PRODUCT_NAME,
    AVG(i.INVENTORY_COUNT) AS AVG_INVENTORY_YEAR
    from dbo.PRODUCT p
    join dbo.INVENTORY_LEVELS i on p.PRODUCT_NUMBER = i.PRODUCT_NUMBER
    where (p.PRODUCT_NUMBER = @prod_number OR p.PRODUCT_NUMBER = NULL)
    group by i.INVENTORY_YEAR, i.PRODUCT_NUMBER, p.PRODUCT_NAME
;
```

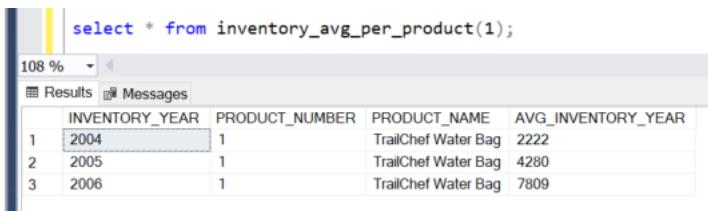
Door het product nummer uit de tabel producten mee te geven aan de functie worden de voorraad gegevens opgehaald. Als er niets aan de functie wordt meegegeven worden voor alle producten de voorraad gegevens getoond.

Na het uitvoeren van de functie is deze zichtbaar aan de linkerzijde in de Object Explorer via *Programmability -> Functions -> Table-valued Functions*.



Afb. Aangemaakte functie 'inventory_avg_per_product' op de database.

De table function is als tabel te gebruiken, en door een product nummer mee te geven worden de voorraad gegevens getoond.



A screenshot of a SQL query results window. The query is:

```
select * from inventory_avg_per_product(1);
```

The results show a table with four columns: INVENTORY_YEAR, PRODUCT_NUMBER, PRODUCT_NAME, and AVG_INVENTORY_YEAR. The data is as follows:

	INVENTORY_YEAR	PRODUCT_NUMBER	PRODUCT_NAME	AVG_INVENTORY_YEAR
1	2004	1	TrailChef Water Bag	2222
2	2005	1	TrailChef Water Bag	4280
3	2006	1	TrailChef Water Bag	7809

Afb. Aanroep van de table function.

Table Function Retailer City Count

De sales manager heeft gevraagd om inzichtelijk te maken hoeveel vestigingen er zijn per plaats. Door een plaats op te geven moet een lijst met vestigen worden getoond van die plaats. Om dit te realiseren is een table functie gemaakt en ziet er als volgt uit.

The screenshot shows the Object Explorer on the left with the database 'DKFZ3J' selected. In the center, a query window titled 'SQLQuery1.sql - DK... (ASRLAN\EDR (66)*' displays the T-SQL code for creating the function:

```
CREATE FUNCTION GetRetailerCityCount (@CompanyCity CHAR(50))
RETURNS TABLE
AS RETURN (select R.COMPANY_NAME, s.CITY, count(s.CITY) AS aantal_vestigingen
from RETAILER r
left join RETAILER_SITE s on s.RETAILER_CODE = r.RETAILER_CODE
where s.city = @CompanyCity
group by R.COMPANY_NAME, s.CITY
);
```

The status bar at the bottom right shows 'Completion time: 2023-10-01T09:47:41.2348632+02:00'.

Afb. Creatie van de table function *GetRetailerCityCount*.

De onderliggende query voor bijvoorbeeld Amsterdam geeft totaal 4 vestigingen terug.

The screenshot shows the results of the query:

```
select R.COMPANY_NAME, s.CITY, count(s.CITY) AS aantal_vestigingen -- join 391 / left join 391
from RETAILER r
join RETAILER_SITE s on s.RETAILER_CODE = r.RETAILER_CODE
where s.CITY = 'Amsterdam'
group by R.COMPANY_NAME, s.CITY
order by 2;
```

	COMPANY_NAME	CITY	aantal_vestigingen
1	Beter Buitenleven	Amsterdam	2
2	Get Out	Amsterdam	1
3	Holland Zonzoekers	Amsterdam	1

Afb. Query van de table function *GetRetailerCityCount*.

Door de functie uit te voeren krijgen we de vestigingen terug van de locatie Amsterdam

The screenshot shows the results of the query:

```
select * from GetRetailerCityCount('Amsterdam');
```

	COMPANY_NAME	CITY	aantal_vestigingen
1	Beter Buitenleven	Amsterdam	2
2	Get Out	Amsterdam	1
3	Holland Zonzoekers	Amsterdam	1

Afb. uitvoer van de table function *GetRetailerCityCount*.

Scalar Function

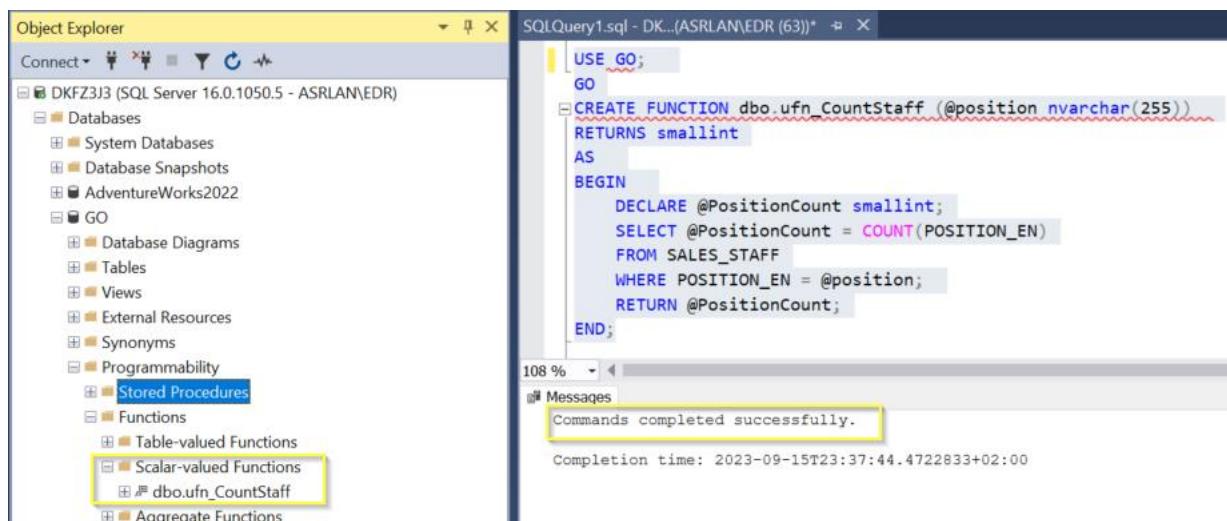
Great Outdoors heeft ook een sales afdeling voor de verkoop van alle artikelen. De general sales manager van Great Outdoors wil graag kunnen zien hoeveel personen er in dienst zijn met een bepaalde functie.

Om dit aantal uit de database te halen heb ik een user defined scalar function gemaakt. De functie haalt het aantal personen in dienst op van een opgegeven functienaam.

```
CREATE FUNCTION dbo.ufn_CountStaff (@position nvarchar(255))
RETURNS smallint
AS
BEGIN
    DECLARE @PositionCount smallint;
    SELECT @PositionCount = COUNT(POSITION_EN)
    FROM SALES_STAFF
    WHERE POSITION_EN = @position;
    RETURN @PositionCount;
END;
```

Door de functienaam uit de tabel SALES_STAFF mee te geven aan de functie wordt het aantal personeelsleden opgehaald voor die functie.

Na het uitvoeren van de functie is deze zichtbaar aan de linkerzijde in de Object Explorer via *Programmability -> Functions -> Scalar-valued Functions*.



Afb. Aangemaakte functie 'ufn_CountStaff' op de database.

De scalar function kan gebruikt worden door een functie naam mee te geven en laat vervolgens het aantal personen zien met de opgegeven functie.

The screenshot shows a SQL query window titled 'SQLQuery1.sql - DK... (ASRLAN\EDR (63))'. The query is:

```
SELECT dbo.ufn_CountStaff('Regional Manager') AS Aantal_personeel;
```

The results pane shows a single row with the column 'Aantal_personeel' containing the value '2'.

Afb. Aanroep van de scalar function.

Lesley-Ann

Na het analyseren van de bronnen heb ik een kort verlag geschreven over stored procedures en functies.

Stored Procedures

Stored procedures zijn script die in een database worden opgeslagen en snel kunnen worden uitgevoerd als dit nodig is. Het zijn SQL instructies die je kunt zien als een databaseprogramma. De stored procedure wordt uitgevoerd om een specifieke taak uit te voeren. Voorbeelden van taken zijn het bijwerken, invoegen of verwijderen van data, het ophalen van gegevens of het berekenen van cijfers of prestaties, het instellen van toegangscontroles voor een veilige gegevens toegang. (Corey, 2017)

Er zijn verschillende voordelen als je stored procedures gebruikt voor je database. Hieronder noem ik er een aantal:

- Een stored procedure kan parameters accepteren wat er voor zorgt dat ze aan te roepen zijn met verschillende soorten invoer. Dit kan helpen bij het uitvoeren van repetitieve, herhalende taken.
- Omdat de stored procedures opgeslagen worden in de database en voorheen al gecompileerd zijn kan dit tijd besparen bij het schrijven van query's en het uitvragen van informatie.
- Stored procedures kunnen worden opgeroepen vanuit verschillende delen van de applicatie(s). Dit zorgt ervoor dat de logica kan worden hergebruikt zonder dat deze gedupliceerd hoeft te worden.
- Als bedrijfslogica wordt opgeslagen in stored procedures hoeft de applicatiecode niet gewijzigd te worden als er onderhoud aan de database wordt gedaan. Dit kan onderhoud sneller en makkelijker maken.
- Stored procedures maken het mogelijk om meerdere SQL instructies als een geheel uit te voeren. Dit kan behulpzaam zijn omdat als er een fout optreedt, alle wijzigingen terug gedraaid kunnen worden met een rollback. Dit zorgt ervoor dat er geen wijzigingen in de database worden aangebracht voordat de update compleet is. (Babu, 2019) (Tewatia, 2023)

Stored procedures heeft dus vele voordelen voor de gebruiker maar er zijn ook negatieve gevolgen als de stored procedures niet goed worden onderhouden worden. Een voorbeeld hiervan is als stored procedures een beveiligingsrisico vormen. Slecht geschreven of slecht onderhouden stored procedures kunnen kwetsbaarheden veroorzaken als het gaat om veiligheid. Ook versiebeheer kan uitdagender worden omdat de stored procedures ook zorgvuldig bijgehouden moeten worden voor nieuwe versies van de software. Het is dus belangrijk dat stored procedures zorgvuldig worden bijgehouden en gedocumenteerd worden. Als dit juist gedaan wordt zullen er overwegend meer voordelen aan het gebruik van stored procedures zitten.

Ik heb een voorbeeld van een stored procedure voor de GO database bijgevoegd. In deze stored procedure worden het totaal aantal werknemers per branch, op basis van de stad weergegeven.

```

USE GODATABASE
CREATE PROCEDURE GetTotalEmployeesPerLocation
AS
BEGIN
    SELECT SB.SALES_BRANCH_CODE, SB.CITY, COUNT(SS.SALES_STAFF_CODE) AS TotalEmployees
    FROM dbo.SALES_BRANCH SB
    LEFT JOIN dbo.SALES_STAFF SS ON SB.SALES_BRANCH_CODE = SS.SALES_BRANCH_CODE
    GROUP BY SB.SALES_BRANCH_CODE, SB.CITY
    ORDER BY SB.SALES_BRANCH_CODE;
END

EXEC GetTotalEmployeesPerLocation;

```

	SALES_BRANCH_CODE	CITY	TotalEmployees
1	6	Paris	4
2	7	Milano	3
3	9	Amsterdam	5
4	13	Hamburg	4
5	14	München	3
6	15	Kista	4
7	17	Calgary	5
8	18	Toronto	5
9	19	Boston	6

Figuur 5 stored procedure

Functions

In SQL zijn er verschillende soorten functies

Scalar functies:

Scalar functies zijn user defined functies die een enkele waarde retourneren. Voorbeelden hiervan zijn een getal of datum. Vaak worden deze gebruikt binnen SQL statements wanneer een enkele waarde nodig is. Scalar functies worden gebruikt in SELECT en WHERE queries om gegevens te veranderen, berekeningen te krijgen of informatie uit te vragen. Voorbeelden hiervan zijn: het omzetten van valuta van euro naar dollar of het omzetten van temperatuur metingen van Celsius naar Fahrenheit. (D, 2021) (Scalar Functions in SQL, 2023)

Tabel valued functies:

Tabel valued functies zijn user defined functies die een tabel resultaat retourneren. Dit kan handig zijn wanneer je een verzameling van data of logica wil inzien. Tabel valued functies worden gebruikt met de FROM queries. Dit staat toe om resultaten te filteren, bewerken of samenvoegen. Een voorbeeld hiervan is een tabel retourneren op basis van specifieke criteria.

Binnen de tabel valued functies word er onderscheid gemaakt tussen twee soorten functies; de inline en de multi statement functies. Het verschil tussen deze twee soorten functies is de manier waarop ze logica verwerken.

- **Inline tabel valued functies:** inline gedefinieerd, bevat meestal een enkele SELECT query en wordt direct in een SQL query gebruikt. Vaak geschikt voor eenvoudigere taken.

- Multi statement valued functies: bevat meerdere SQL statements, gebruikt BEGIN en END om de logica af te bakenen. Geschikt voor complexere taken omdat ze meerdere statements kunnen bevatten. Staat toe om complexe logica in kleinere stukken op te splitsen. (Erkec, 2019)

In mijn voorbeeld van een scalar functie heb ik een query geschreven die de naam van een land ophaalt op basis van de gegeven landcode. De functie zoekt de naam van een land op in de database en print deze. Er wordt een enkele return geprint tenzij de functie aangepast zou worden om meerdere country codes weer te geven.

```

CREATE FUNCTION dbo.GetCountryName3 (@CountryCode int)
RETURNS nvarchar(50)
AS
BEGIN
    DECLARE @CountryName nvarchar(50)

    SELECT @CountryName = COUNTRY
    FROM COUNTRY
    WHERE COUNTRY_CODE = @CountryCode

    RETURN @CountryName
END

DECLARE @CountryCode int = 1;
DECLARE @CountryName nvarchar(50);

SELECT @CountryName = dbo.GetCountryName3(@CountryCode);
PRINT 'Country Name: ' + @CountryName;

```

Figuur 6 Scalar functie landcode

```

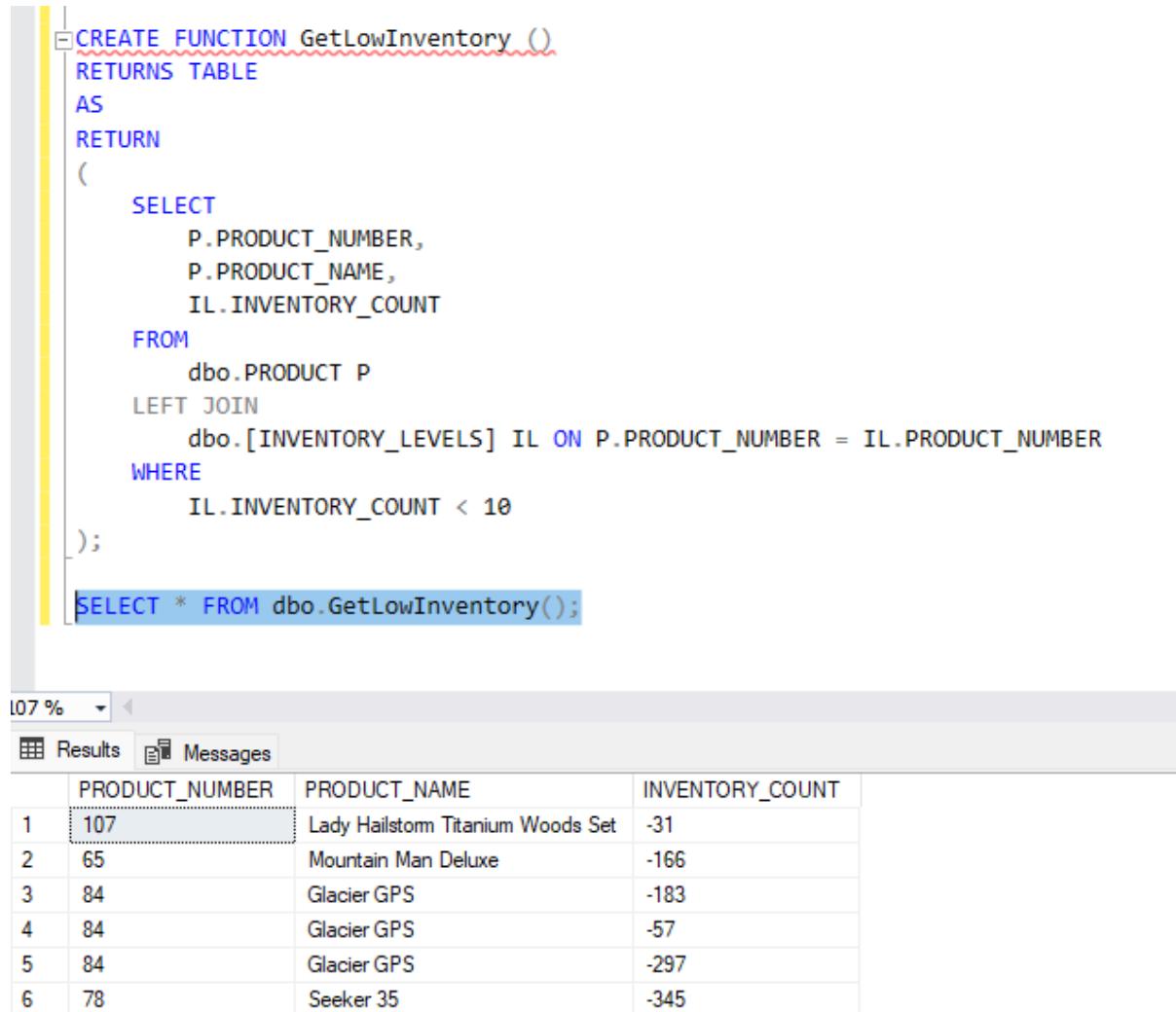
DECLARE @CountryCode int = 2;
DECLARE @CountryName nvarchar(50);

SELECT @CountryName = dbo.GetCountryName3(@CountryCode);
PRINT 'Country Name: ' + @CountryName;

```

Figuur 7 Scalar functie land code voorbeeld 2

In mijn voorbeeld van een table valued functie heb ik een query geschreven die een tabel retourneert met producten waarvan de voorraad minder dan 10 is. Dit is een voorbeeld van een table valued functie omdat er een tabel geretourneerd word en geen enkele eenheid. De kolommen in het resultaat bevat het productnummer, de productnaam en de huidige voorraad. Deze query zou ook makkelijk herbruikt kunnen worden om bijvoorbeeld hoge inventory levels op te vragen.



```

CREATE FUNCTION GetLowInventory()
RETURNS TABLE
AS
RETURN
(
    SELECT
        P.PRODUCT_NUMBER,
        P.PRODUCT_NAME,
        IL.INVENTORY_COUNT
    FROM
        dbo.PRODUCT P
    LEFT JOIN
        dbo.[INVENTORY_LEVELS] IL ON P.PRODUCT_NUMBER = IL.PRODUCT_NUMBER
    WHERE
        IL.INVENTORY_COUNT < 10
);
SELECT * FROM dbo.GetLowInventory();

```

	PRODUCT_NUMBER	PRODUCT_NAME	INVENTORY_COUNT
1	107	Lady Hailstorm Titanium Woods Set	-31
2	65	Mountain Man Deluxe	-166
3	84	Glacier GPS	-183
4	84	Glacier GPS	-57
5	84	Glacier GPS	-297
6	78	Seeker 35	-345

Figuur 8 Table valued function low inventory

De link van functies naar de complexe query's uit opdracht C

De vraag aan ons studenten was of wij de query's uit opdracht C eenvoudiger zouden maken op basis van functies. Mijn kijk op functie is na het literatuur onderzoek en het schrijven van de query's heel positief en ik denk zeker dat we functies zouden kunnen inzetten in de complexe query's uit opdracht C. Ik kan hier verschillende redenen bedenken waarom de complexe query's verbeterd zouden worden met het gebruik van functies. Allereerst is het gebruiken van functies leesbaarder dan een lange query, met de functies kun je de query opsplitsen en deze overzichtelijker en leesbaarder maken. Ook kan dit helpen bij het begrijpen van code, het opsplitsen van code maakt gelijk duidelijk welke delen van de query bij elkaar horen. Dit helpt ook bij het overdragen van code of samenwerken, als de code goed te begrijpen is kunnen de mensen die na jou aan de database werken sneller en beter begrijpen wat jou werk inhoud.

Zoals ik eerder beschreef aan het begin van dit hoofdstuk bevorderen functies herbruikbaarheid omdat ze makkelijk op te slaan en herschrijven zijn. Daarnaast zijn functies minder fout gevoelig. En hoewel dit nu niet op onze opdracht van toepassing is maken ze ook functiebeheer gemakkelijker. Al met al denk ik dat als we meer tijd hadden het zeker een leuke opdracht was geweest om de complexe query's met de groep te herschrijven met functies.

Sander

Stored Procedures

Een Stored Procedure (hierna: SP) is een verzameling van één of meerdere SQL-statements (Microsoft, 2023). Een SP lijkt in bepaalde opzichten op een *functie* zoals bekend van veel andere programmeertalen. Een SP kan bijvoorbeeld:

- Inputparameters hebben
- Een of meerdere waardes teruggeven aan het einde
- Andere SP's aanroepen

Voordelen van het gebruik van SP's zijn onder andere:

- Minder netwerkverkeer, doordat enkel de opdracht om de SP te starten over het netwerk naar de SQL-server gestuurd wordt; De statements worden vervolgens achtereenvolgens door de SQL-server zelf uitgevoerd
- Hergebruik van SQL-statements mogelijk maken
- Client-applicaties hoeven niet altijd meer aangepast te worden wanneer de databasestructuur aangepast is, wanneer data en gedrag allebei in de database zitten (en de client-applicatie slechts de SP oproept per naam) (Oracle Corporation, 1999)

Er zijn verschillende typen SP's. Binnen dit hoofdstuk focus ik me op "User-defined" SP's. Dat zijn SP's die door een gebruiker zelf aan te maken zijn waarbij de gebruiker de statements aanlevert. Een ander belangrijk type SP zijn de System procedures. Dit zijn SP's die de SQL-server zelf beheert.

Voorbeeld voor de GO-database

Kijkend naar het oorspronkelijke database schema van GO, zou ik me kunnen voorstellen dat orders soms foutief worden aangemaakt of om andere redenen geannuleerd moeten worden. Aangezien dit vermoedelijk vaker voorkomt zou het handig kunnen zijn dat er een gestandaardiseerde procedure voor is. Zo'n procedure zou onder andere kunnen bestaan uit het wissen van de order uit de database.

Om hierin te voorzien zou een client-applicatie bijvoorbeeld middels DELETE statements eerst de ORDER_DETAILS regels kunnen verwijderen en daarna de ORDER_HEADER regel. Een alternatief zou zijn om een SP hiervoor te gebruiken. Deze SP zou als inputparameter het ORDER_NUMBER hebben en vervolgens zelf de DELETE statements kunnen formuleren en uitvoeren. Hieronder een voorbeeld van een SP die dit doet (Microsoft, 2023).

```

CREATE OR ALTER PROCEDURE go_delete_order
    @ORDER_NUMBER int
AS
BEGIN TRANSACTION
    -- Delete ORDER_HEADER record
    DELETE ORDER_HEADER WHERE ORDER_NUMBER = @ORDER_NUMBER
    IF @@ROWCOUNT <> 1
        BEGIN
            PRINT 'Expected 1 ORDER_HEADER row to be affected. Perhaps the order was
already deleted?'
            RETURN(1)
        END

    -- Delete ORDER_DETAILS records
    DELETE ORDER_DETAILS WHERE ORDER_NUMBER = @ORDER_NUMBER
COMMIT TRANSACTION
RETURN (0)
GO

```

Deze SP voert de DELETE-statements uit binnen een transactie. Dit zorgt ervoor, dat bij een fout in een van de statements (waarbij het uitvoeren van de SP stopt) de voorgaande statements ook niet worden doorgevoerd in de database (Microsoft, 2023). Hiermee is het dus gegarandeerd dat het verwijderen van de order in zijn geheel is gebeurd aan het einde, of dat het verwijderen volledig geannuleerd is. Dit laatste betekent dat de database weer in dezelfde staat is, als voor het starten van de SP.

De SP kan als volgt uitgevoerd worden:

```

DECLARE @ret_code int
EXECUTE @ret_code = go_delete_order @ORDER_NUMBER = 1234
SELECT @ret_code

```

Hierbij is “1234” het ordernummer. Het SELECT-statement zorgt ervoor dat de return value teruggegeven wordt als scalar value. Deze zal gelijk zijn aan 1 zijn wanneer het mislukt is en gelijk zijn aan 0 zijn wanneer het gelukt is.

Functions

SQL-server biedt naast SP's de mogelijkheid om *functions* te definieren. Functions hebben een handig voordeel ten opzichte van SP's. Ze kunnen namelijk als tabel gequeried worden, waardoor je bijvoorbeeld ook op het resultaat van een function kunt JOIN'en. Ze hebben ook een belangrijk nadeel. Ze kunnen geen wijzigingen (DELETE, CREATE, DROP, etc.) aan de database doen (Job Habraken, 2023).

SQL-server kent drie typen functions:

- Scalar functions, dit type retourneert een enkele waarde; Deze waarde kan alle data typen (zoals int, varchar) hebben behalve *text*, *ntext*, *image*, *cursor* en *timestamp*
- Table-values functions, dit type retourneert een tabel
- System functions, dit zijn hulpfuncties die gedefinieerd zijn binnen SQL-server zelf en kunnen bijvoorbeeld operaties op tekst of getallen doen of kunnen gegevens uit de System Catalog teruggeven

Binnen dit hoofdstuk focus ik mij op de eerste twee.

Functie 1 – Eenvoudiger maken eerdere query

Nadat ik functies onder de loep had genomen, ben ik gaan uitzoeken welke query's hierdoor makkelijker gemaakt zouden kunnen worden. De query "Hoeveelheid werk per ordermethode" (zie hoofdstuk 2 - Queries) kan simpeler hiermee.

We zouden bijvoorbeeld de eerste CTE kunnen afsplitsen. Een functie die de taak van deze CTE overneemt zou er dan als volgt uitzien:

```
CREATE OR ALTER FUNCTION go_total_order_value (@ORDER_NUMBER int)
RETURNS TABLE
AS
RETURN
(
    SELECT o_details.ORDER_NUMBER,
           SUM(o_details.QUANTITY * o_details.UNIT_PRICE) TOTAL_VALUE
    FROM ORDER_HEADER o_header
    INNER JOIN ORDER_DETAILS o_details
        ON o_details.ORDER_NUMBER = o_header.ORDER_NUMBER
   WHERE o_header.ORDER_NUMBER = @ORDER_NUMBER
  GROUP BY o_details.ORDER_NUMBER
)
GO
```

De tweede CTE kunnen we vervolgens als volgt afsplitsen:

```
CREATE OR ALTER FUNCTION go_order_method_to_amount_of_manual_work
(@ORDER_METHOD_CODE int)
RETURNS TABLE
AS
RETURN
(
    SELECT method.ORDER_METHOD_CODE,
           CASE
               WHEN method.ORDER_METHOD_CODE = 1 THEN 'Least manual work'
               WHEN method.ORDER_METHOD_CODE = 2 THEN 'Most manual work'
               WHEN method.ORDER_METHOD_CODE = 3 THEN 'Average manual work'
               WHEN method.ORDER_METHOD_CODE = 4 THEN 'Average manual work'
               WHEN method.ORDER_METHOD_CODE = 5 THEN 'Least manual work'
               WHEN method.ORDER_METHOD_CODE = 7 THEN 'Most manual work'
               WHEN method.ORDER_METHOD_CODE = 8 THEN 'Most manual work'
               ELSE 'UNKNOWN' END AMOUNT_OF_MANUAL_WORK
    FROM ORDER_METHOD method WHERE ORDER_METHOD_CODE = @ORDER_METHOD_CODE
)
```

Het voordeel hiervan kan zijn dat de uiteindelijk hoofdquery een stuk eenvoudiger wordt. Verder zijn de afgesplitste zaken nu los van elkaar bij te werken en kunnen ze deze stukjes eventueel hergebruikt worden in andere queries.

De query uit hoofdstuk 2 - Queries zal er nu als volgt uitzien:

```

SELECT work.AMOUNT_OF_MANUAL_WORK,
       COUNT(*) ORDER_COUNT,
       CAST(SUM(order_v.TOTAL_VALUE) AS DECIMAL(20, 2)) AS TOTAL_REVENUE
FROM   ORDER_METHOD method
       INNER JOIN ORDER_HEADER order_h
              ON order_h.ORDER_METHOD_CODE = method.ORDER_METHOD_CODE
CROSS APPLY go_total_order_value(order_h.ORDER_NUMBER) order_v
CROSS APPLY go_order_method_to_amount_of_manual_work(order_h.ORDER_METHOD_CODE)
work
GROUP BY work.AMOUNT_OF_MANUAL_WORK
ORDER BY ORDER_COUNT DESC

```

De query is hiermee een stuk korter geworden. Wel is het niet meer mogelijk om INNER JOIN te gebruiken, maar is dat CROSS APPLY geworden omdat een JOIN niet mogelijk is in combinatie met functies met parameters.

Functie 2 – Lijst van managers voor medewerker

GO is een onderneming met managers. Veel personeelsleden werken onder een manager en deze manager mogelijk weer onder een andere manager. Het zou zomaar handig kunnen zijn om de lijn van managers op te halen van een medewerker. Bijvoorbeeld om te bepalen wie er ingelicht moet worden bij een probleemsituatie.

De volgende functie kan deze informatie ophalen:

```

CREATE OR ALTER FUNCTION EMPLOYEE_MANAGEMENT_TREE (@SALES_STAFF_CODE int)
RETURNS TABLE
AS
RETURN
(
    WITH EMPLOYEE_TREE_CTE AS
    (
        SELECT MANAGER_CODE
        FROM SALES_STAFF
        WHERE SALES_STAFF_CODE = @SALES_STAFF_CODE
        UNION ALL
        SELECT sst.MANAGER_CODE
        FROM SALES_STAFF sst
        INNER JOIN EMPLOYEE_TREE_CTE et ON et.MANAGER_CODE = sst.SALES_STAFF_CODE
        WHERE sst.SALES_STAFF_CODE <> sst.MANAGER_CODE -- Verify the manager is
not their own manager
    )
    SELECT MANAGER_CODE
    FROM EMPLOYEE_TREE_CTE
)

```

De recursive CTE maakt mogelijk dat dit werkt met een (nagenoeg, zoveel als de SQL Server aan recursie toelaat) oneindige diepte in managementstructuur. Wanneer een medewerker geen manager heeft, zal de query een lege lijst teruggeven. De medewerker zelf komt dus niet voor in het resultaat.

De functie kan als volgt aangeroepen worden in een query:

```
SELECT SALES_STAFF_CODE CODE, CONCAT(LAST_NAME, ' ', FIRST_NAME) FULL_NAME  
FROM EMPLOYEE_MANAGEMENT_TREE (1234) management  
INNER JOIN SALES_STAFF staff ON staff.SALES_STAFF_CODE = management.MANAGER_CODE
```

Hierbij is '1234' op regel 2 de SALES_STAFF_CODE waarvoor de managementstructuur wordt opgehaald. Voor het gemak van de lezer wordt ook de volledige naam van de medewerkers opgehaald naast de medewerkerscode.

Tests

Thierry

Voor het waarborgen van de kwaliteit, betrouwbaarheid en prestaties van een database, is het nodig om tests te schrijven. Testen bieden een gestructureerde aanpak om functionaliteit van een database en de bijhorende applicaties te valideren. Ze stellen ons in staat verschillende scenario's te simuleren, van eenvoudige zoekopdrachten tot complexe transacties, en ervoor te zorgen dat alles werkt zoals verwacht.

Het doel van testen is niet alleen om te controleren of een database functioneel is, maar ook om prestaties te evalueren, mogelijke fouten op te sporen en de reactie van het systeem onder verschillende belastingen te meten.

In deze opdracht heb ik tests geschreven voor een eerder door mij geschreven Stored Procedure. Ook zijn de functies die ik eerder in dit document heb gemaakt, getest. De verwijzingen naar de eerdere koppen zijn opgenomen in de tests.

Stored Procedure test

Voor deze test verwijst ik graag naar mijn eerdere uitwerking van het [Stored Procedure voorbeeld](#). Hierin staat ook een screenshot van het resultaat in een tabel. Hiernaar verwijst ik dan ook graag naar.

In deze test wordt een steekproef genomen door een test query uit te voeren en te kijken of het resultaat vanuit de Stored Procedure klopt.

Om te testen of de waardes kloppen heb ik rij 10 in de tabel met de naam “Bellini”, genomen als voorbeeld genomen voor de steekproef. Hiervoor heb ik een query geschreven die een telling doet op het totaal aantal geplaatste orders.

```
SELECT oh.RETAILER_NAME, count(oh.ORDER_NUMBER) AS total_order_count
FROM Greatoutdoor.ORDER_HEADER oh
JOIN Greatoutdoor.ORDER_DETAILS od ON oh.ORDER_NUMBER = od.ORDER_NUMBER
WHERE oh.RETAILER_NAME = 'Bellini'
GROUP BY oh.RETAILER_NAME
```

De SELECT query selecteert de RETAILER_NAME uit de ORDER_HEADER tabel en telt het aantal bestellingen van de opgegeven ORDER_NUMBER. Deze wordt als resultaat opgenomen in de total_orders_count kolom. De JOIN combineert de gegevens vanuit de ORDER_DETAILS en ORDER_HEADER tabel en koppelt ze op basis van het ORDER_NUMBER zodat de gegevens van bestellingen in beide tabellen dezelfde ORDER_NUMBER hebben. De WHERE is de voorwaarde waarmee gefilterd wordt. Als laatste wordt de GROUP BY gebruikt om de resultaten te groeperen op basis van RETAILER_NAME. In dit geval is dit de retailer “Bellini”, dus zal het resultaat maar 1 rij bevatten met het totaal aantal bestellingen specifiek van deze retailer.

Onderstaand het resultaat van deze test. Hieruit valt op te maken dat de uitkomst voldoet aan de werking van de eerder gemaakte SP.

RETAILER_NAME	total_order_count
Bellini	119

Figuur 9 - Resultaat SP test

Function tests

Functie 1

Voor deze test verwijst ik graag naar mijn eerdere uitwerking van een [Function voorbeeld](#). Hierin staat een screenshot van het resultaat na uitvoering van de function in een tabel. In deze test heb ik een steekproef gedaan voor het product met naam 'Hibernator'. Om niet in herhaling te vallen, kan onderstaand resultaat vergeleken worden met de tabel vanuit de verwijzing hierboven.

```
SELECT PRODUCT_NAME, PRODUCTION_COST FROM Greatoutdoor.PRODUCT
WHERE PRODUCTION_COST BETWEEN 10.0 AND 100.0
AND PRODUCT_NAME = 'Hibernator';
```

Bovenstaande SELECT query is vrij simpel. De SELECT spreekt voor zich. De WHERE-clause zal voor de PRODUCTION_COST de waardes filteren van de kosten tussen de 10 en 100. Ook moet deze voorwaardelijk overeenkomen met het PRODUCT_NAME 'Hibernator'.

Onderstaand het resultaat van deze test. Het resultaat voldoet met de eerder gemaakte functie 1.

PRODUCT_NAME	PRODUCTION_COST
Hibernator	86

Figuur 10 - Resultaat function test query 1

Functie 2

Voor deze test verwijst ik graag naar mijn eerdere uitwerking van een [Function voorbeeld](#). Hierin staat een screenshot van het resultaat na uitvoering van de function in een tabel. In deze test heb ik een steekproef gedaan voor order met nummer '1153'. Om niet in herhaling te vallen, kan onderstaand resultaat vergeleken worden met die vanuit de verwijzing in dit document.

```
SELECT ORDER_NUMBER,
       SUM(QUANTITY * UNIT_COST) AS TotalCost
  FROM Greatoutdoor.ORDER_DETAILS
 WHERE ORDER_NUMBER = 1153
 GROUP BY ORDER_NUMBER;
```

Ook deze SELECT query is vrij rechttoe rechtaan als het gaat om de uitleg hiervan. De SELECT retourneert een tabel met ORDER_NUMBER en een SUM van het aantal items vermenigvuldigd met de UNIT_COST als TotalCost. Deze moet aan de voorwaarde voldoen dat het ORDER_NUMBER 1153 in dit geval. Hierna volgt een GROUP_BY om ze te groeperen. Vanuit deze SELECT zal dit 1 regel opleveren die voldoet. Het resultaat van deze test voldoet aan de werking van de werkelijke functie 2.

<input type="checkbox"/> ORDER_NUMBER ▾	<input type="checkbox"/> TotalCost ▾
1153	15556.32

Figuur 11 - Resultaat function test query 2

Edwin

Testen Stored Procedure

De gerealiseerde stored procedure moet ook getest worden of deze doet wat moet doen, namelijk het ophogen van de waarde in de production_cost kolom a.d.h.v een mee te geven product type en het verhogingspercentage.

Uitgangssituatie

Om alle producten die vallen onder product type 'Cooking Gear' op de halen heb ik de volgende query uitgevoerd op de database.

The screenshot shows the Object Explorer on the left and a SQL Query window on the right. The query retrieves all columns from the 'dbo.PRODUCT' table where the 'PRODUCT_TYPE_CODE' is in a subquery that selects 'PRODUCT_TYPE_CODE' from 'dbo.PRODUCT_TYPE' where 'PRODUCT_TYPE_EN' is 'Cooking Gear'. The results show 10 rows of product data, including columns like PRODUCT_NUMBER, INTRODUCTION_DATE, PRODUCT_TYPE_CODE, PRODUCTION_COST, MARGIN, PRODUCT_IMAGE, and PRODUCT_NAME.

PRODUCT_NUMBER	INTRODUCTION_DATE	PRODUCT_TYPE_CODE	PRODUCTION_COST	MARGIN	PRODUCT_IMAGE	PRODUCT_NAME
1	1995-02-15 00:00:00.0000000	1	4	33	P01CE1CG1.jpg	TrailChef Water Bag
2	1995-02-15 00:00:00.0000000	1	9,22	23	P02CE1CG1.jpg	TrailChef Canteen
3	1995-02-15 00:00:00.0000000	1	15,93	28	P03CE1CG1.jpg	TrailChef Kitchen Kit
4	1995-02-15 00:00:00.0000000	1	5	28	P04CE1CG1.jpg	TrailChef Cup
5	1995-02-15 00:00:00.0000000	1	34,97	3	P05CE1CG1.jpg	TrailChef Cook Set
6	1997-03-05 00:00:00.0000000	1	85,11	28	P06CE1CG1.jpg	TrailChef Deluxe Cook Set
7	1995-02-15 00:00:00.0000000	1	46,38	28	P07CE1CG1.jpg	TrailChef Single Flame
8	1997-03-05 00:00:00.0000000	1	75	41	P08CE1CG1.jpg	TrailChef Double Flame
9	1997-03-05 00:00:00.0000000	1	9	25	P09CE1CG1.jpg	TrailChef Kettle
10	1995-02-15 00:00:00.0000000	1	10	4	P10CE1CG1.jpg	TrailChef Utensils

Afb. Resultaat van de uitgevoerde query met alle producten.

Voor al deze 10 producten willen we een opgehoogde production_cost waarde zien.

Wat we niet willen is dat er onbedoeld een ander product type ook wordt meegenomen in de update. Als we kijken naar product type 'Tents' in onderstaande afbeelding mag deze waarde *niet* aangepast worden.

The screenshot shows the same SQL query as before, but it includes an additional WHERE clause to filter for 'Tents' products. The results show 16 rows. The production_cost for product number 11 is highlighted with a yellow box, showing a value of 250, which is different from the original values in the first screenshot.

PRODUCT_NUMBER	INTRODUCTION_DATE	PRODUCT_TYPE_CODE	PRODUCTION_COST	MARGIN	PRODUCT_IMAGE	PRODUCT_NAME
1	1995-02-15 00:00:00.0000000	1	4	33	P01CE1CG1.jpg	TrailChef Water Bag
2	1995-02-15 00:00:00.0000000	1	9,22	23	P02CE1CG1.jpg	TrailChef Canteen
3	1995-02-15 00:00:00.0000000	1	15,93	28	P03CE1CG1.jpg	TrailChef Kitchen Kit
4	1995-02-15 00:00:00.0000000	1	5	28	P04CE1CG1.jpg	TrailChef Cup
5	1995-02-15 00:00:00.0000000	1	34,97	3	P05CE1CG1.jpg	TrailChef Cook Set
6	1997-03-05 00:00:00.0000000	1	85,11	28	P06CE1CG1.jpg	TrailChef Deluxe Cook Set
7	1995-02-15 00:00:00.0000000	1	46,38	28	P07CE1CG1.jpg	TrailChef Single Flame
8	1997-03-05 00:00:00.0000000	1	75	41	P08CE1CG1.jpg	TrailChef Double Flame
9	1997-03-05 00:00:00.0000000	1	9	25	P09CE1CG1.jpg	TrailChef Kettle
10	1995-02-15 00:00:00.0000000	1	10	4	P10CE1CG1.jpg	TrailChef Utensils
11	1995-02-15 00:00:00.0000000	2	250	28	P11CE1TN2.jpg	Star Lite
12	1997-03-05 00:00:00.0000000	2	473,07	23	P12CE1TN2.jpg	Star Dome
13	1995-02-15 00:00:00.0000000	2	392,57	25	P13CE1TN2.jpg	Star Gazer 2
14	1997-03-05 00:00:00.0000000	2	476	2	P14CE1TN2.jpg	Star Gazer 3
15	1997-03-05 00:00:00.0000000	2	490	33	P15CE1TN2.jpg	Star Gazer 6
16	1997-03-05 00:00:00.0000000	2	1	5	P16CE1TN2.jpg	Star Peg

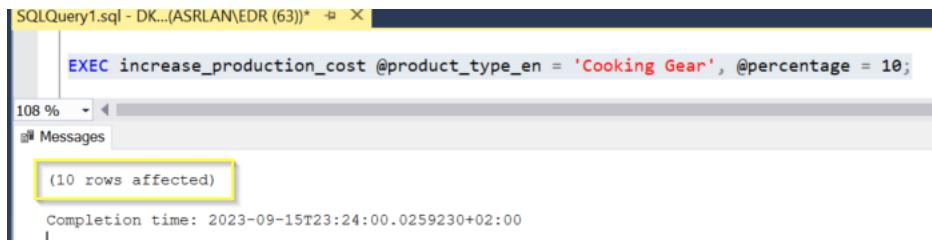
Afb. In geel gemarkeerde prijs moet ongewijzigd blijven.

Uitvoeren test

We roepen de procedure aan met het volgende statement en ik voer deze uit.

```
EXEC increase_production_cost @product_type_en = 'Cooking Gear', @percentage = 10;
```

We zien dat de stored procedure is uitgevoerd met het volgende resultaat.



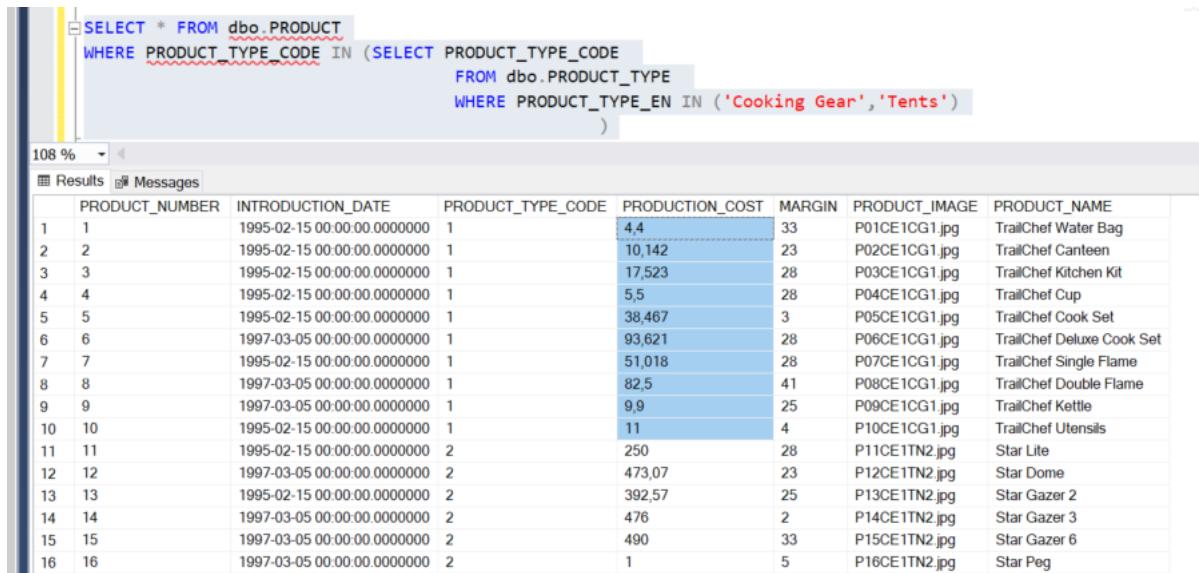
```
SQLQuery1.sql - DK...(ASRLAN\EDR (63))*  X |  
EXEC increase_production_cost @product_type_en = 'Cooking Gear', @percentage = 10;  
108 %  <  Messages  
(10 rows affected)  
Completion time: 2023-09-15T23:24:00.0259230+02:00
```

Afb. Uitgevoerde stored procedure.

Test resultaat

We zien dat er inderdaad 10 rijen zijn geraakt, maar zijn dit ook de juiste rijen van product type 'Cooking Gear'?

Om dit te controleren voer ik opnieuw de opgestelde test query uit.



```
SELECT * FROM dbo.PRODUCT  
WHERE PRODUCT_TYPE_CODE IN (SELECT PRODUCT_TYPE_CODE  
                             FROM dbo.PRODUCT_TYPE  
                             WHERE PRODUCT_TYPE_EN IN ('Cooking Gear', 'Tents'))
```

	PRODUCT_NUMBER	INTRODUCTION_DATE	PRODUCT_TYPE_CODE	PRODUCTION_COST	MARGIN	PRODUCT_IMAGE	PRODUCT_NAME
1	1	1995-02-15 00:00:00.0000000	1	4,4	33	P01CE1CG1.jpg	TrailChef Water Bag
2	2	1995-02-15 00:00:00.0000000	1	10,142	23	P02CE1CG1.jpg	TrailChef Canteen
3	3	1995-02-15 00:00:00.0000000	1	17,523	28	P03CE1CG1.jpg	TrailChef Kitchen Kit
4	4	1995-02-15 00:00:00.0000000	1	5,5	28	P04CE1CG1.jpg	TrailChef Cup
5	5	1995-02-15 00:00:00.0000000	1	38,467	3	P05CE1CG1.jpg	TrailChef Cook Set
6	6	1997-03-05 00:00:00.0000000	1	93,621	28	P06CE1CG1.jpg	TrailChef Deluxe Cook Set
7	7	1995-02-15 00:00:00.0000000	1	51,018	28	P07CE1CG1.jpg	TrailChef Single Flame
8	8	1997-03-05 00:00:00.0000000	1	82,5	41	P08CE1CG1.jpg	TrailChef Double Flame
9	9	1997-03-05 00:00:00.0000000	1	9,9	25	P09CE1CG1.jpg	TrailChef Kettle
10	10	1995-02-15 00:00:00.0000000	1	11	4	P10CE1CG1.jpg	TrailChef Utensils
11	11	1995-02-15 00:00:00.0000000	2	250	28	P11CE1TN2.jpg	Star Lite
12	12	1997-03-05 00:00:00.0000000	2	473,07	23	P12CE1TN2.jpg	Star Dome
13	13	1995-02-15 00:00:00.0000000	2	392,57	25	P13CE1TN2.jpg	Star Gazer 2
14	14	1997-03-05 00:00:00.0000000	2	476	2	P14CE1TN2.jpg	Star Gazer 3
15	15	1997-03-05 00:00:00.0000000	2	490	33	P15CE1TN2.jpg	Star Gazer 6
16	16	1997-03-05 00:00:00.0000000	2	1	5	P16CE1TN2.jpg	Star Peg

Afb. Resultaat na uitvoeren van de stored procedure.

Bovenstaande resultaten laten zien dat de gerealiseerde stored procedure `increase_production_cost` doet wat er verwacht wordt, de production cost kolom is opgehoogd met 10%. Tevens zijn er geen onbedoelde rijen geraakt die niet bijgewerkt hadden mogen worden. De test is akkoord.

Testen Table Function Inventory count

De gemaakte table function test ik door een product nummer mee te geven aan de functie.

Uitgangssituatie

Om de voorraad van alle *TrailChef Water Bag* producten te tonen heb ik de volgende query uitgevoerd op de database.

The screenshot shows a SQL query in the SQL Query window and its results in the Results grid. The query selects inventory year, product number, product name, and average inventory year for product number 1. The results show three rows for the years 2004, 2005, and 2006.

```
SQLQuery1.sql - DK... (ASRLAN\EDR (63))*
select i.INVENTORY_YEAR, i.PRODUCT_NUMBER, p.PRODUCT_NAME
, AVG(i.INVENTORY_COUNT) AS AVG_INVENTORY_YEAR
from dbo.PRODUCT p
join dbo.INVENTORY_LEVELS i on p.PRODUCT_NUMBER = i.PRODUCT_NUMBER
where p.PRODUCT_NUMBER = 1
group by i.INVENTORY_YEAR, i.PRODUCT_NUMBER, p.PRODUCT_NAME
```

INVENTORY_YEAR	PRODUCT_NUMBER	PRODUCT_NAME	AVG_INVENTORY_YEAR
1	2004	TrailChef Water Bag	2222
2	2005	TrailChef Water Bag	4280
3	2006	TrailChef Water Bag	7809

Afb. verwachte uitkomst productnummer 1 TrailChef Water Bag.

Als we géén product nummer meegeven, dus ‘NULL’, moeten alle producten met hun aantallen worden getoond. Ik laat alleen de eerste 18 rijen zien en het totaal aantal records. Het totaal aantal rijen dat ik verwacht is dan 324 zoals rechts onder te zien is in onderstaande afbeelding.

The screenshot shows the same query as above, but it is run against the entire database. The results grid shows 324 rows of data for various products across different years. The message bar at the bottom indicates the query was executed successfully.

```
Object Explorer
Connect ▾ SQLQuery1.sql - DK... (ASRLAN\EDR (63))*
select i.INVENTORY_YEAR, i.PRODUCT_NUMBER, p.PRODUCT_NAME
, AVG(i.INVENTORY_COUNT) AS AVG_INVENTORY_YEAR
from dbo.PRODUCT p
join dbo.INVENTORY_LEVELS i on p.PRODUCT_NUMBER = i.PRODUCT_NUMBER
group by i.INVENTORY_YEAR, i.PRODUCT_NUMBER, p.PRODUCT_NAME
```

INVENTORY_YEAR	PRODUCT_NUMBER	PRODUCT_NAME	AVG_INVENTORY_YEAR
1	2004	TrailChef Water Bag	2222
2	2004	TrailChef Canteen	1125
3	2004	TrailChef Kitchen Kit	2684
4	2004	TrailChef Cup	5399
5	2004	TrailChef Cook Set	1053
6	2004	TrailChef Deluxe Cook Set	1025
7	2004	TrailChef Single Flame	2622
8	2004	TrailChef Double Flame	2179
9	2004	TrailChef Kettle	1344
10	2004	TrailChef Utensils	2369
11	2004	Star Lite	606
12	2004	Star Dome	1009
13	2004	Star Gazer 2	881
14	2004	Star Gazer 3	2471
15	2004	Star Gazer 6	942

Query executed successfully. DFKZ3J (16.0 RTM) | ASRLAN\EDR (63) | GO | 00:00:00 | 324 rows

Afb. verwachte uitkomst alle producten met gemiddelde voorraad per jaar.

Uitvoeren test

We roepen de table functie aan met het volgende statement en ik voer deze als volgt uit.

```
select * from inventory_avg_per_product(1);
```

We zien dat de table function is uitgevoerd met het volgende resultaat en klopt met het verwachte resultaat.

```

Object Explorer
Connect ▾ Databases System Databases Database Snapshots AdventureWorks2022 GO Database Diagrams Tables Views External Resources Synonyms Programmability Stored Procedures System Stored Procedures dbo.increase_production_cost Functions Table-valued Functions dbo.inventory_avg_per_product
SQLQuery1.sql - DK... (ASRLAN\EDR (63))*
Results Messages
INVENTORY_YEAR PRODUCT_NUMBER PRODUCT_NAME AVG_INVENTORY_YEAR
1 2004 1 TrailChef Water Bag 2222
2 2005 1 TrailChef Water Bag 4280
3 2006 1 TrailChef Water Bag 7809

```

Afb. Uitgevoerde aanroep van table function.

De volgende test laat zien dat wanneer er géén productnummer wordt meegegeven aan de procedure dat alle producten worden getoond met hun gemiddelde voorraad per jaar. Ik laat alleen de eerste 18 rijen zien én het totaal aantal records.

```

Object Explorer
Connect ▾ Databases System Databases Database Snapshots AdventureWorks2022 GO Database Diagrams Tables Views External Resources Synonyms Programmability Stored Procedures Functions Table-valued Functions dbo.inventory_avg_per_product Scalar-valued Functions Aggregate Functions System Functions Database Triggers Assemblies Types Rules Defaults Plan Guides Sequences Query Store Service Broker Storage
SQLQuery1.sql - DK... (ASRLAN\EDR (63))*
Results Messages
INVENTORY_YEAR PRODUCT_NUMBER PRODUCT_NAME AVG_INVENTORY_YEAR
1 2004 1 TrailChef Water Bag 2222
2 2004 2 TrailChef Canteen 1125
3 2004 3 TrailChef Kitchen Kit 2684
4 2004 4 TrailChef Cup 5399
5 2004 5 TrailChef Cook Set 1053
6 2004 6 TrailChef Deluxe Cook Set 1025
7 2004 7 TrailChef Single Flame 2622
8 2004 8 TrailChef Double Flame 2179
9 2004 9 TrailChef Kettle 1344
10 2004 10 TrailChef Utensils 2369
11 2004 11 Star Lite 606
12 2004 12 Star Dome 1009
13 2004 13 Star Gazer 2 881
14 2004 14 Star Gazer 3 2471
15 2004 15 Star Gazer 6 942
16 2004 16 Star Peg 21786
17 2004 17 Hibernator Lite 3287
18 2004 18 Hibernator 3186
19 2004 19 Hibernator Extreme 2499
20 2004 20 Hibernator Self - Inflating Mat 2083
21 2004 21 Hibernator Pad 1683
22 2004 22 Hibernator Pillow 728
23 2004 23 Hibernator Camp Cot 482
24 2004 24 Canyon Mule Climber Backpack 1769
25 2004 25 Canyon Mule Weekender Backpack 1751
26 2004 26 Canyon Mule Journey Backpack 1706
27 2004 27 Canyon Mule Extreme Backpack 1263

```

Query executed successfully. DKFZ3J3 (16.0 RTM) ASRLAN\EDR (63) GO 00:00:00 324 rows

Afb. Uitgevoerde aanroep van table function voor alle producten.

Test resultaat

We zien dat de beide uitkomsten gelijk zijn aan de verwachte uitkomst. Alle gemiddelde aantallen van product nummer 1 worden correct getoond. Ook als we niets meegeven aan de functie worden alle producten getoond met hun gemiddelde aantallen voorraad per jaar.

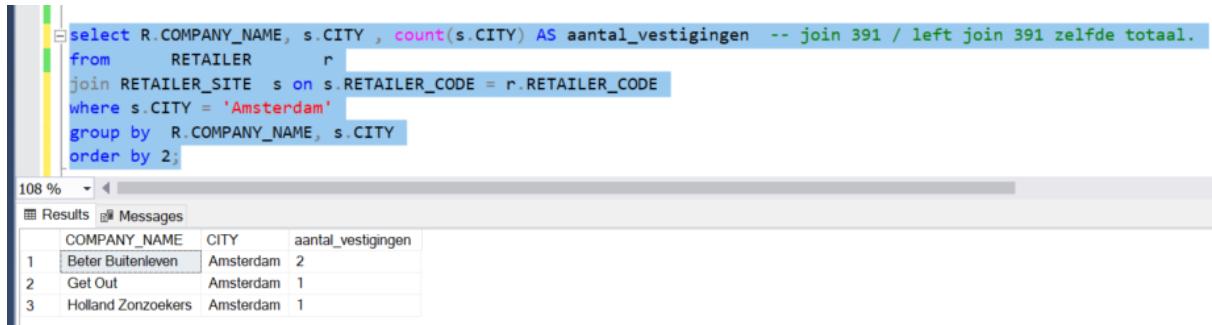
Bovenstaande testen laat zien dat de table function, inventory_avg_per_product doet wat er verwacht wordt, de test is akkoord.

Testen Table Function Retailer City Count Geautomatiseerd

De gemaakte table function test ik door een plaats mee te geven aan de functie. Verder heb ik mijn test geautomatiseerd. De test kan een output valideren aan de hand van een vooraf opgegeven verwachte waarde.

Uitgangssituatie

Om het aantal vestigingen van Amsterdam te tonen heb ik de volgende query uitgevoerd op de database.



The screenshot shows a SQL query in the query editor and its execution results. The query is:

```
select R.COMPANY_NAME, s.CITY , count(s.CITY) AS aantal_vestigingen -- join 391 / left join 391 zelfde totaal.
from RETAILER r
join RETAILER_SITE s on s.RETAILER_CODE = r.RETAILER_CODE
where s.CITY = 'Amsterdam'
group by R.COMPANY_NAME, s.CITY
order by 2;
```

The results pane displays the following table:

	COMPANY_NAME	CITY	aantal_vestigingen
1	Beter Buitenleven	Amsterdam	2
2	Get Out	Amsterdam	1
3	Holland Zonzoekers	Amsterdam	1

Afb. verwachte uitkomst City 'Amsterdam'.

We zien dus het totaal voor de vestiging Amsterdam is 4 vestigingen.

Uitvoeren geautomatiseerde test

We roepen de table functie aan met het volgende statement en ik voer deze als volgt uit.

```

SQLQuery1.sql - DK...(ASRLAN\EDR (66))  X SQL_AC_Database.s...(ASRLAN\EDR (62))
/* testen functie*/

DECLARE @TestCity1 CHAR(50) = 'Amsterdam';

DECLARE @ResultTable TABLE (
    COMPANY_NAME CHAR(50),
    CITY CHAR(50),
    aantal_vestigingen INT
);

INSERT INTO @ResultTable
SELECT * FROM dbo.GetRetailerCityCount(@TestCity1);

DECLARE @ExpectedCity1 CHAR(50) = 'Amsterdam';
DECLARE @ExpectedCount1 INT = 4;

/* testen resultaat */
IF EXISTS (
    SELECT 1
    FROM (
        SELECT MAX(COMPANY_NAME) AS COMPANYNAME, CITY, SUM(aantal_vestigingen) AS TotalCount
        FROM @ResultTable
        GROUP BY CITY
    ) AS Subquery
    WHERE CITY = @ExpectedCity1
        AND TotalCount = @ExpectedCount1
)
BEGIN
    PRINT 'Test Case 1 Passed ' + @ExpectedCity1 + CAST(@ExpectedCount1 AS VARCHAR);
END
ELSE
BEGIN
    PRINT 'Test Case 1 Failed ' + @ExpectedCity1 + CAST(@ExpectedCount1 AS VARCHAR);
END;

```

108 %

Messages

(3 rows affected)
Test Case 1 Passed Amsterdam
Completion time: 2023-10-01T10:25:32.4691797+02:00

Afb. verwachte uitkomst City 'Amsterdam' met 4 vestigen.

Test resultaat

De test is geslaagd het de verwachte opgegeven waarde komt overeen met de uitkomst van de test. We zien in bovenstaande afbeelding dat CASE 1 is passed.

We doen nog een test om te zien wat er gebeurd bij invoer van een fout aantal vestigen. We geven nu 2 op terwijl we 4 verwachten.

```
/* testen functie*/

DECLARE @TestCity1 CHAR(50) = 'Amsterdam';

DECLARE @ResultTable TABLE (
    COMPANY_NAME CHAR(50),
    CITY CHAR(50),
    aantal_vestigingen INT
);

INSERT INTO @ResultTable
SELECT * FROM dbo.GetRetailerCityCount(@TestCity1);

DECLARE @ExpectedCity1 CHAR(50) = 'Amsterdam';
DECLARE @ExpectedCount1 INT = 2;

/* testen resultaat */
IF EXISTS (
    SELECT 1
    FROM (
        SELECT MAX(COMPANY_NAME) AS COMPANYNAME, CITY, SUM(aantal_vestigingen) AS TotalCount
        FROM @ResultTable
        GROUP BY CITY
    ) AS Subquery
    WHERE CITY = @ExpectedCity1
        AND TotalCount = @ExpectedCount1
)
BEGIN
    PRINT 'Test Case 1 Passed ' + @ExpectedCity1 + CAST(@ExpectedCount1 AS VARCHAR);
END
ELSE
BEGIN
    PRINT 'Test Case 1 Failed ' + @ExpectedCity1 + CAST(@ExpectedCount1 AS VARCHAR);
END;

108 % < Messages >
(3 rows affected)
Test Case 1 Failed Amsterdam 2
Completion time: 2023-10-01T10:41:14.5206369+02:00
```

Afb. verwachte uitkomst City 'Amsterdam' met 2 vestigen.

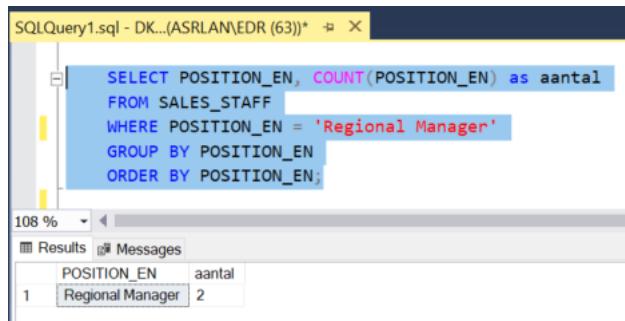
De test is geslaagd het de verwachte opgegeven waarde komt NIET overeen met de uitkomst van de test. We zien in bovenstaande afbeelding dat CASE 1 is Failed omdat 'Amsterdam' 4 vestigen heeft en geen 2.

Testen Scalar Function

De gemaakte scalar function test ik door een functienaam mee te geven aan de functie.

Uitgangssituatie

Om het aantal personen met de functienaam *Regional Manager* op te halen heb ik de volgende query uitgevoerd op de database.



The screenshot shows a SQL query in the query editor and its results in the results pane. The query counts the number of staff members in the SALES_STAFF table who have the position 'Regional Manager'. The results show one row with the position 'Regional Manager' and a count of 2.

```
SQLQuery1.sql - DK...(ASRLAN\EDR (63))*
SELECT POSITION_EN, COUNT(POSITION_EN) as aantal
FROM SALES_STAFF
WHERE POSITION_EN = 'Regional Manager'
GROUP BY POSITION_EN
ORDER BY POSITION_EN;
```

POSITION_EN	aantal	
1	Regional Manager	2

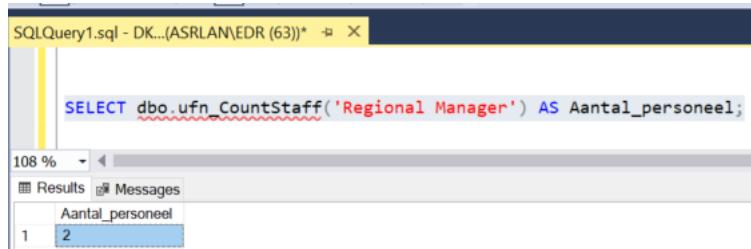
Afb. verwachte uitkomst, 2 regional managers.

Uitvoeren test

We roepen de Scalar Function aan met het volgende statement en ik voer deze uit.

```
SELECT dbo.ufn_CountStaff('Regional Manager') AS Aantal_personeel;
```

We zien dat de scalar function is uitgevoerd met het volgende resultaat.



The screenshot shows a query in the query editor that calls the scalar function dbo.ufn_CountStaff with the argument 'Regional Manager'. The results pane shows the output of the function, which is the value 2.

```
SQLQuery1.sql - DK...(ASRLAN\EDR (63))*
SELECT dbo.ufn_CountStaff('Regional Manager') AS Aantal_personeel;
```

Aantal_personeel	
1	2

Afb. Uitgevoerde scalar function.

Test resultaat

We zien dat de uitkomst van de scalar functie gelijk is aan de verwachte uitkomst. Bovenstaande test laat zien dat de scalar function, ufn_CountStaff doet wat er verwacht wordt, de test is akkoord.

Lesley-Ann

Het testen van een database is belangrijk om de integriteit van de database te bewaren. Het testen van queries, stored procedures en functies zorgt ervoor dat er na gegaan wordt of deze correct werken. Dit geeft een extra check om te zorgen dat gegevens juist en betrouwbaar zijn. Fouten kunnen leiden tot verlies van gegevens, onjuiste data, onjuiste antwoorden op query's etc. Dit alles kan negatieve gevolgen hebben op vele vlakken. Denk bijvoorbeeld aan verkeerde jaarcijfers die uit een database geprint worden of security gevoeligheden.

Ook kan een slecht geschreven database of slecht geschreven query's, procedures en functions het onderhouden van een database moeilijk maken. Tijdens onderhoud en upgrades moeten functies en stored procedures mogelijk worden aangepast. Testen helpen eraan mee dat wijzigingen geen

onverwachtse gevolgen hebben voor de functionaliteit van de database. Ook vermindert het de kans op regressiefouten bij het upgraden van het systeem. (SDET, 2021)

Tests kunnen ook fungeren als een vorm van documentatie voor de databasecode, dit kan nuttig zijn voor andere die ook aan de database werken.

Testten zorgen ervoor dat je op tijdproblemen kan opsporen en oplossen. Dit voorkomt fouten op een later moment. Het testen van stored procedures en functies is een belangrijk deel van het onderhouden van een kwalitatief gezonde database. (Factor, SQL Server User-Defined Functions, 2017)

Voor de query's die ik geschreven heb, heb ik ook tests geschreven en uitgevoerd.

Stored Procedure test

Uitgangssituatie:

Voor deze test verwijst ik naar mijn eerdere uitwerking in mijn persoonlijke stuk in het stored procedure hoofdstuk. Hieronder nogmaals de desbetreffende stored procedure:

```
USE GODATABASE
GO
CREATE PROCEDURE GetTotalEmployeesPerLocation
AS
BEGIN
    SELECT SB.SALES_BRANCH_CODE, SB.CITY, COUNT(SS.SALES_STAFF_CODE) AS TotalEmployees
    FROM dbo.SALES_BRANCH SB
    LEFT JOIN dbo.SALES_STAFF SS ON SB.SALES_BRANCH_CODE = SS.SALES_BRANCH_CODE
    GROUP BY SB.SALES_BRANCH_CODE, SB.CITY
    ORDER BY SB.SALES_BRANCH_CODE;
END
EXEC GetTotalEmployeesPerLocation;
```

Results

	SALES_BRANCH_CODE	CITY	TotalEmployees
1	6	Paris	4
2	7	Milano	3
3	9	Amsterdam	5
4	13	Hamburg	4
5	14	München	3
6	15	Kista	4
7	17	Calgary	5
8	18	Toronto	5
9	19	Boston	6

Ik heb een test geschreven die controleert of het aantal rijen in de tijdelijke tabel overeenkomt met het verwachte aantal tabellen. Dit doet de test door de resultaten in een test tabel op te slaan en deze te vergelijken met de verwachte tabellen die de user aangeeft. In het eerste voorbeeld is de test geslaagd en word 'correct' geprint, het aantal tabellen komen inderdaad overeen met de verwachte

antwoorden. In voorbeeld 2 voer ik expres de foute hoeveelheid tabellen in om te checken dat de test dan ook een error geeft. Dit gebeurt en de test is daarom geslaagd.

The screenshot shows a SQL script window with the following code:

```
CREATE TABLE #TestResults (
    SALES_BRANCH_CODE INT,
    CITY NVARCHAR(40),
    TotalEmployees INT
);

INSERT INTO #TestResults
EXEC GetTotalEmployeesPerLocation;

IF (SELECT COUNT(*) FROM #TestResults) = 28
BEGIN
    PRINT 'Correct aantal rijen';
END
ELSE
BEGIN
    PRINT 'Failed: Onjuist aantal rijen.';
END
```

The status bar at the bottom left shows ".07 %". Below the window, the "Messages" pane displays the output: "Correct aantal rijen". The completion time is shown as "Completion time: 2023-09-17T21:36:46.3653144+02:00".

Figuur 12 Voorbeeld 1

The screenshot shows a SQL script window with the same code as in Figure 12, but with a different result. The status bar at the bottom left shows "107 %". Below the window, the "Messages" pane displays the output: "Failed: Onjuist aantal rijen.".

Figuur 13 Voorbeeld 2

Function tests

Functie 1

Uitgangssituatie:

Voor deze test verwijst ik naar mijn eerdere uitwerking in mijn persoonlijke stuk in het functions hoofdstuk. Hieronder nogmaals de desbetreffende function:

The screenshot shows the SQL code for creating a function and its execution. The function `dbo.GetCountryName3` takes an integer parameter `@CountryCode` and returns a country name as an nvarchar(50). It uses a cursor to select the country name from the `COUNTRY` table where the `COUNTRY_CODE` matches the input parameter. The function is then tested by declaring a variable `@CountryCode` to 1, declaring a variable `@CountryName` as nvarchar(50), executing the function, and printing the result.

```
CREATE FUNCTION dbo.GetCountryName3 (@CountryCode int)
RETURNS nvarchar(50)
AS
BEGIN
    DECLARE @CountryName nvarchar(50)

    SELECT @CountryName = COUNTRY
    FROM COUNTRY
    WHERE COUNTRY_CODE = @CountryCode

    RETURN @CountryName
END

DECLARE @CountryCode int = 1;
DECLARE @CountryName nvarchar(50);

SELECT @CountryName = dbo.GetCountryName3(@CountryCode);
PRINT 'Country Name: ' + @CountryName;
```

07 % ▾

Messages

Country Name: France

Completion time: 2023-09-17T12:00:27.8772592+02:00

Een handmatige check voor het verifiëren of de query de juiste informatie geeft is het nakijken in de database. Dit heb ik gedaan met een aantal voorbeelden en dit klopte.

The screenshot shows the results of a query on the `COUNTRY` table. The table has columns `COUNTRY_CODE`, `COUNTRY`, and `LANGUAGE`. The data shows 14 rows of country codes and names, all in English (EN).

	COUNTRY_CODE	COUNTRY	LANGUAGE
1	1	France	EN
2	2	Germany	EN
3	3	United States	EN
4	4	Canada	EN
5	5	Austria	EN
6	6	Italy	EN
7	7	Netherlands	EN
8	8	Switzerland	EN
9	9	United Kingdom	EN
10	10	Sweden	EN
11	11	Japan	EN
12	12	Taiwan	EN
13	13	Korea	EN
14	14	China	EN

Wil je de test automatiseren dan is het schrijven van een test een optie. Handmatig nakijken kan in een kleine database maar dit is niet realistisch in een grotere database. Deze test heb ik gedaan door de verwachte informatie in te voeren en deze te vergelijken met de informatie die de query geeft.

In voorbeeld 1 heb ik de juiste informatie aangeleverd uit de database, deze komt overeen met de informatie uit de query. De uitkomst is 'correct'. In voorbeeld 2 verander ik de verwachte informatie naar een fout, ik verander het land naar een land die niet overeenkomt met code 3 in de database. De uitkomst is zoals verwacht 'failed'.

```
DECLARE @CountryCode int = 3;
DECLARE @ExpectedCountryName nvarchar(50) = 'United States';

DECLARE @ActualCountryName nvarchar(50);
SELECT @ActualCountryName = dbo.GetCountryName3(@CountryCode);

IF @ActualCountryName = @ExpectedCountryName
    PRINT 'Correct';
ELSE
    PRINT 'Failed';
```

The screenshot shows a SQL query window with the following content. The code declares variables for a country code (3) and an expected country name ('United States'). It then selects the actual country name from a function (dbo.GetCountryName3) based on the country code. Finally, it compares the actual and expected country names. If they are equal, it prints 'Correct'; otherwise, it prints 'Failed'. The status bar at the bottom left indicates '07 %'.

07 %

Messages

Correct

Figuur 14 Voorbeeld 1

```
DECLARE @CountryCode int = 3;
DECLARE @ExpectedCountryName nvarchar(50) = 'Peru';

DECLARE @ActualCountryName nvarchar(50);
SELECT @ActualCountryName = dbo.GetCountryName3(@CountryCode);

IF @ActualCountryName = @ExpectedCountryName
    PRINT 'Correct';
ELSE
    PRINT 'Failed';
```

The screenshot shows a SQL query window with the same code as Figure 14, but with a different expected country name ('Peru'). The comparison fails because the actual country name (United States) does not match the expected country name (Peru). The status bar at the bottom left indicates '107 %'.

107 %

Messages

Failed

Figuur 15 Voorbeeld 2

Functie 2

Uitgangssituatie:

Voor deze test verwijst ik naar mijn eerdere uitwerking in mijn persoonlijke stuk in het functions hoofdstuk. Hieronder nogmaals de desbetreffende function:

```

CREATE FUNCTION GetLowInventory()
RETURNS TABLE
AS
RETURN
(
    SELECT
        P.PRODUCT_NUMBER,
        P.PRODUCT_NAME,
        IL.INVENTORY_COUNT
    FROM
        dbo.PRODUCT P
    LEFT JOIN
        dbo.[INVENTORY_LEVELS] IL ON P.PRODUCT_NUMBER = IL.PRODUCT_NUMBER
    WHERE
        IL.INVENTORY_COUNT < 10
);
SELECT * FROM dbo.GetLowInventory();

```

107 %

	PRODUCT_NUMBER	PRODUCT_NAME	INVENTORY_COUNT
1	107	Lady Hailstorm Titanium Woods Set	-31
2	65	Mountain Man Deluxe	-166
3	84	Glacier GPS	-183
4	84	Glacier GPS	-57
5	84	Glacier GPS	-297
6	78	Seeker 35	-345

Een mogelijke eerste check die gedaan kan worden om te kijken of de informatie die geprint wordt door de functie overeen komt met de informatie in de database is om de database met de antwoorden te vergelijken:

```

SELECT
    PRODUCT_NUMBER,
    INVENTORY_YEAR,
    INVENTORY_MONTH,
    INVENTORY_COUNT
FROM
    dbo.INVENTORY_LEVELS
ORDER BY
    INVENTORY_COUNT ASC;

```

107 % ▶

	PRODUCT_NUMBER	INVENTORY_YEAR	INVENTORY_MONTH	INVENTORY_COUNT
1	78	2006	10	-345
2	84	2005	6	-297
3	84	2005	4	-183
4	65	2004	10	-166
5	84	2005	5	-57
6	107	2004	3	-31
7	107	2004	2	31
8	107	2004	1	31

Figuur 16 Eerste check

De informatie lijkt juist te zijn maar dit is geen methode die je kan aanhouden om informatie te verifiëren. Dit zou bijvoorbeeld onbegonnen werk zijn in een grote database. Voor een betere check heb ik een test geschreven.

Eerst word de tijdelijke variabele @Result aangemaakt om de resultaten van de functie op te nemen. De functie word uitgevoerd en de resultaten hiervan worden opgenomen in @Result. Dan word er in de test gekeken of er een rij is in @Result waarbij de INVENTORY_COUNT gelijk is of groter dan 10. Als dit zo is word ‘Failed’ geprint. Als die niet zo is word ‘Correct’ geprint.

Zoals te zien is in voorbeeld 1 word ‘correct’ geprint en is de test geslaagd. In voorbeeld 2 pas ik de test aan, ik verander de parameter en geeft aan dat de test moet failen als er een INVENTORY_COUNT is onder de 10. Ik verwacht dat hij failed omdat de verwachte INVENTORY_COUNTs allemaal onder de 10 liggen. De test print zoals verwacht ‘failed’.

```

DECLARE @Result TABLE (
    PRODUCT_NUMBER INT,
    PRODUCT_NAME NVARCHAR(255),
    INVENTORY_COUNT INT
);

INSERT INTO @Result
SELECT * FROM dbo.GetLowInventory();

IF EXISTS (SELECT 1 FROM @Result WHERE INVENTORY_COUNT >= 10)
BEGIN
    PRINT 'Failed, contains items with count of 10 or more.';
END
ELSE
BEGIN
    PRINT 'Correct, contains only items with count < 10.';
END

```

107 %

Messages

```

(6 rows affected)
Correct, contains only items with count < 10.

```

Figuur 17 Test ronde 1

```

DECLARE @Result TABLE (
    PRODUCT_NUMBER INT,
    PRODUCT_NAME NVARCHAR(255),
    INVENTORY_COUNT INT
);

INSERT INTO @Result
SELECT * FROM dbo.GetLowInventory();

IF EXISTS (SELECT 1 FROM @Result WHERE INVENTORY_COUNT <= 10)
BEGIN
    PRINT 'Failed';
END
ELSE
BEGIN
    PRINT 'Correct';
END

```

107 %

Messages

```

(6 rows affected)
Failed

```

Figuur 18 Test ronde 2

Sander

Om te kunnen garanderen dat mijn voorgestelde functions en stored procedures de juiste resultaten geven en/of modificaties doen heb ik enkele tests bedacht en uitgevoerd. Wanneer SQL Server gebruikt wordt in combinatie met Microsoft Visual Studio, kan een ingebouwde unit test-oplossing gebruikt worden (Microsoft, 2023). Echter is het gebruik van Microsoft Visual Studio hiervoor wat mij

betreft out of scope. Dit omdat het een additionele grote installatie vereist, waardoor de testresultaten door anderen lastiger te verifiëren zouden zijn.

Daarom beperk ik mij tot de mogelijkheden die te gebruiken zijn vanaf SQL Server zelf in combinatie met een tool zoals SQL Server Management Studio en JetBrains DataGrip. De tests zullen dan ook enkel gebruik maken van nieuwe stored procedures in combinatie met (indien de test gebruik maakt van DML zoals INSERT, UPDATE, DELETE) transacties. Dankzij transacties kunnen we het resultaat van DML-statements controleren zonder de wijzigingen daadwerkelijk op te slaan (Microsoft, 2023).

Stored procedure

Voor het hoofdstuk over stored procedures en functions heb ik de stored procedure (hierna: SP) go_delete_order geschreven. Deze SP verwijdert de gegeven ORDER_HEADER record samen met de ORDER_DETAILS records die bij de ORDER_HEADER record horen. Wanneer dit verwijderen gelukt is, zou de SP de waarde 0 moeten returnen. Wanneer het niet gelukt is, zou de SP de waarde 1 moeten returnen. Daarnaast zal de test controleren of het verwachte aantal record verwijderd is.

```

CREATE OR ALTER PROCEDURE go_delete_order_test_with_known_existing_order
AS
DECLARE @ret_code int, @fail_reason varchar(255)
DECLARE @before_order_header_rowcount int, @after_order_header_rowcount int
DECLARE @before_order_details_rowcount int, @after_order_details_rowcount int
DECLARE @order_number int = 1235

-- Make sure database won't be altered using a transaction
BEGIN TRANSACTION

-- Determine the count of rows before
SET @before_order_header_rowcount = (SELECT COUNT(*) FROM ORDER_HEADER)
SET @before_order_details_rowcount = (SELECT COUNT(*) FROM ORDER_DETAILS)

-- Determine whether order actually exists and has details
IF (SELECT COUNT(*) FROM ORDER_HEADER WHERE ORDER_NUMBER = @order_number) <> 1
    BEGIN
        -- ORDER_HEADER probably not existing
        SET @fail_reason = 'Test failed: ORDER_HEADER record not existing'
        GOTO THROW_ERROR
    END

IF (SELECT COUNT(*) FROM ORDER_HEADER WHERE ORDER_NUMBER = @order_number) = 0
    BEGIN
        -- ORDER_DETAILS probably not existing
        SET @fail_reason = 'Test failed: ORDER_DETAILS records not existing'
        GOTO THROW_ERROR
    END

-- Now actually run order deletion routine
EXECUTE @ret_code = go_delete_order @ORDER_NUMBER = @order_number
IF @ret_code <> 0
    BEGIN
        -- go_delete_order failed
        SET @fail_reason = 'Test failed: go_delete_order returned 1'
        GOTO THROW_ERROR
    END

-- Determine the count of rows after deletion
SET @after_order_header_rowcount = (SELECT COUNT(*) FROM ORDER_HEADER)
SET @after_order_details_rowcount = (SELECT COUNT(*) FROM ORDER_DETAILS)

-- Test ORDER_HEADER row count
IF @after_order_header_rowcount <> (@before_order_header_rowcount - 1)
    BEGIN
        SET @fail_reason = 'Test failed: unexpected ORDER_HEADER row count'
        GOTO THROW_ERROR
    END

-- Test ORDER_DETAILS row count
IF @after_order_details_rowcount <> (@before_order_details_rowcount - 3)
    BEGIN
        SET @fail_reason = 'Test failed: unexpected ORDER_DETAILS row count'
        GOTO THROW_ERROR
    END

-- Determine whether correct order has been deleted
IF (SELECT COUNT(*) FROM ORDER_HEADER WHERE ORDER_NUMBER = @order_number) <> 0
    BEGIN

```

```

        SET @fail_reason = 'Test failed: ORDER_HEADER record still exists'
        GOTO THROW_ERROR
    END

    IF (SELECT COUNT(*) FROM ORDER_HEADER WHERE ORDER_NUMBER = @order_number) <> 0
    BEGIN
        SET @fail_reason = 'Test failed: ORDER_DETAILS records still exist'
        GOTO THROW_ERROR
    END

    -- Make sure changes are not saved
    ROLLBACK
    -- All good!
    RETURN (0)

    THROW_ERROR:
    THROW 50000, @fail_reason, 1
GO

```

In deze test is gekozen om te testen met de order met ORDER_NUMBER 1235. Dit kan op regel 6 aangepast worden. Voor deze ORDER_HEADER bestaan er 3 ORDER_DETAILS in de verkregen data.

De test:

- Bepaalt het aantal regels in de ORDER_HEADER en ORDER_DETAILS tabellen vooraf.
- Controleert of de ORDER_HEADER echt bestaat en er ook ORDER_DETAILS voor de order bestaan.
- Voert de go_delete_order SP uit en controleert de RETURN waarde hiervan.
- Bepaalt en controleert het aantal regels in de ORDER_HEADER en ORDER_DETAILS opnieuw (deze zouden verminderd moeten zijn met respectievelijk 1 en 3).
- Controleert of de ORDER_HEADER en ORDER_DETAILS records inderdaad niet meer bestaan.

Door de laatste 2 onderdelen van de test is het logisch gezien onmogelijk dat de verkeerde regels gewist zijn.

De test kan als volgt uitgevoerd worden:

```
EXEC go_delete_order_test_with_known_existing_order
```

Functie 1 - Total order value

De volgende test bepaalt of de function go_total_order_value de juiste waarde geeft:

```

CREATE OR ALTER PROCEDURE go_total_order_value_test
AS
DECLARE @order_number int = 1235
DECLARE @expected_value float = 38907.18

    -- Determine order value
DECLARE
    @actual_value float = (SELECT TOTAL_VALUE FROM
go_total_order_value(@order_number))

    -- Test if order exists
IF (SELECT COUNT(*) FROM ORDER_DETAILS WHERE ORDER_NUMBER = @order_number) = 0
    BEGIN
        THROW 50000, 'Test failed: Order has no details', 1
    END

    -- Test order value
IF @actual_value <> @expected_value
    BEGIN
        THROW 50000, 'Test failed: Unexpected order value', 1
    END

    -- All good!
RETURN (0)
GO

```

Ik test met order 1235. Hiervan heb ik reeds gecontroleerd dat de waarde 38907,18 is. Dit kan aangepast worden op respectievelijk regel 3 en 4. De test controleert eerst of de order wel details heeft. Daarna wordt de waarde opgevraagd via de functie die we testen en controleren we vervolgens of dit klopt met de verwachte waarde.

Indien de order geen regels heeft (of niet bestaat) zal dit gemeld worden.

De test kan als volgt uitgevoerd worden:

```
EXEC go_total_order_value_test
```

Functie 2 - Lijst van managers voor medewerker

De tweede functie die ik gemaakt had, geeft de managementstructuur van een gegeven medewerker. Voor de medewerker met code 4 heeft dies manager code 18. De medewerker met 18 heeft vervolgens een manager met code 13. De medewerker met code 13 heeft vervolgens weer een manager met code 108. De medewerker met code 108 heeft geen manager. In de verkregen data is dat te zien als dat de medewerker dies eigen manager is (oftewel: manager code gelijk aan de staff code).

We kunnen dit als volgt testen:

```

CREATE OR ALTER PROCEDURE employee_management_tree_test
AS
DECLARE @sales_staff_code int = 4

    -- Create temp table containing the expected management structure
CREATE TABLE #ExpectedManagementStructure (MANAGER_CODE int)
INSERT INTO #ExpectedManagementStructure VALUES (18)
INSERT INTO #ExpectedManagementStructure VALUES (13)
INSERT INTO #ExpectedManagementStructure VALUES (108)

    -- Determine management structure
CREATE TABLE #ActualManagementStructure (MANAGER_CODE int)
INSERT INTO #ActualManagementStructure SELECT * FROM
EMPLOYEE_MANAGEMENT_TREE(@sales_staff_code)

    -- Determine checksums for both tables
DECLARE @ChecksumExpected INT, @ChecksumActual INT;
SELECT @ChecksumExpected = BINARY_CHECKSUM(*) FROM
#ExpectedManagementStructure;
SELECT @ChecksumActual = BINARY_CHECKSUM(*) FROM #ActualManagementStructure;

    -- Test if tables have same contents
IF @ChecksumExpected <> @ChecksumActual
    BEGIN
        THROW 50000, 'Test failed: Different result', 1
    END

    -- All good!
RETURN (0)
GO

```

De test maakt eerst een tijdelijke tabel aan met het verwachte resultaat erin. Vervolgens wordt de EMPLOYEE_MANAGEMENT_TREE function aangeroepen, hiervan wordt het resultaat bewaard. Vervolgens vergelijken we de tabellen middels hun checksum.

Deze test kan als volgt uitgevoerd worden:

```
EXEC employee_management_tree_test
```

Uitbreiding objectsoort

Thierry

Vanuit de opdracht is ons gevraagd de behoefte van een nieuw objectsoort te bedenken. Hierbij waren een tabel of SP, wat ook objectsoorten zijn, niet meer toegestaan. Om dit vraagstuk verder uit te werken ben ik allereerst op zoek gegaan naar welke objecten MSSQL biedt. Ik kwam na wat speurwerk uit op de site van Careerride.com (Careerride.com, n.d.).

Er leven heel wat objectsoorten binnen een database. Omdat de SP en Table niet binnen de scope liggen, heb ik mijn keuze gemaakt uit de volgende opties: INDEX, TRIGGER, of VIEW. Deze keuze was achteraf niet geheel lastig, omdat ik de VIEW enigszins bekend mee ben en met de INDEX en TRIGGER (nog) niet.

VIEW zijn virtuele tabellen dat werkelijke tabellen vanuit de database kan combineren en opslaan (Choudhury, n.d.). Het nut van een VIEW is dat het beveiliging voor de database biedt. Het kan complexe query's versimpelen. Dit omdat de complexe query maar één keer hoeft te worden bedacht en opgeslagen wordt. De view kan door iedereen hergebruikt worden zonder opnieuw de complexiteit in te duiken. Ook dragen views bij aan query performance. De database engine optimaliseert de onderliggende query voordat ze worden opgeslagen. Een opgeslagen query draait sneller dan een direct uitgevoerde query op bijvoorbeeld de console.

Voor een nieuw objectsoort gebruik ik een VIEW. Een behoefte hiervoor binnen het bedrijf kan zijn het inzichtelijk maken welk product het meest wordt geretourneerd. Ik heb hier een VIEW voor geschreven die dit inzichtelijk kan maken.

```
CREATE VIEW MostReturnedProducts AS
    SELECT od.PRODUCT_NUMBER,
           SUM(ri.RETURN_QUANTITY) AS total_returned_quantity
    FROM Greatoutdoor.ORDER_DETAILS od
        JOIN Greatoutdoor.RETURNED_ITEM ri ON od.ORDER_DETAIL_CODE = ri.ORDER_DETAIL_CODE
    GROUP BY od.PRODUCT_NUMBER

    SELECT *
    FROM MostReturnedProducts
    ORDER BY total_returned_quantity DESC
```

Deze view werkt als volgt. CREATE VIEW begint met het definiëren van een nieuwe view. Een virtuele tabel dus gebaseerd op het resultaat van een query. Hierna volgt de SELECT statement. Deze selecteert uit de tabellen ORDER_DETAILS en RETURNED_ITEM het PRODUCT_NUMBER en RETURN_QUANTITY. De SUM zorgt voor een berekening van de totale hoeveelheid geretourneerde producten. De JOIN-operatie combineert op zijn beurt weer de gegevens uit de RETURNED_ITEM en ORDER_DETAILS tabellen. Dit op basis van de ORDER_DETAIL_CODE. Als laatst wordt de GROUP BY clausule gebruikt om te groeperen op basis van PRODUCT_NUMBER.

Het resultaat van de VIEW is als onderstaand.

<input type="checkbox"/> PRODUCT_NUMBER	<input type="checkbox"/> total_returned_quantity
16	596
40	578
33	424
1	378
4	372
55	366
29	294
30	274
77	266
12	246
74	244
36	224
17	222
35	200
89	194
32	190

Figuur 19 - Resultaat van de view

Zoals hierboven is te zien, wordt product met PRODUCT_CODE 16 het meest geretourneerd. Voor de volledigheid heb ik een SELECT uitgevoerd om ook de naam bij de code te tonen. Zie hieronder voor het resultaat.

```
SELECT PRODUCT_NUMBER, PRODUCT_NAME FROM Greatoutdoor.PRODUCT
WHERE PRODUCT_NUMBER = '16'
```

<input type="checkbox"/> PRODUCT_NUMBER	<input type="checkbox"/> PRODUCT_NAME
16	Star Peg

Figuur 20 - Test op de view

(Aanvulling herkansing - Thierry)

Als extra verduidelijking heb ik nog een view uitgewerkt met de volgende behoeftestelling voor het bedrijf Great Outdoor. Het bedrijf wil een overzicht kunnen krijgen van het aantal producten dat per seizoen is verkocht aan de hand van het opgegeven seizoen. Zo kan men zien of er seisoensgebonden producten meer worden gekocht in een bepaald seizoen. Het bedrijf zou in deze maanden extra reclame kunnen maken en klantgerichter kunnen adverteren om nog meer omzet te kunnen maken op het specifieke product. Het bedrijf kan zo ook inspelen op de vraag door extra voorraad in te slaan, of een marktonderzoek uitvoeren of het mogelijk is de prijs te verhogen voor extra omzet. Maar het zou uit het onderzoek ook kunnen blijken dat er juist kortingen gegeven kan worden om hierdoor meer van het product om te kunnen zetten, en dus meer inkomsten te kunnen vergaren.

De view die hier inzicht in kan geven heb ik hieronder uitgewerkt. Allereerst zorg ik ervoor dat de ORDER_DATE uit de ORDER_HEADER tabel wordt geconverteerd naar het type DATE. Dit omdat de initiële datetime type te veel van het goede is. De tijd heb ik niet nodig in deze view. Vervolgens haal ik de jaarcomponent uit de date omdat ik een kolom YEAR wil hebben. Ik doe hetzelfde in de CASE

statement met MONTH. De CASE zorgt ervoor dat ik per kwartaal op seizoen kan groeperen. Om te zorgen dat ik over de informatie van verkochte producten kan beschikken voeg ik de tabellen ORDER_HEADER en ORDER_DETAILS tabellen samen. Dit allereerst om de datum aan de hand van het ORDER_NUMMER te matchen. Ook koppel ik het ORDER_DETAILS tabel en PRODUCT tabel aan elkaar. Beide beschikken ze over de PRODUCT_NAME attribuut. Aan de hand van deze joins kan ik per order het aantal producten in deze order optellen en koppelen aan de datum van de order.

Als laatste stap groepeer ik de gegevens op basis van het geconverteerde datumveld, het jaar, het seizoen, het productnummer en de productnaam. Voor elke groep wordt de totale hoeveelheid verkochte producten berekend met de SUM-functie in een TOTAL_QUANTITY_SOLD-kolom.

```

CREATE VIEW ProductSalesPerMonth AS
SELECT
    DATEPART(YEAR, OH.ORDER_DATE) AS [YEAR],
    CASE
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (9, 10, 11) THEN 'Herfst'
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (12, 1, 2) THEN 'Winter'
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (3, 4, 5) THEN 'Lente'
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (6, 7, 8) THEN 'Zomer'
    END AS SEASON,
    P.PRODUCT_NUMBER,
    P.PRODUCT_NAME,
    SUM(OD.QUANTITY) AS TOTAL_QUANTITY SOLD
FROM Greatoutdoor.ORDER_HEADER AS OH
JOIN Greatoutdoor.ORDER_DETAILS AS OD ON OH.ORDER_NUMBER = OD.ORDER_NUMBER
JOIN Greatoutdoor.PRODUCT AS P ON OD.PRODUCT_NUMBER = P.PRODUCT_NUMBER
GROUP BY
    DATEPART(YEAR, OH.ORDER_DATE),
    CASE
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (9, 10, 11) THEN 'Herfst'
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (12, 1, 2) THEN 'Winter'
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (3, 4, 5) THEN 'Lente'
        WHEN DATEPART(MONTH, OH.ORDER_DATE) IN (6, 7, 8) THEN 'Zomer'
    END,
    P.PRODUCT_NUMBER,
    P.PRODUCT_NAME;

```

Figuur 21 - VIEW ProductSalesPerMonth

Helaas moest ik na het uitvoeren van deze view een onderzoek starten. Ik kwam er namelijk achter dat er enkel orders uit het jaar 2004 t/m 2006 werden getoond. Dit riep direct vraagtekens op. Bij nader onderzoek kwam ik tot de volgende conclusie. Ik combineerde de ORDER_HEADER tabel met de ORDER_DETAIL tabel. De ORDER_HEADER tabel heeft orders t/m het jaar 2021 in zich. Echter heeft ORDER_DETAIL enkel t/m het jaar 2006 orders in zich. Dit is hoe ik erachter kwam.

Order met de hoogste datum in de ORDER_HEADER Tabel.

<input type="checkbox"/> ORDER_DATE	<input type="checkbox"/> ORDER_NUMBER
2021-12-28	18963

Figuur 22 - Laatste order aan de hand van de datum in de ORDER_HEADER tabel

Order met het hoogste ordernummer op datum in de ORDER_HEADER tabel

<input type="checkbox"/> ORDER_DATE	<input type="checkbox"/> ORDER_NUMBER
2010-12-18	20752

Figuur 23 - Hoogste ordernummer in de ORDER_HEADER tabel

Wanneer ik onderstaande query uitvoerde op de ORDER_DETAILS tabel kom ik niet hoger dan order '9275'.

```
SELECT max(ORDER_NUMBER) AS ORDER_NUMBER  
from Greatoutdoor.ORDER_DETAILS
```

<input type="checkbox"/> ORDER_NUMBER
9275

Figuur 24 - Hoogste ordernummer bekend vanuit de tabel ORDER_DETAILS

Om erachter te komen welk ordernummer uit de ORDER_DETAILS tabel de laatste order is ten opzichte van de tijd (datum), heb ik de volgende query uitgevoerd.

```
SELECT TOP 1 oh.ORDER_DATE, od.ORDER_NUMBER  
FROM Greatoutdoor.ORDER_DETAILS AS od  
JOIN Greatoutdoor.ORDER_HEADER AS oh ON oh.ORDER_NUMBER = od.ORDER_NUMBER  
ORDER BY oh.ORDER_DATE DESC;
```

<input type="checkbox"/> ORDER_DATE	<input type="checkbox"/> ORDER_NUMBER
2006-12-24 06:14:01.000	6717

Door ook nog een test uit te voeren op de ORDER_HEADER tabel is te zien dat de order met het hoogste ordernummer uit de ORDER_DETAIL tabel, met de volgende ORDER_DATE terugkomt.

```
SELECT ORDER_NUMBER, ORDER_DATE FROM Greatoutdoor.ORDER_HEADER  
Where ORDER_NUMBER = 9275
```

<input type="checkbox"/> ORDER_NUMBER	<input type="checkbox"/> ORDER_DATE
9275	2006-12-16 14:37:34.000

Om toch met dit VIEW objectsoort verder te gaan en door aan te tonen dat deze view van nut is voor een bedrijf als Great Outdoor, baseer ik mijn voorbeelden op de beschikbare data in ORDER_DETAILS. Dus van het jaar 2004 t/m 2006. De reden hiervoor is dat ik PRODUCT_NAME nodig heb om producten te koppelen aan orders en het aantal verkochte items op te tellen voor een totaaloverzicht. PRODUCT_NAME komt alleen voor in de tabel ORDER_DETAILS en PRODUCT. Niet in ORDER_HEADER. Ik heb ook de ORDER_HEADER tabel nodig voor het bepalen van de order datum, ORDER_DATE. Deze komt enkel hierin voor.

Een medewerker van het bedrijf Great Outdoor zou door middel van onderstaande query per seizoen kunnen bekijken welk artikel het meest is verkocht. In onderstaand voorbeeld is gekozen om het totaal aantal producten verkocht in de zomer van 2004 te bekijken.

```

SELECT YEAR, SEASON, PRODUCT_NAME, TOTAL_QUANTITY SOLD
FROM ProductSalesPerMonth
WHERE SEASON = 'zomer'
AND YEAR = '2004'
ORDER BY TOTAL_QUANTITY SOLD DESC;

```

<input type="checkbox"/> YEAR	<input type="checkbox"/> SEASON	<input type="checkbox"/> PRODUCT_NAME	<input type="checkbox"/> TOTAL_QUANTITY SOLD
2004	Zomer	BugShield Extreme	9694
2004	Zomer	Sun Shelter 30	9684
2004	Zomer	Sun Shield	8320
2004	Zomer	BugShield Lotion	7928
2004	Zomer	BugShield Lotion Lite	7508
2004	Zomer	Sun Shelter 15	6958
2004	Zomer	BugShield Spray	6296
2004	Zomer	BugShield Natural	5130
2004	Zomer	Sun Blocker	4242
2004	Zomer	Insect Bite Relief	3440
2004	Zomer	Calamine Relief	2680
2004	Zomer	Firefly 2	2416
2004	Zomer	Firefly 4	2342

Figuur 25 – Top 14 bestverkopende producten zomer 2004

Zo zou dit ook vergeleken kunnen worden met een jaar verder, 2005 en verder 2006.

<input type="checkbox"/> YEAR	<input type="checkbox"/> SEASON	<input type="checkbox"/> PRODUCT_NAME	<input type="checkbox"/> TOTAL_QUANTITY SOLD
2005	Zomer	Sun Shelter 30	6802
2005	Zomer	BugShield Extreme	6800
2005	Zomer	BugShield Lotion Lite	5478
2005	Zomer	BugShield Lotion	4750
2005	Zomer	Sun Shield	4578
2005	Zomer	Sun Shelter 15	4550
2005	Zomer	Firefly 4	4484
2005	Zomer	Granite Carabiner	4210
2005	Zomer	Firefly Extreme	3516
2005	Zomer	BugShield Natural	3444
2005	Zomer	Firefly 2	3428
2005	Zomer	Sun Blocker	3336
2005	Zomer	TrailChef Water Bag	3190
2005	Zomer	BugShield Spray	3118

Figuur 26 -Top 14 bestverkopende producten zomer 2005

<input type="checkbox"/> YEAR	<input type="checkbox"/> SEASON	<input type="checkbox"/> PRODUCT_NAME	<input type="checkbox"/> TOTAL_QUANTITY SOLD
2006	Zomer	Granite Carabiner	5100
2006	Zomer	Mountain Man Digital	3522
2006	Zomer	BugShield Extreme	3318
2006	Zomer	Firefly 4	3282
2006	Zomer	Star Peg	3242
2006	Zomer	TrailChef Water Bag	2992
2006	Zomer	Double Edge	2908
2006	Zomer	Firefly Extreme	2872
2006	Zomer	Firefly 2	2858
2006	Zomer	Sun Shelter 30	2772
2006	Zomer	BugShield Spray	2666
2006	Zomer	BugShield Lotion Lite	2648
2006	Zomer	TrailChef Kitchen Kit	2610

Figuur 27 – Top 14 bestverkope producten zomer 2006

Bij nadere analyse lijkt het dat de “Bugshield Extreme” een seizoensgebonden product is. Weliswaar is dit product afhankelijk van het type zomer, veel zon, weinig zon, regen etc. Echter zou het bedrijf hier op voorhand dus meer van op voorraad moeten hebben in de zomers.

Om te testen of deze view klopt heb ik onderstaande select query gemaakt die test of het klopt dat er in de zomer van het jaar 2006, ‘3318’ stuks van het “BugShield Extreme” verkocht zijn. Dit is een optelsom van het totaal aantal verkochte stuks in de maanden juni, juli en augustus (maand 6, 7, 8).

Hiervóor heb ik onderstaande test query geschreven.

```
SELECT PRODUCT_NAME, SUM(OD.QUANTITY) AS TOTAL_BUGSHIELD_EXTREME_SOLD
FROM Greatoutdoor.ORDER_HEADER AS OH
JOIN Greatoutdoor.ORDER_DETAILS AS OD ON OH.ORDER_NUMBER = OD.ORDER_NUMBER
JOIN Greatoutdoor.PRODUCT AS P ON OD.PRODUCT_NUMBER = P.PRODUCT_NUMBER
WHERE DATEPART(YEAR, OH.ORDER_DATE) = 2006
    AND DATEPART(MONTH, OH.ORDER_DATE) IN (6, 7, 8)
    AND P.PRODUCT_NAME = 'BugShield Extreme'
group by PRODUCT_NAME;
```

Zoals onderstaand is af te lezen, klopt de uitkomst van deze test met het resultaat die wij ook vanuit de view krijgen.

<input type="checkbox"/> PRODUCT_NAME	<input type="checkbox"/> TOTAL_BUGSHIELD_EXTREME_SOLD
BugShield Extreme	3318

Hiermee ronden we het testen van de view af.

Edwin

Views

Een view is een database object en is een virtuele tabel (Assaf, 2023) waarvan de inhoud wordt gedefineerd door de query. Net zoals een tabel bestaat een view uit kolommen en rijen. De rijen en kolommen met data die uit de tabellen komen wordt dynamisch geproduceerd wanneer de view wordt gebruikt.

Een view werkt als een soort filter op de onderliggende tabellen die door de view gebruikt worden. De query definitie van de view kan gebruik maken van één of meer tabellen of kan gebruik maken van andere views in de huidige of een andere database.

Als de view gebruik maakt van data van een andere server is dit een gedistribueerde query. De gedistribueerde queries worden mogelijk gemaakt door de Distributed Query Processor (DQP) van SQL Server (Houser, 2022), echter gaat dit voor nu te ver om te beschrijven.

Over het algemeen worden views gebruikt om te focussen, simplificeren en te personaliseren van data uit de database. Verder worden views gebruikt als beveiligingsmechanisme zodat gebruikers toegang hebben tot de data via de view, zonder de gebruikers direct rechten te geven op de onderliggende tabellen in de view. Ook kunnen views gebruikt worden als backwards compatible interface. Met deze terugwaartse compatibiliteit bedoel ik dat het onderliggende schema kan wijzigen zonder dat de gebruiker hier last van heeft. De view werkt dus als een sort tussenlaag.

Indexed Views

Een indexed view is een view die gematerialiseerd is. Dit betekent dat de view berekend is en dat het resultaat van de view is opgeslagen net als een tabel. Indexed views kunnen de performance flink verhogen van sommige queries en werken het beste voor geagregeerde sets. Als de onderliggende data veel wijzigt is de indexed view minder geschikt.

System views

System view zijn views die catalog data bevatten en geven informatie over de metadata in de system base tables. Een voorbeeld van System views is gegeven in het hoofdstuk van de System catalog in voorbeeld 1.

Partitioned Views

Gepartitioneerde views zijn views die horizontal gepartitioneerde data van onderliggende tabellen van één of meer servers samenvoegd. (Assaf, 2023) Zo lijkt het dat de data uit één tabel komt. Als deze view onderliggende tabellen van dezelfde instantie als van SQL Server gebruikt dan spreken we van een local partitioned view.

Een gepartitioneerde view is bevat een UNION ALL van alle onderliggende tabellen. Deze tabellen moeten dezelfde kolom volgorde hebben en deze tabellen kunnen van dezelfde of verschillende servers afkomstig zijn. Een view word partitioned genoemd als deze de volgende opbouw heeft.

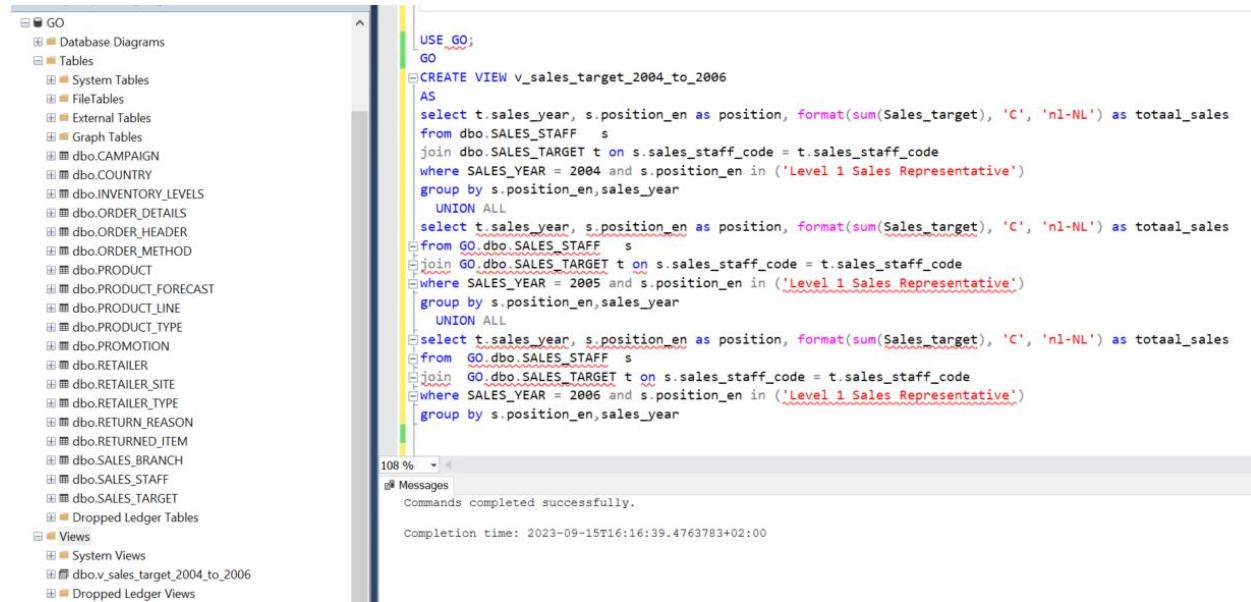
```
SELECT <select_list1>
FROM T1
UNION ALL
SELECT <select_list2>
FROM T2
UNION ALL
...
SELECT <select_listn>
FROM Tn;
```

Voorbeeld

De branch managers willen graag zien hoe de totale omzet zich heeft ontwikkeld van de *Level 1 Sales Representatives*.

Omdat alléén de informatie van de *Level 1 Sales Representatives* getoond dient te worden is er in de selectie een filter gebruikt. Om deze informatie te tonen heb ik gebruik gemaakt van een partitioned view.

De view is als volgt opgebouwd.



The screenshot shows the Object Explorer on the left with the 'Views' node expanded, showing 'dbo.v_sales_target_2004_to_2006'. The main pane displays the T-SQL code for creating the view:

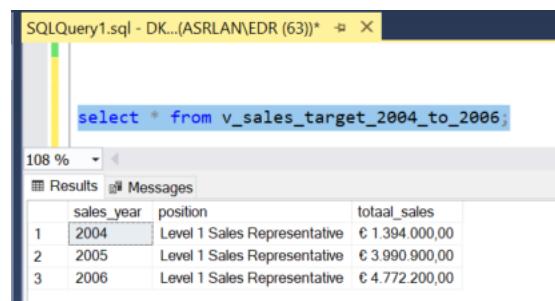
```
USE [GO];
GO
CREATE VIEW v_sales_target_2004_to_2006
AS
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'n1-NL') as totaal_sales
from dbo.SALES_STAFF s
join dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2004 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
UNION ALL
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'n1-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2005 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
UNION ALL
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'n1-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2006 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
```

The status bar at the bottom right indicates 'Commands completed successfully.' and 'Completion time: 2023-09-15T16:16:39.4763783+02:00'.

Afb. Local partitioned view

Hier is te zien dat er horizontaal gepartitioneerd is door de verschillende sets in de union samen te voegen. De gebruiker ziet hierdoor een versimpelde weergave net zoals in een tabel.

Het resultaat is view *v_sales_target_2004_to_2006* en met een select te benaderen.



The screenshot shows the results of the query 'select * from v_sales_target_2004_to_2006;' in the SQL Query window. The results grid shows the following data:

	sales_year	position	totaal_sales
1	2004	Level 1 Sales Representative	€ 1.394.000,00
2	2005	Level 1 Sales Representative	€ 3.990.900,00
3	2006	Level 1 Sales Representative	€ 4.772.200,00

Afb. view *v_sales_target_2004_to_2006*

In de Views folder is de aangemaakte view te zien.



Afb. Partitioned view.

Testen Partitioned View

Uitgangssituatie

De selectie van de query moet weergegeven worden via de aangemaakte view. Er mogen niet meer of minder rijen worden getoond in de aangemaakte view. De query is als volgt.

The screenshot shows a SQL query in the query editor and its execution results in the results pane. The query is a UNION ALL of three separate SELECT statements, each filtering for a specific year (2004, 2005, or 2006) and position ('Level 1 Sales Representative'). The results show the total sales for each year and position.

```
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'nl-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2004 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
UNION ALL
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'nl-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2005 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
UNION ALL
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'nl-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2006 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
```

sales_year	position	totaal_sales
1 2004	Level 1 Sales Representative	€ 1.394.000,00
2 2005	Level 1 Sales Representative	€ 3.990.900,00
3 2006	Level 1 Sales Representative	€ 4.772.200,00

Afb. Selectie voor de Partitioned view.

Uitvoeren test

Door de query en de view met elkaar te vergelijken controleer ik of de view de juiste resultaten bevat.

Query resultaat:

The screenshot shows the same query as above, but this time it is run against a view named 'v_sales_target_2004_to_2006'. The results are identical to the previous screenshot, showing the total sales for each year and position.

```
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'nl-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2004 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
UNION ALL
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'nl-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2005 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
UNION ALL
select t.sales_year, s.position_en as position, format(sum(Sales_target), 'C', 'nl-NL') as totaal_sales
from GO.dbo.SALES_STAFF s
join GO.dbo.SALES_TARGET t on s.sales_staff_code = t.sales_staff_code
where SALES_YEAR = 2006 and s.position_en in ('Level 1 Sales Representative')
group by s.position_en,sales_year
```

sales_year	position	totaal_sales
1 2004	Level 1 Sales Representative	€ 1.394.000,00
2 2005	Level 1 Sales Representative	€ 3.990.900,00
3 2006	Level 1 Sales Representative	€ 4.772.200,00

Afb. Query resultaat t.b.v. de partition view.

Resultaat van de view `v_sales_target_2004_to_2006`:

The screenshot shows a SQL query being run in SSMS:

```
select * from v_sales_target_2004_to_2006
```

The results pane displays the following data:

	sales_year	position	totaal_sales
1	2004	Level 1 Sales Representative	€ 1.394.000,00
2	2005	Level 1 Sales Representative	€ 3.990.900,00
3	2006	Level 1 Sales Representative	€ 4.772.200,00

Afb. View resultaat.

Test resultaat

De selectie van de query levert hetzelfde resultaat als de uitvoer van de view. Het resultaat van de test is akkoord.

Lesley-Ann

In de laatste opdracht werd ons gevraagd om de behoefte van minimaal een nieuwe objectsoort te bedenken. Hierbij dacht ik aan een VIEW of TRIGGER

Ik heb gekozen om te werken met een view, voordat ik mijn voorbeeld presenteert eerst wat uitleg over view. Views zijn database objecten die door de query hun inhoud ontvangen. Een view bestaat uit rijen en kolommen met data uit de database. De tabellen worden dynamisch aangemaakt wanneer de view wordt aangeroepen. Dit zorgt ervoor dat de view een virtuele tabel is. Een virtuele tabel slaat geen gegevens op maar geeft gegevens pas weer wanneer deze wordt aangeroepen. De view kan gebruik maken van een of meer tabellen en kan ook gebruik maken van andere views. Views worden vaak gebruikt om data uit te lichten en data simpel te presenteren. Ook worden views gebruikt als beveiligingsmethode, views staan gebruikers toe informatie uit de database in te zien zonder dat de gebruiker de rechten heeft over de tabellen die de view gebruikt. (Microsoft, 2023)

De behoefte die ik bedacht heb voor de opdracht is dat de sales manager en voorraadbeheerder informatie willen verkrijgen over de inventaris van het bedrijf en de verwachte verkoop van 2006. Deze query zou ook aangepast kunnen worden voor de komende jaren en zo hergebruikt kunnen worden. Zo kan er een analyse worden uitgevoerd over de inkoop en verkoop en kan een sales manager kijken hoeveel teveel er is ingekocht of hoeveel er bijvoorbeeld bij gekocht zou moeten worden in de toekomt.

De resultaten worden weergegeven in 'SalesByCategory', dit kan worden aangeroepen om snel inzicht te krijgen in de voorraad en verkoop details van de database.

```

CREATE VIEW SalesByCategory2 AS
SELECT
    P.PRODUCT_NUMBER,
    P.PRODUCT_NAME,
    SUM(IL.INVENTORY_COUNT) AS TotalInventoryCount,
    SUM(PF.EXPECTED_VOLUME) AS TotalExpectedVolume
FROM
    dbo.PRODUCT P
LEFT JOIN
    dbo.INVENTORY_LEVELS IL ON P.PRODUCT_NUMBER = IL.PRODUCT_NUMBER
LEFT JOIN
    dbo.PRODUCT_FORECAST PF ON P.PRODUCT_NUMBER = PF.PRODUCT_NUMBER
    AND PF.YEAR = 2006
GROUP BY
    P.PRODUCT_NUMBER, P.PRODUCT_NAME;

SELECT * FROM SalesByCategory2;

```

107 %

	PRODUCT_NUMBER	PRODUCT_NAME	TotalInventoryCount	TotalExpectedVolume
1	23	Hibernator Camp Cot	419136	38232
2	46	Granite Climbing Helmet	751056	45096
3	69	Polar Ice	408144	42444
4	92	Sun Shelter Stick	2401728	91116
5	115	Course Pro Gloves	1238544	252900
6	29	Canyon Mule Carryall	2060328	260712
7	75	Edge Extreme	1211568	474228
8	9	TrailChef Kettle	801912	145404
9	15	Star Gazer 6	492216	143172
10	109	Course Pro Putter	1224864	166104
11	89	BugShield Lotion	8222616	403956
12	3	TrailChef Kitchen Kit	2053416	373428
13	52	Granite Pulley	1814952	145680
14	95	Sun Shield	8411256	401292
15	72	Polar Extreme	701136	145584
16	66	Mountain Man Combination	356064	114048

Figuur 28 View sales by category

Ook voor deze VIEW heb ik een test geschreven. In deze test word gecheckt of het aantal rows overeenkomt met het verwachte aantal rows. De rows die ik verwacht heb ik bekijken in de database, deze heb ik ingevuld in de query. Het ging om 115 rows, dit bleek correct te zijn en de test was geslaagd. In test 2 heb ik het verwachte aantal rows foutief ingevuld als 120 rows, zoals verwacht faalde de test.

```
CREATE TABLE #TestResults (
    ProductNumber INT,
    ProductName NVARCHAR(255),
    TotalQuantitySold INT,
    TotalSalesAmount MONEY
);

INSERT INTO #TestResults
SELECT
    P.PRODUCT_NUMBER,
    P.PRODUCT_NAME,
    SUM(OD.QUANTITY) AS TotalQuantitySold,
    SUM(OD.UNIT_PRICE * OD.QUANTITY) AS TotalSalesAmount
FROM
    dbo.PRODUCT P
JOIN
    dbo.ORDER_DETAILS OD ON P.PRODUCT_NUMBER = OD.PRODUCT_NUMBER
GROUP BY
    P.PRODUCT_NUMBER, P.PRODUCT_NAME;

IF (SELECT COUNT(*) FROM #TestResults) = 115
BEGIN
    PRINT 'Correct amount of rows';
END
ELSE
BEGIN
    PRINT 'Failed, Incorrect amount of rows.';
END
```

Messages

Warning: Null value is eliminated by an aggregate or other SET operation.

(115 rows affected)

Correct amount of rows

Figuur 29 View test 1

```

CREATE TABLE #TestResults (
    ProductNumber INT,
    ProductName NVARCHAR(255),
    TotalQuantitySold INT,
    TotalSalesAmount MONEY
);

INSERT INTO #TestResults
SELECT
    P.PRODUCT_NUMBER,
    P.PRODUCT_NAME,
    SUM(OD.QUANTITY) AS TotalQuantitySold,
    SUM(OD.UNIT_PRICE * OD.QUANTITY) AS TotalSalesAmount
FROM
    dbo.PRODUCT P
JOIN
    dbo.ORDER_DETAILS OD ON P.PRODUCT_NUMBER = OD.PRODUCT_NUMBER
GROUP BY
    P.PRODUCT_NUMBER, P.PRODUCT_NAME;

IF (SELECT COUNT(*) FROM #TestResults) = 120
BEGIN
    PRINT 'Correct amount of rows';
END
ELSE
BEGIN
    PRINT 'Failed, Incorrect amount of rows.';
END

```

107 %

Messages

Warning: Null value is eliminated by an aggregate or other SET operation.

(115 rows affected)

Failed, Incorrect amount of rows.

Figuur 30 View test 2

Sander

Voor het uitbreiden van de database met een nieuw (nog niet eerder bestudeerd) objecttype, heb ik nogmaals besloten te kijken naar de query “Hoeveelheid werk per ordermethode” uit hoofdstuk 2 - Queries. Ik vermoedde namelijk dat deze nog simpeler kon door niet functions naar views te gebruiken om de CTE’s mee te vervangen.

De eerste CTE zou door de volgende view vervangen kunnen worden:

```

CREATE OR ALTER VIEW go_view_total_order_value AS
SELECT o_details.ORDER_NUMBER,
       SUM(o_details.QUANTITY * o_details.UNIT_PRICE) TOTAL_VALUE
FROM ORDER_HEADER o_header
INNER JOIN ORDER_DETAILS o_details
       ON o_details.ORDER_NUMBER = o_header.ORDER_NUMBER
GROUP BY o_details.ORDER_NUMBER

```

De tweede CTE zou door de volgende view vervangen kunnen worden:

```

CREATE OR ALTER VIEW go_view_order_method_to_amount_of_manual_work AS
SELECT method.ORDER_METHOD_CODE,
CASE
    WHEN method.ORDER_METHOD_CODE = 1 THEN 'Least manual work'
    WHEN method.ORDER_METHOD_CODE = 2 THEN 'Most manual work'
    WHEN method.ORDER_METHOD_CODE = 3 THEN 'Average manual work'
    WHEN method.ORDER_METHOD_CODE = 4 THEN 'Average manual work'
    WHEN method.ORDER_METHOD_CODE = 5 THEN 'Least manual work'
    WHEN method.ORDER_METHOD_CODE = 7 THEN 'Most manual work'
    WHEN method.ORDER_METHOD_CODE = 8 THEN 'Most manual work'
    ELSE 'UNKNOWN' END AMOUNT_OF_MANUAL_WORK
FROM ORDER_METHOD method

```

De views filteren niet meer op respectievelijk ORDER_NUMBER en ORDER_METHOD_CODE, maar geven de resultaten voor alle regels van respectievelijk alle orders en alle order methodes.

De hoofdquery zal er als volgt uit gaan zien:

```

SELECT work.AMOUNT_OF_MANUAL_WORK,
       COUNT(*)                               ORDER_COUNT,
       CAST(SUM(order_v.TOTAL_VALUE) AS DECIMAL(20, 2)) AS TOTAL_REVENUE
FROM ORDER_METHOD method
      INNER JOIN ORDER_HEADER order_h
              ON order_h.ORDER_METHOD_CODE = method.ORDER_METHOD_CODE
INNER JOIN go_view_total_order_value order_v ON order_v.ORDER_NUMBER =
order_h.ORDER_NUMBER
INNER JOIN go_view_order_method_to_amount_of_manual_work work ON
work.ORDER_METHOD_CODE = order_h.ORDER_METHOD_CODE
GROUP BY work.AMOUNT_OF_MANUAL_WORK
ORDER BY ORDER_COUNT DESC

```

De CROSS APPLYS zijn vervangen door equivalente INNER JOINS, wat mogelijk gemaakt is door het gebruik van views in plaats van functions.

Conclusie

Door de gegeven opdrachten en literatuur over database gerelateerde concepten hebben wij zowel individueel als, als groep veel geleerd. Veel delen van database ontwerp zijn langsgekomen. Wij hebben onze tanden kunnen zetten in onder andere complexe query's, stored procedures, views, normalisering en testten. Dit hebben wij als team kunnen doen in de gezamenlijke delen maar wij mochten onze individuele creativiteit ook loslaten dankzij de individuele opdrachten.

Wij hebben als groep geconcludeerd dat kennis over database objecten en query's van belang is om een efficiënte database te kunnen onderhouden. Het ontwerpen, ontwikkelen en onderhouden van een database luistert nauw en kan foutgevoelig zijn. Wij zijn van mening dat wij in een korte tijd van drie weken veel hebben kunnen leren maar wij erkennen ook dat er nog veel te leren en ontdekken valt als het gaat om databases.

Kortom, deze opdracht heeft de waarde van kennis en vaardigheden over SQL benadrukt voor ons als software engineers. Wij kijken uit naar de rest van het semester om onze kennis van data en databases verder op te bouwen.

Bibliography

- Griffin, J. (2013, september 10). *Understanding the SQL Server System Catalog*. Retrieved from logicalread.com: <https://logicalread.com/sql-server-system-catalog-mc03/#:~:text=The%20system%20catalog%20consists%20of,the%20system%20to%20functions%20properly>.
- Petkovic, D. (2020). Microsoft SQL Server 2019 - A Beginner's Guide. In D. Petkovic, *Microsoft SQL Server 2019 - A Beginner's Guide*. McGraw-Hill. Retrieved from <https://learning.oreilly.com/library/view/microsoft-sql-server/9781260458886/ch09.xhtml#ch9lev1sec2>
- RouseTechnologie, M. (2014, september 17). *Systeemcatalogus*. Retrieved from techopedia.com: <https://www.techopedia.com/definition/22442/system-catalog>
- Microsoft. (2023, 05 23). *Microsoft SQL Server*. Retrieved from SELECT - OVER Clause (Transact-SQL): [https://learn.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view\(sql-server-ver16](https://learn.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view(sql-server-ver16)
- Petković, D. (2020). *Microsoft SQL Server 2019: A Beginner's Guide*. McGraw-Hill.
- Assaf, W. D. (2023, 3 1). *Views*. Retrieved from Microsoft SQL Server: [https://learn.microsoft.com/en-us/sql/relational-databases/views/views?view\(sql-server-ver16](https://learn.microsoft.com/en-us/sql/relational-databases/views/views?view(sql-server-ver16)
- Houser, C. (2022, 04 19). *Query Processor*. Retrieved from Microsoft : <https://learn.microsoft.com/en-us/host-integration-server/core/query-processor2>
- Microsoft. (2023, April 3). *Stored Procedures (Database Engine)*. Retrieved September 12, 2023, from Microsoft Learn: [https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view\(sql-server-ver16](https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view(sql-server-ver16)
- Oracle Corporation. (1999). *Advantages of Stored Procedures*. Retrieved September 12, 2023, from Oracle 8i/Java Stored Procedures Developer's Guide: https://docs.oracle.com/cd/F49540_01/DOC/java.815/a64686/01_intr3.htm
- Microsoft. (2023, March 1). *Transactions (Transact-SQL)*. Retrieved September 13, 2023, from Microsoft Learn: [https://learn.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view\(sql-server-ver16](https://learn.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view(sql-server-ver16)
- Microsoft. (2023, July 13). *Return data from a stored procedure*. Retrieved September 13, 2023, from Microsoft Learn: [https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/return-data-from-a-stored-procedure?view\(sql-server-ver16](https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/return-data-from-a-stored-procedure?view(sql-server-ver16)
- Microsoft. (2023, June 30). *Create user-defined functions (Database Engine)*. Retrieved September 14, 2023, from Microsoft Learn: [https://learn.microsoft.com/en-us/sql/relational-databases/user-defined-functions/create-user-defined-functions-database-engine?view\(sql-server-ver16](https://learn.microsoft.com/en-us/sql/relational-databases/user-defined-functions/create-user-defined-functions-database-engine?view(sql-server-ver16)
- Job Habraken, A. M. (2023, September). *Brightspace - De Haagse Hogeschool*. Retrieved September 14, 2023, from Week 3 - Donderdag: Functions, Stored Procedures & verdieping: https://brightspace.hhs.nl//content/enforced/56642-H-SE-AD-22_2023_DT/Week%203%20DB.pdf?_d2lSessionVal=sP0yTGZoKMAiYlyLmWzeDu0QJ&ou=56642

- Microsoft. (2023, May 23). *User-defined functions*. Retrieved September 14, 2023, from Microsoft Learn: <https://learn.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions?view=sql-server-ver16>
- Microsoft. (2023, 01 03). *sys.allocation_units (Transact-SQL)*. Retrieved 09 16, 2023, from learn.microsoft.com: <https://learn.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-allocation-units-transact-sql?view=sql-server-ver16>
- Hughes, A. (2019, April). *stored procedure*. Retrieved from techtarget.com: <https://www.techtarget.com/searchoracle/definition/stored-procedure>
- Hogeschool, H. (2023, September). *Brightspace*. Retrieved from Week 3 Advanced Databases: https://brightspace.hhs.nl/content/enforced/56642-H-SE-AD-22_2023_DT/Week%203%20DB.pdf?ou=56642
- W3Schools. (n.d.). *SQL Stored Procedures for SQL Server*. Retrieved from W3Schools: https://www.w3schools.com/sql/sql_stored_procedures.asp
- Microsoft. (2023, Juli 19). *Create a stored procedure*. Retrieved from lean.microsoft.com: <https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/create-a-stored-procedure?view=sql-server-ver16>
- Factor, P. (n.d.). *SQL Server User-Defined Functions*. Retrieved from Redgate: <https://www.red-gate.com/simple-talk/databases/sql-server/t-sql-programming-sql-server/sql-server-user-defined-functions/>
- Microsoft. (2023, Mei 23). *What are the SQL database functions?* Retrieved from learn.mircrosoft.com: <https://learn.microsoft.com/en-us/sql/t-sql/functions/functions?view=sql-server-ver16>
- Papiernik, M. (2022, November 2022). *How To Use Functions in SQL*. Retrieved from Digitalocean: <https://www.digitalocean.com/community/tutorials/how-to-use-functions-in-sql>
- Larsen, G. (2019, Juni 6). *Displaying the Object Explorer Details Pane in SQL Server Management Studio*. Retrieved from Database journal: <https://www.databasejournal.com/ms-sql/displaying-the-object-explorer-details-pane-in-sql-server-management-studio/>
- Careerride.com. (n.d.). *SQL Server - Database Objects*. Retrieved from Careerride.com: <https://www.careerride.com/SQL-Server-database-objects.aspx>
- Choudhury, D. (n.d.). *Creating SQL VIEWS: Explained Step-by-Step*. Retrieved from Geekflare: <https://geekflare.com/sql-views-explained/>
- Corey, T. (2017, 9 20). *SQL Stored Procedures - What They Are, Best Practices, Security, and More*. Retrieved 9 15, 2023, from Youtube: https://www.youtube.com/watch?v=Sggdhot-MoM&ab_channel=IAmTimCorey
- Babu, R. (2019, 7 29). *SQL Server stored procedures for beginners*. Retrieved 9 12, 2023, from SQL Shack: <https://www.sqlshack.com/sql-server-stored-procedures-for-beginners/>
- Tewatia, D. (2023, 1 11). *Why We Use Stored Procedure In SQL Server*. Retrieved 9 15, 2023, from C# Corner: <https://www.c-sharpcorner.com/article/why-we-use-stored-procedure-in-sql-server/>
- D, J. (2021, 2 21). *SCALAR VALUED FUNCTION: THE ULTIMATE GUIDE FOR BEGINNERS*. Retrieved 9 16, 2023, from Simple SQL tutorials: <https://simplesqltutorials.com/scalar-valued-functions-guide-for-beginners/>

- Factor, P. (2017, 5 23). *SQL Server User-Defined Functions*. Retrieved 9 16, 2023, from RedGate:
<https://www.red-gate.com/simple-talk/databases/sql-server/t-sql-programming-sql-server/sql-server-user-defined-functions/>
- Scalar Functions in SQL. (2023, 9 16). Retrieved from Java T point:
<https://www.javatpoint.com/scalar-functions-in-sql>
- Erkec, E. (2019, 8 29). *SQL Server inline table-valued functions*. Retrieved 9 16, 2023, from SQL Shack:
<https://www.sqlshack.com/sql-server-inline-table-valued-function/>
- SDET. (2021, 10 18). *Database Testing / Stored Procedure Testing / How To Test Stored Procedures*. Retrieved 9 15, 2023, from Youtube: https://www.youtube.com/watch?v=-FZo3BolyT0&ab_channel=SDET-QAAutomationTechie
- Microsoft. (2023, 5 23). *CREATE VIEW (Transact-SQL)*. Retrieved 9 15, 2023, from Learn Microsoft:
<https://learn.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql?view=sql-server-ver16>
- Microsoft. (2023, 05 23). *SUM (Transact-SQL)*. Retrieved 9 16, 2023, from Microsoft SQL Server:
<https://learn.microsoft.com/en-us/sql/t-sql/functions/sum-transact-sql?view=sql-server-ver16>
- Microsoft. (2023, Juli 19). *SQL Server 2022*. Retrieved 9 16, 2023, from Create a stored procedure:
<https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/create-a-stored-procedure?view=sql-server-ver16>
- Microsoft. (2023, March 3). *Walkthrough: Creating and Running a SQL Server Unit Test*. Retrieved September 30, 2023, from Microsoft Learn: <https://learn.microsoft.com/en-us/sql/ssdt/walkthrough-creating-and-running-a-sql-server-unit-test?view=sql-server-ver16>

Bijlagen

Bijlage 1

