Access voor Beginners - Hoofdstuk 17

Handleiding van Helpmij.nl

Auteur: OctaFish

Februari 2013

Variabele gegevens gebruiken (Normaliseren vervolg)

Vanaf het begin van de cursus Access heb ik regelmatig het begrip '*Normaliseren'* laten vallen. En ook op het forum kom je die term regelmatig tegen in berichten. Maar wat zijn de valkuilen van een te goed genormaliseerde tabel? En hoe ver moet je überhaupt gaan met normaliseren? Op die vragen ga ik nu proberen een antwoord te geven.

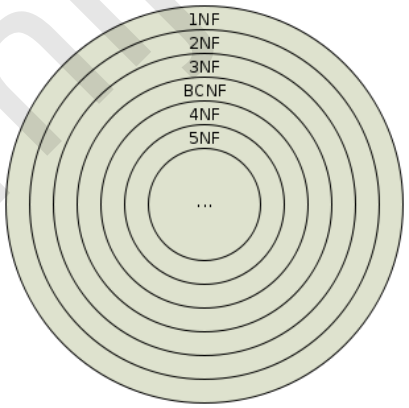
Om nog even op te frissen hoe normaliseren ook al weer werkt, een kort fragment uit Hoofdstuk 2 van de cursus:

Het normaliseren van een database

Normaliseren is, zoals we waarschijnlijk nu wel weten, het proces waarbij we de structuur van de database beoordelen op o.a. het voorkomen en voorkomen (wat is onze taal toch mooi: één woord met twee totaal verschillende betekenissen: vóórkomen en voorkómen bedoel ik dus J) van dubbele gegevens, de zogenoemde *dataredundantie*. Daarom slaan we in een tabel Bestellingen niet alle klantgegevens op, maar alleen een KlantID. Want op basis van dat klantID kun je de voor de bestelling noodzakelijke gegevens middels een query makkelijk opzoeken. En voor het bestelde artikel vullen we het ArtikelID, en niet alle artikelgegevens.

Een proces om gegevensverzamelingen te optimaliseren is het *normaliseren* van de tabellen.

Daarbij wordt per tabel bekeken welke gegevens we willen gebruiken, en hoe die gegevens zich verhouden tot de overige gegevens in dezelfde tabel. We onderscheiden daarbij een aantal niveaus:



- 1e Normaalvorm
- 2e Normaalvorm
- 3e Normaalvorm
- 4e Normaalvorm
- 5e Normaalvorm

Deze normalisatie vorm is vastgelegd in de Boyce-Codd normaalbeschrijving. In het lijstje geldt dat elk opvolgend niveau het vorige niveau overerft, en er nieuwe voorwaarden aan toevoegt.
De 2e normaalvorm voldoet dus aan de 1e normaalvorm en voegt nieuwe eisen toe; de 3e normaalvorm voldoet aan de 2e normaalvorm (en dus automatisch ook aan de 1e) en voegt nieuwe voorwaarden toe.

Meestal wordt een database genormaliseerd tot de 3e normaalvorm; dat is voor een gemiddelde database doorgaans genoeg. In deze cursus gaan we dan ook niet verder dan de 3e normaalvorm.

Nevenaspecten van normaliseren

Normaliseren is prachtig, en helpt je bij het voorkomen van dubbele gegevens. Tot zover dus niks dan goeds. Maar sommige gegevens fluctueren regelmatig, en laten zich niet zomaar normaliseren. Om dat te visualiseren, hoef je alleen maar naar de tabel Klanten te kijken.

Klantgegevens zijn doorgaans stabiele gegevens. Een klantnaam verandert zelden, zijn adres zelden. In die gevallen dat er een mutatie is, kun je die simpel doorvoeren in de tabel Klanten. Contactpersonen zullen wat sneller veranderen, maar daarvoor heb je de tabel Contactpersonen, en die bekijk je dan weer op dezelfde manier: het geslacht van een CP zal niet heel snel veranderen, zijn naam ook niet etc. Misschien de functie of telefoonnummer, maar die zijn ook snel aangepast.

Anders wordt het als je naar de Klanten kijkt in relatie tot een Bestelling. De klant(gegevens) liggen redelijk vast, maar wat te denken van het Afleveradres? Een klant kan een bestelling laten afleveren

op zijn bekende adres, maar een specifieke bestelling kan net zo goed ergens anders afgeleverd moeten worden. Dus waar is dat afleveradres dan van afhankelijk? Van de klant, of van de bestelling? Je zou zeggen: het afleveradres is een eigenschap van de Bestelling, niet van de klant!

Kortom: de Normalisatie zou zeggen: klantID à Klantadres, maar bij een bestelling kun je net zo goed aparte velden nodig hebben voor de aflevering. Hoe je dat oplost, is een ander verhaal (in tabel Bestellingen, of middels een gekoppelde tabel Afleveradressen bijvoorbeeld).

Ander voorbeeldje: adressen. Normaliseren schrijft voor dat je herhalende gegevens uit de tabel verwijdert en in een eigen tabel opslaat. In extremis kun je zeggen dat straatnamen en huisnummers in een grote tabel met adressen ook herhalend zijn. Toch zal niemand het in zijn hoofd halen om huisnummers en straatnamen in eigen tabellen onder te brengen, en ze te koppelen. Al kun je, als je echt wilt standaardiseren, natuurlijk een koppeling maken met een postcodetabel, en adressen evalueren a.d.h.v. postcode + huisnummer.

Kortom: Normaliseren is een goede basis, maar zonder gezond verstand kun je eigenlijk geen goede database maken.

Case: de variabele artikelprijzen

In een database die in de 3e Normaal staat, heb je een aparte tabel voor Bestellingen met daarin de klantgegevens en bestellinggegevens, en een aparte tabel met de bestelde artikelen. Die zijn dus uit de tabel Bestellingen genormaliseerd door in de tabel [tBestellingRegels] een veld [BestellingID] op te nemen, en de tabel [tBestellingRegels] is op zijn beurt weer gekoppeld aan de tabel [tArtikelen] op basis van het veld [ArtikelID].

Een tabel [tBestellingRegels] zou er dus zo uit kunnen zien qua velden:

BestelRegelID, BestellingID, ArtikelID, Aantal, Datum_Geleverd

Bij elk nieuw artikel in de bestelling wordt dus een nieuw record toegevoegd, waarbij verwezen wordt naar het Bestelnummer, en het Artikelnummer. Uiteraard moet je wel het aantal te leveren artikelen per artikel invoeren, en is een veld Datum_Geleverd per artikel ook wel nuttig, omdat het kan voorkomen dat bij een aflevering niet alle bestelde artikelen geleverd kunnen worden. Die blijven dan in backorder staan.

De tabel [tArtikelen] bevat dan deze velden:

ArtikelID, Artikel, Prijs, BedrijfID, Opmerkingen, Inactief

In een query (voor een factuur bijvoorbeeld) kun je dan een volledige bestelling genereren door uit tBestellingen, tBestellingRegels, tArtikelen en tKlanten de juiste velden te selecteren, zoals [Prijs] uit [tArtikelen] en [Aantal] uit [tBestellingRegels]. In die query kun je dan een veld maken met de formule =[Aantal] * [Prijs] en daarmee weet je dan wat de totaalprijs is voor dat bestelde artikel.

Deze constructie is fantastisch genormaliseerd, maar heeft minstens één groot nadeel: als de prijs in de tabel [tArtikelen] verandert, dan heb je een stevig probleem! Niet zozeer voor de klant, want die heeft (bijvoorbeeld per mail) een document ontvangen met daarop de bedragen die hij moet betalen. Het probleem ligt in de db, waarin de ontvangen bedragen na verloop van tijd (namelijk vanaf het moment dat de eerste prijs is gewijzigd) niet meer kloppen met de overzichten die je zelf gebruikt, bijvoorbeeld als je een Totalen query maakt om de maandomzet te berekenen. Bij elke prijsverandering verandert de maandomzet. En dat mag natuurlijk niet!

De oplossingen!

De oplossing is op een aantal manieren uit te voeren, waarvan ik er in dit hoofdstuk 4 ga behandelen. En om het simpel te houden, zijn die oplossingen weer onder te verdelen in twee hoofdgroepen: een oplossing met behoud van (prijs)historie, en een oplossing zonder historie. De laatste variant is daarbij het simpelst, dus daar gaan we mee beginnen!

Oplossing 1: prijzen opslaan zonder historie

Als je wilt voorkomen dat de prijzen in een bestelling niet meer kloppen, dan is er maar één oplossing: de prijs opslaan in de tabel! En daartoe heb je dus een extra veld nodig in de tabel [tBestellingRegels], die je dan natuurlijk [Prijs] noemt. De tabel ziet er dan zo uit:

BestelRegelID, BestellingID, ArtikelID, Aantal, Prijs, Datum_Geleverd

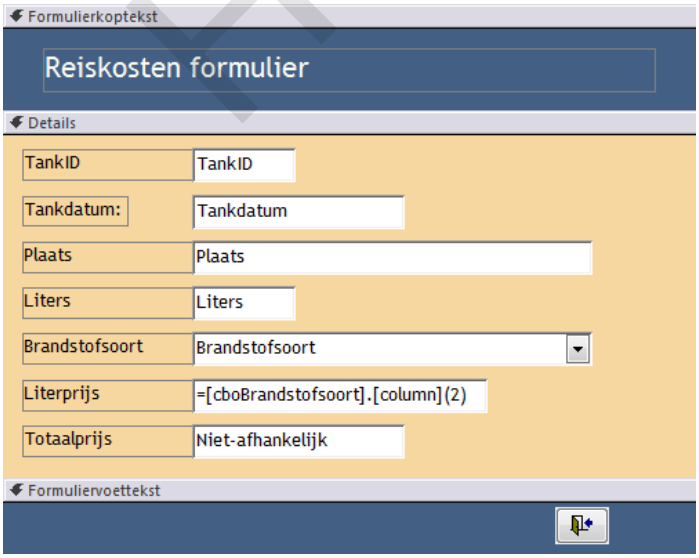
Dit levert dan wel gelijk weer een probleem op, want hoe ga je dat vullen met de juiste artikelprijs? Het ArtikelID halen we uit de tabel tArtikelen, waar ook de prijs staat. Je kunt natuurlijk een keuzelijst maken voor ArtikelID, en één voor ArtikelPrijs. Maar dat is niet erg handig, want dat laat de mogelijkheid open om een verkeerde prijs bij een artikel te selecteren. En we willen natuurlijk geen fouten inbouwen.

Laten we eens kijken naar een voorbeeldje op basis van een Reiskosten formulier. Dat maakt gebruik van deze tabel:

	PrijsID	BrandstofType	Wijzigingsdatum	Prijs
	0001	Euro Normaal	08-01-2013	€ 1,68.9
	0002	Diesel	15-01-2013	€ 1,48.8
	0003	Super	28-12-2012	€ 1,88.2
	0004	Gas	11-01-2013	€ 0,97.4
*	(Nieuw)		18-01-2013	

Je ziet een viertal brandstoftypes met hun prijs. Het datumveld is niet eens nodig, want je doet daar niet zoveel mee, anders dan controleren of de medewerkers de prijs wel op de juiste dag hebben bijgewerkt. Maar het gaat natuurlijk om het veld [Prijs].

In het formulier Reiskosten gebruiken we de tabel [Reiskosten] als basis. We slaan een datum op, het aantal liters, en middels de keuzelijst Brandstofsoort selecteren we het juiste type brandstof. De tabel is al aangepast volgens de techniek die hierboven is uitgelegd, dus we slaan de prijs van de brandstof per invuldag op.



Normaal gesproken zouden we, als we de prijs niet opslaan, maar wel willen gebruiken, voor txtLiterPrijs een niet-afhankelijk veld gebruiken, en de prijs uit de keuzelijst cboBrandstofsoort halen. Die heeft deze code als Rijbron:

```
SELECT DISTINCT PrijsID, Switch([BrandstofType]=1,"Euro
Normaal",[BrandstofType]=2,"Super",[BrandstofType]=3,"Diesel",
```

[BrandstofType]=4,"Gas") AS Brandstof, Prijs FROM tBrandstofprijs_eenvoudig;

In de query zit de functie SWITCH die nog niet in de cursus aan bod is gekomen, en die misschien enige uitleg behoeft. De functie kun je gebruiken als een vervanger van de IIF functie, als het gaat om het vervangen van een veldwaarde door een vervangende waarde of tekst. Je geeft in de functie aan om welk veld het gaat, wat de waarde moet zijn en wat dan de vervangende waarde is. In algemene zin:

Switch([Veld]=Waarde1;"Nieuwe tekst 1"; [Veld]=Waarde2;"Nieuwe tekst 2")

Switch is in tegenstelling tot IIF eendimensionaal; je moet de functie dus niet gebruiken om functies te nesten. In IIF kun je maximaal 7 niveaus diep nesten, en met wat trucjes kun je dus zo'n 14 verschillende waarden substitueren. Switch heeft die beperking niet; bovendien is de syntax vele malen overzichtelijker dan een op 7 niveaus geneste IIF functie. In dit geval wordt de Switch functie gebruikt om de keuzelijst voor brandstof van de juiste tekst te voorzien. De keuzelijst in de tabel maakt gebruik van twee kolommen, waarbij kolom 1 de getallen 1-4 bevat, en de tweede kolom de naam van de brandstof. En met de Switch worden de getallen weer terugvertaald naar tekst.

Het tekstvak Literprijs vul je door als besturingselementbron de formule = cboBrandstofsoort.Column(2) te gebruiken, zoals in de afbeelding.

Maar dat levert dus het eerder aangehaalde probleem op, dat de prijs steeds verandert al naargelang de waarde in de tabel tBrandstof. Het veld [Prijs] moet dus gekoppeld zijn aan het tekstveld. De oplossing is, dat we het tekstveld txtLiterPrijs vullen *vanuit de keuzelijst*. Oftewel: als we een brandstof kiezen, dan wordt de prijs daarvan gelijk ingevuld in het tekstveld txtLiterPrijs. Je kunt dat doen bij de gebeurtenis van de keuzelijst. Tegelijkertijd kun je dan een formule maken die daarna de Totaalprijs berekent, want als je de prijs weet, en het aantal liters is ingevuld, dan kun je [Aantal] * [Prijs] uitvoeren om de totaalprijs te berekenen. De code op de keuzelijst ziet er dan zo uit:

```
Private Sub cboBrandstofsoort_Click()
   Me.txtLiterprijs = Me.cboBrandstofsoort.Column(2)
   Me.txtTotaalprijs = Me.txtLiterprijs * Me.txtLiters
End Sub
```

Korte uitleg: in de keuzelijst cboBrandstofsoort zitten 3 velden, waarvan het derde veld de prijs bevat. Deze wordt met Column(2) uitgelezen (lees de eerdere hoofdstukken als je niet weet waarom dit is). Vervolgens wordt de berekening uitgevoerd.

Voordelen van deze methode:

- Makkelijk te implementeren (veld toevoegen, en koppelen in het formulier)
- Betrouwbaar

Nadeel:

- Geen historie op prijzen
- Extra veld nodig voor de prijs
- De prijs is simpel aan te passen in de tabel, dus data lastig te locken

Of het nadeel ook echt een nadeel is, moet je natuurlijk zelf bepalen. Ben je niet geïnteresseerd in de historie van je prijzen, dan kan deze methode prima werken, en hoef je niet verder te lezen!

Oplossing 2: opslaan met historie

Wil je de prijshistorie vastleggen, dan het je een iets andere werkwijze nodig. Om te beginnen: in de tabel [tBrandstofprijzen] volstaat het om een datumveld te hebben en een Prijs veld. Voor een bruikbare indeling hoef ik eigenlijk alleen naar de afbeelding van de tabel uit de vorige case te verwijzen, want dat is exact de tabel die we nu ook gebruiken. Alleen dus met meer records. Een voorbeeldje is bijvoorbeeld deze tabel:

PrijsID	BrandstofType	Wijzigingsdatum	Prijs
0001	Euro Normaal	07-12-2012	€ 1,68.9
0002	Diesel	08-12-2012	€ 1,48.8
0003	Super	14-12-2012	€ 1,88.2
0004	Gas	11-12-2012	€ 0,97.4
0005	Euro Normaal	22-12-2012	€ 1,71.6
0007	Diesel	25-12-2012	€ 1,51.5
0008	Super	22-12-2012	€ 1,83.4
0009	Gas	18-12-2012	€ 0,89.7
0010	Euro Normaal	27-12-2012	€ 1,74.2
0011	Super	26-12-2012	€ 1,86.2
0012	Diesel	31-12-2012	€ 1,44.7
0013	Gas	27-12-2012	€ 0,97.3
0014	Euro Normaal	06-01-2013	€ 1,66.2
0015	Super	07-01-2013	€ 1,79.3
0016	Diesel	06-01-2013	€ 1,41.2
0017	Gas	09-01-2013	€ 1,01.3
0018	Euro Normaal	16-01-2013	€ 1,69.9
0019	Super	14-01-2013	€ 1,85.2
0020	Diesel	15-01-2013	€ 1,47.2
0021	Gas	17-01-2013	€ 0,99.3
(Nieuw)		18-01-2013	

Je ziet dat er inmiddels voor elke brandstof wel een aantal prijswijzigingen is geweest, en dat de tabel dus een stevig aantal records bevat. De meest logische gedachte is natuurlijk, dat als je het reiskosten formulier invult, dat dan de meest recente records worden gebruikt. Oftewel: op 18 januari zij de actieve prijzen als volgt:

- Euro normaal: € 1,69.9
- Super: € 1,85.2
- Diesel: € 1,47.2
- Gas: € 0,99.3

Dat betekent dus, dat we de keuzelijst cboBrandstofsoort van het formulier zodanig moeten aanpassen dat alleen deze waarden te kiezen zijn. Als dat mogelijk is, werkt het formulier eigenlijk hetzelfde als in de vorige situatie: we slaan netjes de prijs op bij het record, en dat verandert dan niet meer. Hoe vaak ook daarna de prijs nog wijzigt.

Omdat de hele situatie verder niet verandert, hoef ik alleen maar te laten zien hoe de query van de keuzelijst er nu uit komt te zien. En dat is als volgt:

```
SELECT DISTINCT PrijsID, Switch([BrandstofType]=1,"Euro Normaal",
[BrandstofType]=2,"Super",[BrandstofType]=3,"Diesel",
[BrandstofType]=4,"Gas") AS Brandstof, Prijs, Wijzigingsdatum FROM
tBrandstofprijs WHERE (PrijsID In (SELECT Last(PrijsID) AS LaatstePrijsID
FROM tBrandstofprijs GROUP BY BrandstofType;));
```

De query maakt, net als in het vorige voorbeeld, gebruik van de Switch voor de brandsoort. Wat nu het onderscheid maakt, is de WHERE voorwaarde. Zoals je in de tabel kunt zien, zijn er per brandsoort meerdere records. De opdracht is dus om de tabel zodanig te filteren dat van *elke brandstofsoort* het laatste record wordt opgezocht. Daarvoor wordt een subquery gebruikt, die groepeert op BrandstofType, en daarvan het laatste PrijsID pakt. Dit werkt alleen als er van elke brandstof minstens één record is, dus als je met een lege tabel begint, moet je wel voor elke brandstof één record aanmaken. Die fungeert dan als laatste, en wordt dan getoond in de query. In het voorbeeld wordt de

query dan als uitkomst vertaald naar:

Where PrijsID In(18;19;20;21)

Kortom: de subquery maakt een filter van (in dit geval) 4 records, en dat zijn de brandstofprijzen die in de keuzelijst te kiezen zijn.

De werking van het formulier (en de tabellen) is dus in principe identiek aan het eerste voorbeeld, alleen bewaar je nu de oude prijzen, en bouw je een historie op. Altijd leuk om daar later weer grafiekjes op los te laten.

Voordelen van deze methode:

- Redelijk makkelijk te implementeren
- Opbouwen van prijshistorie
- Behoorlijk betrouwbaar

Nadeel:

- Lastige query voor de keuzelijst
- Extra veld nodig voor de prijs
- Data makkelijk te wijzigen, en dus niet te locken

Deze oplossing is redelijk makkelijk te maken, en je legt de prijshistorie op de achtergrond ook nog vast.

Oplossing 3: Genormaliseerde tabel met twee datumvelden

De moeilijkste oplossingen maken geen gebruik van een extra (prijs)veld, en daarmee blijft de database dus goed genormaliseerd. Dat betekent, dat we de prijs moeten gaan bepalen op basis van het datumveld (of velden) in de tabel met prijzen. Die datums gaan we dan vergelijken met de declaratiedatum. Daarbij stuiten we echter al heel snel op een probleem. Kijk eens naar de datums in de volgende afbeelding waarop de prijzen van Euro Normaal zijn verhoogd. Ik pak er even twee uit: 06-01-2013 en 16-01-2013. Als je een declaratie invult op die twee datums, dan krijg je prima resultaat: € 1,66.2 en € 1,69.9 resp. Maar als de declaratiedatum 11-01-2013 is, heb je in Access een probleem.

In Excel zou zo'n klusje heel eenvoudig zijn, want je zoekt de datum dan simpel op met Vert.Zoeken. Deze functie zoekt 'fuzzy' naar een datum, en als die datum niet gevonden wordt, naar de eerstvolgende lagere datum. Oftewel: als je zoekt op 11-01-2013, dan vind je in de zoektabel 06-01-2013. En het resultaat dat je terugleest is dan de prijs die op die datum is vastgesteld. Simpel dus: Access zal ook wel zo'n functie hebben? Nee dus. Vandaar dat we een moeilijke oplossing moeten gaan maken.

Maar ook moeilijke oplossingen komen in gradaties, dus ook hier: twee varianten. Daarbij is de eerste variant een stuk simpeler dan de tweede, dus daar moesten we maar mee beginnen!

De prijzentabel ziet er nu zo uit:

PrijsID	BrandstofType	BeginDatum	EindDatum	Prijs
0001	Euro Normaal	07-12-2012	23-12-12	€ 1,68.9
0002	Diesel	08-12-2012	24-12-12	€ 1,48.8
0003	Super	13-12-2012	21-12-12	€ 1,88.2
0004	Gas	11-12-2012	17-12-12	€ 0,97.4
0005	Euro Normaal	24-12-2012	26-12-12	€ 1,71.6
0007	Diesel	25-12-2012	30-12-12	€ 1,51.5
0008	Super	22-12-2012	25-12-12	€ 1,83.4
0009	Gas	18-12-2012	26-12-12	€ 0,89.7
0010	Euro Normaal	27-12-2012	05-01-13	€ 1,74.2
0011	Super	26-12-2012	06-01-13	€ 1,86.2
0012	Diesel	31-12-2012	05-01-13	€ 1,44.7
0013	Gas	27-12-2012	08-01-13	€ 0,97.3
0014	Euro Normaal	06-01-2013	15-01-13	€ 1,66.2
0015	Super	07-01-2013	13-01-13	€ 1,79.3
0016	Diesel	06-01-2013	14-01-13	€ 1,41.2
0017	Gas	09-01-2013	16-01-13	€ 1,01.3
0018	Euro Normaal	16-01-2013		€ 1,69.9
0019	Super	14-01-2013		€ 1,85.2
0020	Diesel	15-01-2013		€ 1,47.2
0021	Gas	17-01-2013		€ 0,99.3
(Nieuw)		18-01-2013		

Zoals je ziet, is er een veld bijgekomen: het veld *Einddatum*. Elk record met een prijs heeft nu dus een startdatum, en een datum waarop periode afloopt. Die einddatum ligt logischerwijs één dag voor de begindatum van de volgende periode. Zo zie je dat de periode die ik net al noemde op 06-01-2013 en op 15-01-2013 eindigt, en de volgende periode op 16-01-2013 begint en geen einddatum heeft. En dat is gelijk een indicatie van de actieve periode, want die is al wel begonnen, maar nog niet afgelopen. Er is derhalve nog geen einddatum. In de query gaat dat straks een klein probleem opleveren, maar gelukkig hebben we daar een simpele oplossing voor, zoals je kunt zien.

Het formulier dat we gebruiken om de records aan te maken is vergelijkbaar met dat van oplossing 2, dus daar besteden we nu geen aandacht aan. Het enige dat niet nodig is, is het opslaan van de feitelijke prijs van de brandstof. Die gaan we tenslotte dynamisch genereren. In de reiskosten tabel staan dan ook alleen deze gegevens:

TankID	Tankdatum	Plaats	Brandstofsoort	Liters
0001	12-12-2012	Utrecht	1	42,9
0002	20-12-2012	Rotterdam	3	62,0
0003	15-12-2012	Breda	2	52,3
0004	23-01-2013	Utrecht	1	73,0
0005	04-01-2013	Breda	4	54,0
0006	28-12-2012	Rotterdam	2	62,4
0007	08-01-2013	Utrecht	2	84,0
0008	18-12-2012	Groningen	1	94,0
0009	09-01-2013	Breda	1	52,0
0010	15-01-2013	Rotterdam	2	36,7
(Nieuw)	18-01-2013			

Voor de berekeningen (denk aan rapporten etc.) hebben we een query nodig, die de bovenstaande gegevens laat zien, en uit de tabel met de prijzen de juiste prijs ophaalt voor het betreffende brandstoftype. Daarvoor gebruiken we een *Cartesisch product*. We hebben die al eens eerder gemaakt, dus het principe ervan zal ik niet al te uitgebreid uitleggen. Waar het op neer komt: je pakt twee tabellen die (al dan niet) gemeenschappelijke velden hebben, en die tabellen koppel je niet aan elkaar. Als je de query nu uitvoert, worden alle records uit Tabel1 gekoppeld aan alle records uit Tabel2. In ons voorbeeld: 20 (Tabel Brandstof) X 10 (tabel Reiskosten) records = 210 records. Een

deel ervan zie je hier:

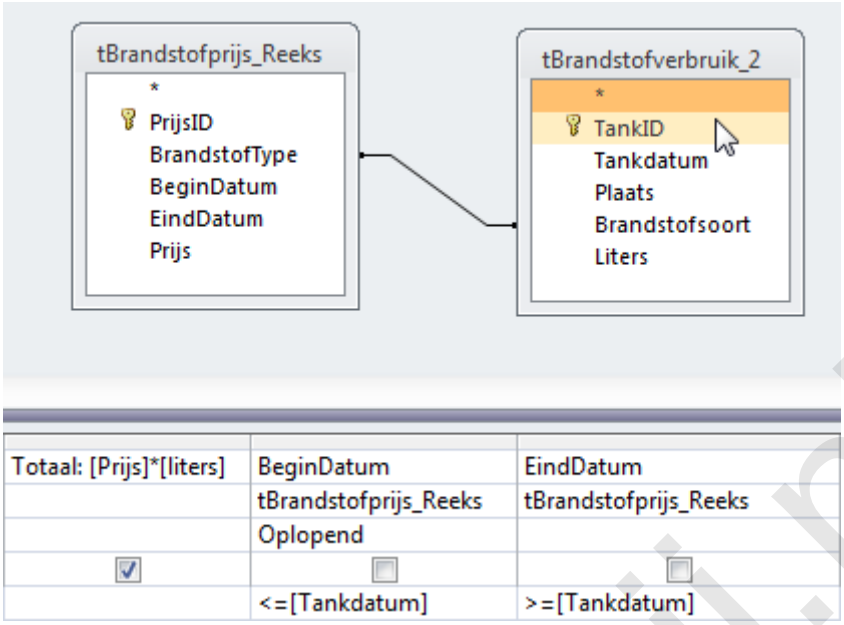
TankID	Tankdatum	BeginDatum	EindDatum	Plaats	BrandstofType	Liters	Prijs	Totaal
0001	12-12-2012	27-12-2012	05-01-13	Utrecht	Euro Normaal	42,9	€ 1,74.2	€ 74,73
0001	12-12-2012	13-12-2012	21-12-12	Utrecht	Super	42,9	€ 1,88.2	€ 80,74
0001	12-12-2012	27-12-2012	08-01-13	Utrecht	Gas	42,9	€ 0,97.3	€ 41,74
0001	12-12-2012	07-12-2012	23-12-12	Utrecht	Euro Normaal	42,9	€ 1,68.9	€ 72,46
0001	12-12-2012	24-12-2012	26-12-12	Utrecht	Euro Normaal	42,9	€ 1,71.6	€ 73,62
0001	12-12-2012	26-12-2012	06-01-13	Utrecht	Super	42,9	€ 1,86.2	€ 79,88
0001	12-12-2012	15-01-2013		Utrecht	Diesel	42,9	€ 1,47.2	€ 63,15
0001	12-12-2012	11-12-2012	17-12-12	Utrecht	Gas	42,9	€ 0,97.4	€ 41,78
0001	12-12-2012	06-01-2013	15-01-13	Utrecht	Euro Normaal	42,9	€ 1,66.2	€ 71,30
0001	12-12-2012	14-01-2013		Utrecht	Super	42,9	€ 1,85.2	€ 79,45
0001	12-12-2012	16-01-2013		Utrecht	Euro Normaal	42,9	€ 1,69.9	€ 72,89
0001	12-12-2012	08-12-2012	24-12-12	Utrecht	Diesel	42,9	€ 1,48.8	€ 63,84
0001	12-12-2012	07-01-2013	13-01-13	Utrecht	Super	42,9	€ 1,79.3	€ 76,92
0001	12-12-2012	18-12-2012	26-12-12	Utrecht	Gas	42,9	€ 0,89.7	€ 38,48
0001	12-12-2012	17-01-2013		Utrecht	Gas	42,9	€ 0,99.3	€ 42,60
0001	12-12-2012	25-12-2012	30-12-12	Utrecht	Diesel	42,9	€ 1,51.5	€ 64,99
0001	12-12-2012	31-12-2012	05-01-13	Utrecht	Diesel	42,9	€ 1,44.7	€ 62,08
0001	12-12-2012	09-01-2013	16-01-13	Utrecht	Gas	42,9	€ 1,01.3	€ 43,46
0001	12-12-2012	22-12-2012	25-12-12	Utrecht	Super	42,9	€ 1,83.4	€ 78,68
0001	12-12-2012	06-01-2013	14-01-13	Utrecht	Diesel	42,9	€ 1,41.2	€ 60,57
0003	15-12-2012	13-12-2012	21-12-12	Breda	Super	52,3	€ 1,88.2	€ 98,43
0003	15-12-2012	06-01-2013	14-01-13	Breda	Diesel	52,3	€ 1,41.2	€ 73,85
0003	15-12-2012	08-12-2012	24-12-12	Breda	Diesel	52,3	€ 1,48.8	€ 77,82
0003	15-12-2012	27-12-2012	08-01-13	Breda	Gas	52,3	€ 0,97.3	€ 50,89
0003	15-12-2012	06-01-2013	15-01-13	Breda	Euro Normaal	52,3	€ 1,66.2	€ 86,92

Op zichzelf is dit een onbruikbare query; TankID 1 zien we bijvoorbeeld 20 keer terug, met de daarbij horende gegevens. Aan de velden BeginDatum en EindDatum kun je al een beetje zien wat de bedoeling is. Die velden komen uit [tBrandstof], en geven de verschillende periodes aan waarin een prijs geldt. Het veld [Brandstoftype] komt ook uit [tBrandstof]. Omdat elke tankbeurt maar één product bevat, kunnen we de lijst al een heel stuk inkorten door de velden [tBrandstof].[Brandstoftype] en [tBrandstof].[Brandstofsoort] aan elkaar te koppelen. Daarmee wordt de lijst al een stuk zinniger:

TankID	Tankdatum	BeginDatum	EindDatum	Plaats	BrandstofType	Liters	Prijs	Totaal
0001	12-12-2012	07-12-2012	23-12-12	Utrecht	Euro Normaal	42,9	€ 1,68.9	€ 72,46
0001	12-12-2012	24-12-2012	28-12-12	Utrecht	Euro Normaal	42,9	€ 1,71.6	€ 73,62
0001	12-12-2012	29-12-2012	05-01-13	Utrecht	Euro Normaal	42,9	€ 1,74.2	€ 74,73
0001	12-12-2012	06-01-2013	15-01-13	Utrecht	Euro Normaal	42,9	€ 1,66.2	€ 71,30
0001	12-12-2012	16-01-2013		Utrecht	Euro Normaal	42,9	€ 1,69.9	€ 72,89
0002	20-12-2012	08-12-2012	24-12-12	Rotterdan	Diesel	62,0	€ 1,48.8	€ 92,26
0002	20-12-2012	25-12-2012	30-12-12	Rotterdan	Diesel	62,0	€ 1,51.5	€ 93,93
0002	20-12-2012	31-12-2012	05-01-13	Rotterdan	Diesel	62,0	€ 1,44.7	€ 89,71
0002	20-12-2012	06-01-2013	14-01-13	Rotterdan	Diesel	62,0	€ 1,41.2	€ 87,54
0002	20-12-2012	15-01-2013		Rotterdan	Diesel	62,0	€ 1,47.2	€ 91,26
0003	15-12-2012	13-12-2012	21-12-12	Breda	Super	52,3	€ 1,88.2	€ 98,43
0003	15-12-2012	22-12-2012	25-12-12	Breda	Super	52,3	€ 1,83.4	€ 95,92
0003	15-12-2012	26-12-2012	06-01-13	Breda	Super	52,3	€ 1,86.2	€ 97,38
0003	15-12-2012	07-01-2013	13-01-13	Breda	Super	52,3	€ 1,79.3	€ 93,77
0003	15-12-2012	14-01-2013		Breda	Super	52,3	€ 1,85.2	€ 96,86
0004	26-12-2012	07-12-2012	23-12-12	Utrecht	Euro Normaal	73,0	€ 1,68.9	€ 123,30
0004	26-12-2012	24-12-2012	28-12-12	Utrecht	Euro Normaal	73,0	€ 1,71.6	€ 125,27
0004	26-12-2012	29-12-2012	05-01-13	Utrecht	Euro Normaal	73,0	€ 1,74.2	€ 127,17
0004	26-12-2012	06-01-2013	15-01-13	Utrecht	Euro Normaal	73,0	€ 1,66.2	€ 121,33
0004	26-12-2012	16-01-2013		Utrecht	Euro Normaal	73,0	€ 1,69.9	€ 124,03

Er zitten nog steeds herhalingen in de recordset. Maar voor elke tankbeurt zijn er nu nog maar 5 records i.p.v. 20. Je kunt nu al wel zien welke records we moeten hebben, en welke de verkeerde prijs laten zien. Kijken we bijvoorbeeld naar TankID 0004, dan zien we dat tankdatum (26-12-2012) in de eerste record de verkeerde prijsrecord koppelt. De datumrange loopt namelijk van7-12 t/m 23-12. De tweede record is wel goed (26-12 ligt in de reeks 24-12 t/m 28-12) en de derde en vierde record zijn weer fout: de begindatums liggen al voorbij de tankdatum. Op basis van deze tabel kunnen we al bepalen hoe de query gefilterd moet worden: de tankdatum moet groter (of gelijk) zijn aan de begindatum, en kleiner (of gelijk) aan de einddatum. Zoals gezegd: het tweede record voldoet aan dit criterium.

Gewapend met deze kennis kunnen we de query gaan maken. Dat doen we natuurlijk in het queryraster. En dat ziet er dan zo uit:



Alle belangrijke verbindingen zijn in deze afbeelding ondergebracht: de koppeling op basis van het veld [Brandstofsoort], en de velden BeginDatum en EindDatum met hun respectievelijke criteria.

Het resultaat is nu:

TankID	Tankdatum	Plaats	BrandstofType	Liters	Prijs	Totaal
0001	12-12-2012	Utrecht	Euro Normaal	42,9	€ 1,68.9	€ 72,46
0002	20-12-2012	Rotterdam	Diesel	62,0	€ 1,48.8	€ 92,26
0003	15-12-2012	Breda	Super	52,3	€ 1,88.2	€ 98,43
0004	26-12-2012	Utrecht	Euro Normaal	73,0	€ 1,71.6	€ 125,27
0005	04-01-2013	Breda	Gas	54,0	€ 0,97.3	€ 52,54
0006	28-12-2012	Rotterdam	Super	62,4	€ 1,86.2	€ 116,19
0007	08-01-2013	Utrecht	Super	84,0	€ 1,79.3	€ 150,61
0008	18-12-2012	Groningen	Euro Normaal	94,0	€ 1,68.9	€ 158,77
0009	09-01-2013	Breda	Euro Normaal	52,0	€ 1,66.2	€ 86,42

Zoals je kunt zien, wordt bij elke tankbeurt de juiste prijs opgezocht in de tabel. En op basis van de prijs en het aantal liters, is het nu simpel om de totaalprijs uit te rekenen.

Voordelen van deze methode:

- Flexibele opslag en historie van prijzen
- Prijzen gefixeerd, en niet te veranderen en dus veilig
- Database is nog steeds redelijk genormaliseerd

Nadeel:

- Moeilijkheidsgraad ligt al redelijk hoog
- Lastige query om de prijs uit te rekenen
- Extra veld (Einddatum) nodig voor de prijs
- Het extra datumveld vergroot de kans op verkeerde waarden in de tabel, want er is geen relatie tussen de einddatum en de startdatum van de volgende periode

Deze methode is een stuk lastiger te maken, omdat je goed moet nadenken hoe de subquery er uit moet zien. In het voorbeeld is een wat uitgebreider criterium opgenomen (op brandstoftype). Heb je één artikel, dan hoeft dat natuurlijk niet en wordt de query wat simpeler. Je kunt hem a.d.h.v. het

voorbeeld hopelijk nu ook uitbreiden met meer voorwaarden; de structuur daarvoor blijft hetzelfde.

Oplossing 4: genormaliseerde prijsberekening

De lastigste oplossing bewaren we uiteraard voor het laatst. Deze oplossing bouwt voort op oplossing 3, maar nu op basis van een tabel met maar één datumveld voor de prijswijzigingen. Dat maakt het invoeren van prijswijzigingen een heel stuk simpeler, want er is nu maar één mutatiedatum.

PrijsID	BrandstofType	Wijzigingsdatum	Prijs
0001	Euro Normaal	07-12-2012	€ 1,68.9
0002	Diesel	08-12-2012	€ 1,48.8
0003	Super	13-12-2012	€ 1,88.2
0004	Gas	11-12-2012	€ 0,97.4
0005	Euro Normaal	24-12-2012	€ 1,71.6
0007	Diesel	25-12-2012	€ 1,51.5
0008	Super	22-12-2012	€ 1,83.4
0009	Gas	18-12-2012	€ 0,89.7
0010	Euro Normaal	27-12-2012	€ 1,74.2
0011	Super	26-12-2012	€ 1,86.2
0012	Diesel	31-12-2012	€ 1,44.7
0013	Gas	27-12-2012	€ 0,97.3
0014	Euro Normaal	06-01-2013	€ 1,66.2
0015	Super	07-01-2013	€ 1,79.3
0016	Diesel	06-01-2013	€ 1,41.2
0017	Gas	09-01-2013	€ 1,01.3
0018	Euro Normaal	16-01-2013	€ 1,69.9
0019	Super	14-01-2013	€ 1,85.2
0020	Diesel	15-01-2013	€ 1,47.2
0021	Gas	17-01-2013	€ 0,99.3
(Nieuw)		19-01-2013	

Probleem is nu wel: hoe kun je op basis van één datum kijken of een tankdatum in een bepaalde periode valt? Je hebt nu immers alleen de startdatum van de periode, en niet de einddatum. Wel, die zal op de een of andere manier berekend moeten worden! En dat kan door een hulptabel in te voeren d.m.v. een query. Die query levert als eindresultaat straks een tabel op die er precies zo uitziet als de prijstabel die we in oplossing 3 hebben gebruikt. En dat is ook logisch, want we hebben per prijsmutatie een begindatum en een einddatum nodig. We kunnen dat bereiken door een subquery te gebruiken als basis voor een extra veld. Tot nu toe hebben we subqueries gebruikt om recordsets te filteren (ook weer in dit hoofdstuk). Maar dus nog niet om een nieuw veld te maken. We concentreren ons nu dus op die specifieke techniek.

```
SELECT PrijsID, BrandstofType, Prijs, Wijzigingsdatum AS BeginDatum, (SELECT
MIN(tmpBrandstof.Wijzigingsdatum) FROM tBrandstofprijs AS tmpBrandstof WHERE
tmpBrandstof.Wijzigingsdatum > tBrandstofprijs.Wijzigingsdatum AND tmpBrandstof.BrandstofType =
tBrandstofprijs.BrandstofType)-1 AS VolgendeDatum FROM tBrandstofprijs;
```

Zoals je gewend bent in een query, benoem je na SELECT de velden in de query. De subquery is er daar één van. Het stuk code waar het om gaat is dus:

```
(SELECT MIN(tmpBrandstof.Wijzigingsdatum)
FROM tBrandstofprijs AS tmpBrandstof
WHERE tmpBrandstof.Wijzigingsdatum > tBrandstofprijs.Wijzigingsdatum
AND tmpBrandstof.BrandstofType = tBrandstofprijs.BrandstofType)-1
AS VolgendeDatum
```

Wat gebeurt hier nu eigenlijk? We maken een query die de kleinste waarde van het veld Wijzigingsdatum opzoekt in de tabel tBrandstofprijs, die ook nog eens met zichzelf vergeleken wordt. Oftewel: als je deze query als zelfstandige query zou maken, dan moet je de tabel tBrandstofprijs twee keer toevoegen aan je raster. De twee tabellen worden vervolgens niet aan elkaar gekoppeld (geen INNER JOIN), maar zoals je in de code ziet, wordt er wel gefilterd op basis van het BrandstofType. Want we willen natuurlijk datums zien voor de gebruikte brandstofsoort.

Het datumveld wordt verder gefilterd op het datumveld uit tBrandstofprijs, dat als criterium heeft dat er records moeten worden getoond die groter zijn dan de datum in de andere tabel. Dat zijn er uiteraard een hele hoop, als je meerdere records hebt. Daarom wordt met MIN de laagste uitkomst gefilterd. Kortom: we hebben een query gemaakt die alle datums laat zien die groter zijn dan de startdatum, en daar pakken we de kleinste waarde van. En de kleinste is natuurlijk maar één waarde, en dat is uiteindelijk het resultaat dat in de query wordt getoond. Een deel van het resultaat zie je hier:

PrijsID	BrandstofType	Prijs	BeginDatum	VolgendeDatum
0001	Euro Normaal	€ 1,68.9	07-12-2012	23-12-12
0002	Diesel	€ 1,48.8	08-12-2012	24-12-12
0003	Super	€ 1,88.2	13-12-2012	21-12-12
0004	Gas	€ 0,97.4	11-12-2012	17-12-12
0005	Euro Normaal	€ 1,71.6	24-12-2012	26-12-12
0007	Diesel	€ 1,51.5	25-12-2012	30-12-12
0008	Super	€ 1,83.4	22-12-2012	25-12-12
0009	Gas	€ 0,89.7	18-12-2012	26-12-12
0010	Euro Normaal	€ 1,74.2	27-12-2012	05-01-13
0011	Super	€ 1,86.2	26-12-2012	06-01-13
0012	Diesel	€ 1,44.7	31-12-2012	05-01-13
0013	Gas	€ 0,97.3	27-12-2012	08-01-13
0014	Euro Normaal	€ 1,66.2	06-01-2013	15-01-13
0015	Super	€ 1,79.3	07-01-2013	13-01-13
0016	Diesel	€ 1,41.2	06-01-2013	14-01-13
0017	Gas	€ 1,01.3	09-01-2013	16-01-13
0018	Euro Normaal	€ 1,69.9	16-01-2013	
0019	Super	€ 1,85.2	14-01-2013	
0020	Diesel	€ 1,47.2	15-01-2013	
0021	Gas	€ 0,99.3	17-01-2013	

Je ziet, dat voor alle mutatiedatums een extra datum is gevonden, namelijk de volgende mutatiedatum voor dat brandstoftype. En, ook niet onbelangrijk, er is nog steeds geen einddatum voor de meest recente wijziging. Omdat we nu met een query werken, en niet met een tabel, is dat nog wel op te lossen met een extra IIF op het veld. De code ziet er nu zo uit:

```
SELECT PrijsID, BrandstofType, Prijs, Wijzigingsdatum AS BeginDatum, IIf((SELECT
MIN(tmpBrandstof.Wijzigingsdatum) FROM tBrandstofprijs AS tmpBrandstof WHERE
tmpBrandstof.Wijzigingsdatum > tBrandstofprijs.Wijzigingsdatum AND tmpBrandstof.BrandstofType =
tBrandstofprijs.BrandstofType) Is Null,Date(),(SELECT MIN(tmpBrandstof.Wijzigingsdatum) FROM
tBrandstofprijs AS tmpBrandstof WHERE tmpBrandstof.Wijzigingsdatum >
tBrandstofprijs.Wijzigingsdatum AND tmpBrandstof.BrandstofType = tBrandstofprijs.BrandstofType)-1)
AS VolgendeDatum FROM tBrandstofprijs;
```

Het resultaat wordt dan:

0014	Euro Normaal	€ 1,66.2	06-01-2013	15-01-13
0015	Super	€ 1,79.3	07-01-2013	13-01-13
0016	Diesel	€ 1,41.2	06-01-2013	14-01-13
0017	Gas	€ 1,01.3	09-01-2013	16-01-13
0018	Euro Normaal	€ 1,69.9	16-01-2013	19-01-13
0019	Super	€ 1,85.2	14-01-2013	19-01-13
0020	Diesel	€ 1,47.2	15-01-2013	19-01-13
0021	Gas	€ 0,99.3	17-01-2013	19-01-13

In deze afbeelding zie je alleen de laatste records, want daarin wordt de variabele datum (Date())

uitgerekend.

Voordelen van deze methode:

- Flexibele opslag en historie van prijzen
- Slechts één datumveld nodig voor de mutaties
- Prijzen gefixeerd, en niet te veranderen en dus veilig
- Database is nog steeds goed genormaliseerd

Nadeel:

- Moeilijkheidsgraad ligt behoorlijk hoog
- Lastige query om de prijs uit te rekenen
- Het extra datumveld vergroot de kans op verkeerde waarden in de tabel

Wat voor oplossing 3 geldt, is bij deze oplossing ook waar, en wel dat hij best lastig is. Je zult dus zelf moeten afwegen of de flexibiliteit en gegevensstabiliteit die je ervoor terugkrijgt opwegen tegen de tijd die je moet investeren om hem te doorgronden en toe te passen in je eigen situatie.

Samenvatting

We hebben in dit hoofdstuk een lastig onderwerp behandeld, dat toch erg belangrijk is voor de juiste werking van de databases die we maken en dat is de vraag: hoe gaan we om met gegevens die regelmatig veranderen? Ik heb daarvoor 4 methodes aangereikt die elk hun eigen voordelen en nadelen hebben. Er zijn vast meer oplossingen denkbaar en in omloop; heb je zelf een oplossing die je wilt delen, post die dan in het forumdraadje dat bij deze cursus hoort:

http://www.helpmij.nl/forum/showthread.php/591915-Vragen-Opmerken-Reacties-en-Tips-voor-de-Access-cursus

Daar vind je ook de uitwerking van de verschillende opgaves.

Volgende Aflevering

In de volgende aflevering gaan we verder met het principe van een Frontend-Backend database. We maken dan in een gesplitste database een frontend die middels een losse connectie de gegevens inlaadt.