Universiteit Gent - TELIN - DDCM
2019-2020

Workshop: Introductie PostgreSQL & Fysiek ontwerp

1 Introductie

Tijdens de eerste oefeningenlessen heb je geleerd hoe je vanuit een bepaald probleemdomein een EER-diagram kan ontwerpen. Vervolgens hebben we dit EER-diagram gemapt op een logisch databankontwerp specifiek voor het relationele databankmodel. Dit werd ingeoefend in het kader van het ontwerpen van een databank voor het autoverhuurbedrijf 'Rolls Robin', waarna je zelf aan de slag ging met de projectopgave. Met enkel maar een logisch databankontwerp kan je echter in de praktijk nog niet veel doen. Finaal heb je immers een fysieke, praktisch bruikbare databank nodig waarin je data kan opslaan, waarmee je analyses kan doen op de data,. . .

In deze workshop gaan we daarom dieper in op hoe je een logisch databankontwerp kan omzetten in een echt bruikbare, fysieke databank. We starten eerst en vooral met een overzicht van de manieren waarmee je met een relationele databank kan communiceren. Daarna introduceren we typische concepten van een relationele databank en bouwen we een fysiek ontwerp voor de Rolls Robin databank, waarna je zelf het fysiek ontwerp voor het project zal uitwerken. Om jullie te helpen bij het doorlopen van deze workshop, vinden jullie achteraan in de Appendix het relationeel databankschema voor de Rolls Robin databank dat we in de vorige lessen opbouwden.

Vooraleer we starten willen we ook wijzen op het feit dat deze workshop te lang zal zijn om in 1 lesuur te voltooien. Daarom wordt het zeer sterk aangeraden om een kwartiertje voor het einde van het lesuur Sectie 6 te lezen, zodat je jouw werk in deze workshop zeker niet kwijtspeelt.

2 PostgreSQL

Zoals eerder al gezegd, focussen we doorheen dit semester op het relationeel databankmodel. Aangezien dit model enorm populair is, bestaan er heel veel verschillende commerciële implementaties van relationele databankbeheersystemen. Dergelijke programma's implementeren alle concepten van het relationele model (zoals tabellen, primaire sleutels, vreemde sleutels,. . .) en laten dus toe fysieke databanken op te zetten. In dit opleidingsonderdeel hebben we gekozen voor een van de meest gekende, en bovendien een open-source en gratis relationeel databankbeheersysteem: PostgreSQL. Alhoewel dit natuurlijk slechts 1 van de vele relationele databankbeheersystemen is, kan je veel van de aangeleerde concepten ook rechtstreeks toepassen op andere systemen zoals MySQL, Oracle, SQL Server,. . . Om een relationele databank aan te maken, moeten we eerst en vooral het PostgreSQL databankbeheersysteem downloaden en installeren. Om dit correct te doen, hebben we twee handleidingen voorzien: een voor de installatie van PostgreSQL op je eigen PC en een voor het gebruik van PostgreSQL op een labo-PC. Beide handleidingen zijn terug te vinden op Ufora, onder de module 'Workshops'. Lees de handleiding die voor jou van toepassing is grondig door. Pas als je alle stappen ervan hebt doorlopen, kan je doorgaan met het vervolg van deze workshop.

3 Communiceren met een databank

Wanneer je de handleiding hebt doorlopen, zou PostgreSQL op een correcte manier geïnstalleerd moeten zijn en zou bovendien een lokale PostgreSQL cluster moeten draaien. Bij het opstarten van PostgreSQL opent er geen grafisch gebruikersscherm waar je allerlei dingen mee kan doen, zoals je misschien typisch gewoon bent van een programma. PostgreSQL is echter een programma dat in de achtergrond draait op je PC. Bij de installatie ervan wordt standaard 1 databank aangemaakt, met de naam `postgres`. Natuurlijk willen we ook andere databanken, zoals bv. `rollsrobin`, kunnen aanmaken en vervolgens kunnen gebruiken. Daartoe moeten we kunnen verbinden met de PostgreSQL service waarin we de databank willen aanmaken. In dit geval is dat de lokale PostgreSQL cluster die op je eigen PC of op de labo-PC draait, maar het is belangrijk om te weten dat je vanop jouw PC ook kan verbinden met PostgreSQL clusters die op andere computers of servers draaien, indien je hier toegang toe hebt.
Eens verbonden kunnen we dan communiceren met dit databankbeheersysteem door instructies op te stellen die het dbms moet uitvoeren. Een van de grote voordelen van het relationele model is dat er een gestandaardiseerde taal werd ontwikkeld om met relationele databankbeheersystemen te communiceren, genaamd *Structured Query Language* of SQL. De diverse commerciële implementaties gebruiken elk een eigen SQL-dialect, maar de communicatieconcepten die je doorheen de ko-

mende workshops zal leren zijn dus ook grotendeels bruikbaar in het geval van andere relationele implementaties.
Er bestaan heel wat verschillende manieren om SQL-instructies door te geven aan PostgreSQL. In deze sectie zullen we twee manieren aan jullie voorstellen, zijnde psql (een commandolijn-applicatie) en pgAdmin, versie 4 (een applicatie met grafische interface). We zullen voor beide communicatietools tonen hoe we een databank kunnen aanmaken en nadien opnieuw verwijderen.

3.1 Commandolijn

Een eerste manier om met een PostgreSQL databank te communiceren is door rechtstreeks te werken met het psql-programma dat samen komt met de installatie van PostgreSQL. Je kan met dit programma werken via de commandolijn. Dit open je in Windows door op de startknop te klikken, vervolgens '`cmd`' in te typen en op het programma dat verschijnt te klikken. Mac- en Linux-gebruikers openen hun 'Terminal'. Vanuit de commandolijn die je net hebt geopend kan je allerlei programma's uitvoeren die op je computer geïnstalleerd staan, waaronder dus ook psql. Voer om te beginnen volgend commando uit in de commandolijn:

```
psql -h localhost -U postgres -d postgres
```

Door dit commando uit te voeren, verbind je via psql met de lokaal draaiende PostgreSQL cluster (op IP-adres 127.0.0.1, alias localhost). De daaropvolgende -U-optie zorgt ervoor dat je inlogt als gebruiker met gebruikersnaam 'postgres'. De databank waarmee je verbonden bent wordt meegegeven achter de -d-optie en is deze die standaard wordt aangemaakt wanneer je PostgreSQL installeert met de naam `postgres`. Als je dit commando uitvoert, wordt er gevraagd om een wachtwoord. Dit is het wachtwoord van de gebruiker met naam 'postgres' dat je hebt ingesteld bij de installatie. Op een labo-PC is dit wachtwoord opnieuw 'postgres'.
Eens je succesvol geconnecteerd bent kan je het commando `\l` (waarbij de 'l' voor 'list' staat) uitvoeren, waarmee je een overzicht krijgt van alle PostgreSQL databanken die op dit moment draaien op jouw machine. We willen nu echter een nieuwe databank aanmaken met de naam `rollsrobin`. Dit doen we door dit als een instructie door te geven aan het databankbeheersysteem. De PostgreSQL instructie om een databank aan te maken is de volgende:

```
CREATE DATABASE rollsrobin;
```

Voer deze instructie uit en controleer opnieuw welke databanken draaien op je systeem. Als alles goed is gegaan, zou de `rollsrobin` databank er nu moeten tussen staan. Nu de databank is aangemaakt, zouden we kunnen beginnen met het vertalen van het logische databankontwerp naar een fysieke implementatie. Hiervoor moet je eerst connectie maken met de juist aangemaakte `rollsrobin` databank. Dit kan eenvoudigweg door middel van volgende instructie:

```
\c rollsrobin
```

waarbij ‘c’ staat voor ‘connect’ We tonen echter eerst hoe je een databank kan verwijderen, en zullen vervolgens met pgAdmin 4 de databank opnieuw aanmaken. Een databank kan als volgt worden verwijderd:

```
DROP DATABASE rollsrobin;
```

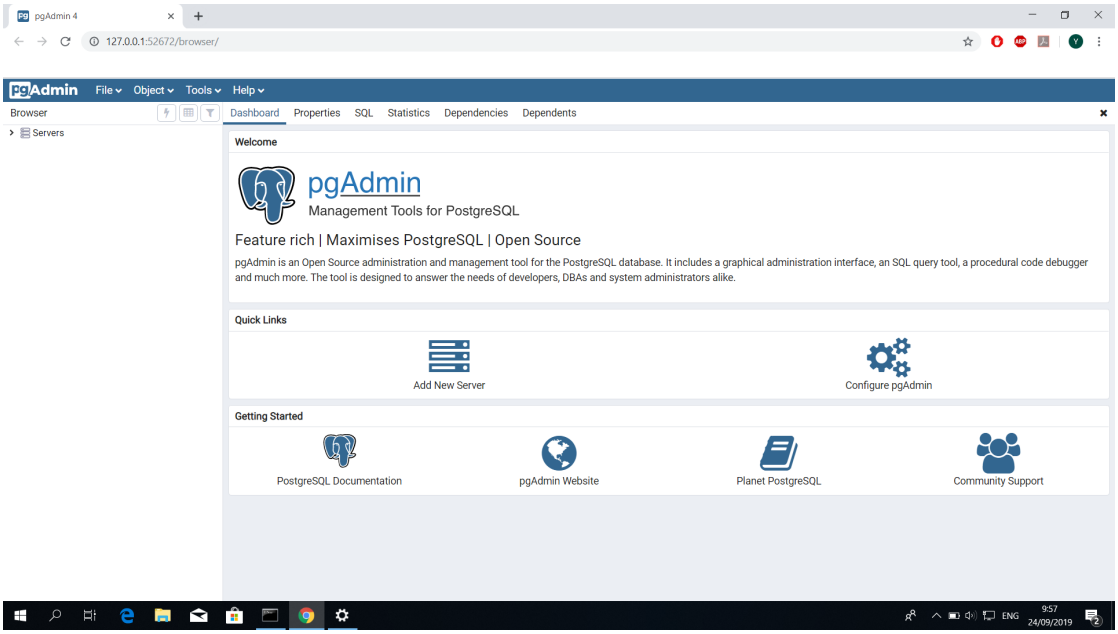
Voer dit commando uit vooraleer verder te gaan met de volgende opdrachten[1]. Let wel op, je kan geen databank verwijderen indien er nog connecties bestaan naar deze databank. Connecteer dus terug eerst met de `postgres` databank en verwijder alle openstaande connecties naar de `rollsrobin` databank. Voor de volledige documentatie van psql kan je steeds terecht op `https://www.postgresql.org/docs/current/app-psql.html`.

Voer bovenstaande opdrachten uit en experimenteer wat met psql vooraleer verder te gaan.

3.2 pgAdmin 4

Zoals eerder al aangekondigd, is een van de tools die gebruikt kan worden om te werken met een PostgreSQL databank pgAdmin 4. Net zoals met psql, kan je via pgAdmin 4 verbinden en communiceren zowel met databanken die op jouw lokale machine draaien als met databanken die op een andere machine te vinden zijn. Bovendien heeft de tool, in tegenstelling tot psql, een grafische interface waardoor je heel intuïtief de gewenste databank of tabel kan selecteren, aanmaken, verwijderen,... Daarnaast biedt pgAdmin, net als psql, echter ook een interface aan om directe instructies (zoals `CREATE DATABASE`) naar een verbonden PostgreSQL cluster te sturen. In pgAdmin 4 bestaan er met andere woorden twee manieren om met een PostgreSQL databank te interageren. We tonen hier beide.
Na het opstarten van pgAdmin 4 opent er een webbrowser waarin de applicatie je begroet met het beginscherm zoals afgebeeld in Figuur 1. Dit scherm is opgedeeld in 2 grote delen: een browservenster links en een detailgedeelte rechts, waar je informatie krijgt over het deel dat links geselecteerd is en waar je interessante acties kan uitvoeren.
Net zoals in psql, zullen we eerst een connectie moeten maken met onze lokaal draaiende PostgreSQL cluster (localhost, poort 5432), vooraleer we een nieuwe databank `rollsrobin` kunnen aanmaken. In het linkervenster heb je een overzicht van

[1] Sluit steeds al je PostgreSQL-specifieke commando’s af met een puntkomma!



Figuur 1: pgAdmin 4 beginscherm.

al je huidige connecties. Een connectie aanmaken met je lokale PostgreSQL cluster doe je zoals beschreven in onderstaande opdracht.

Maak een connectie met ‘localhost’ door in het browservenster met de rechtermuisknop te klikken op Servers - Create - Server. . .
Geef de aangemaakte connectie de naam ‘local’. Standaard draait een PostgreSQL databank op het localhost adres, poort 5432 en is de default user met de naam ‘postgres’ aangemaakt zonder wachtwoord, tenzij je dit bij de installatie hebt gespecificeerd. Voor labo-PC gebruikers is het wachtwoord opnieuw ‘postgres’.

Je kan de verbinding ‘local’ nu openklikken. Onder ‘Databases’ vind je alle databanken die op deze connectie (machine + poort) draaien. Aangezien we daarnet in psql de aangemaakte databank `rollsrobin` opnieuw verwijderd hebben, zie je deze niet meer in het overzicht staan.

Maak de databank `rollsrobin` aan via de grafische interface door rechts te klikken op Databases - Create - Database. . . en vervolgens de naam van de

databank in te vullen.

Normaal gezien staat de databank nu zichtbaar tussen je andere lokale databanken[2]. We zullen de databank echter ook eens aanmaken via het geven van directe instructies in pgAdmin.

Verwijder eerst de databank `rollsrobin`, door er rechts op te klikken en Delete/Drop te selecteren.

In pgAdmin 4 kan je dus ook SQL-instructies doorgeven aan een databank, net zoals we dit eerder deden in psql. Dit kan via de 'query tool' die je opent door rechts te klikken op de **naam van de databank** in het browservenster, en dan deze query tool te selecteren.

Maak de `rollsrobin` databank aan door middel van het reeds geziene commando en klik vervolgens op de bliksemschicht bovenaan om de instructie uit te voeren.

We hebben nu op drie verschillende manieren de `rollsrobin` databank aangemaakt. In het vervolg zullen we gebruik maken van pgAdmin 4 vanwege de grafische ondersteuning, maar zullen we wel via directe instructies ('SQL-query's') blijven werken. Het is immers van cruciaal belang om de SQL taal goed onder de knie te krijgen, zodat je niet afhankelijk bent van de grafische interface van pgAdmin om met een (PostgreSQL) databankbeheersysteem te kunnen communiceren. Het is dus noodzakelijk om met alle soorten tools een databank te kunnen opzetten.

4 Basiscomponenten

In Sectie 3 hebben we op 3 verschillende manieren de lege `rollsrobin` databank aangemaakt. Dit is uiteraard niet voldoende. De tabellen, primaire sleutels, vreemde sleutels en alle andere beperkingen die we in de logische ontwerpsfase hebben bekomen, moeten we immers op een of andere manier ook in de echte databank kunnen implementeren, zodanig dat we vervolgens effectief data aan deze databank

[2] Mogelijks moet je de pagina eens verversen vooraleer nieuw aangemaakte componenten worden weergegeven. Dit kan door met je muis rechts op een component in het browservenster 'Refresh...' te selecteren.

kunnen toevoegen en deze data op een nuttige manier kunnen gebruiken. Daarom gaan we in deze sectie dieper in op elk van de basiscomponenten van het relationele model en hoe deze in PostgreSQL geïmplementeerd kunnen worden. Wat betreft de beperkingen focussen we enkel op basisconcepten. Meer geavanceerde concepten komen aan bod in de workshop 'Functies, triggers & stored procedures'. Meer gedetailleerde informatie in verband met beperkingen in PostgreSQL kan je terugvinden op `https://www.postgresql.org/docs/current/ddl-constraints.html`.

4.1 Basisrelaties

Zoals je ondertussen wel weet, vormt de basisrelatie (of tabel) hét fundament van het relationele model. In het logische databankontwerp voor Rolls Robin hebben we 9 basisrelaties geïdentificeerd: contract, filiaal, locatie, persoon, registratieformulier, wagen, constructeur, type en werknemer. Deze moeten allemaal in de fysieke PostgreSQL databank worden geïmplementeerd. Het aanmaken van een tabel gebeurt in PostgreSQL met behulp van het `CREATE TABLE` statement[3]. Binnen dit statement definieer je de verschillende attributen van de tabel en hun overeenkomstige datatype. Als voorbeeld vind je hieronder het statement waarmee we een fysieke implementatie van de basisrelatie `type` maken. De basisrelatie heeft twee geassocieerde attributen: een attribuut `naam` van het varchar (tekst) datatype en een attribuut `dagtarief` van het numeric datatype. Dit is volledig conform de basisrelatie die werd geïdentificeerd in het logisch ontwerp. Het SQL-statement voor de aanmaak van deze tabel is

```
CREATE TABLE type
(
    naam varchar,
    dagtarief numeric
);
```

Na het uitvoeren van dit statement zien jullie normaal gezien in het browservenster van pgAdmin de tabel `type` verschijnen [4].

4.2 Primaire sleutels

Bij de aanmaak van de tabel `type` zijn we een essentiële component van het relationeel model vergeten: de primaire sleutel! Deze combinatie van attributen is per definitie uniek voor elk record dat in de tabel wordt opgeslagen. Bovendien dient deze verzameling attributen ook irreducibel te zijn, in die zin dat wanneer uit de

[3] `https://www.postgresql.org/docs/current/sql-createtable.html`

[4] Een overzicht van alle tabellen vind je in het browservenster van pgAdmin 4 onder de naam van de databank, Schemas, de naam van het schema (bv. `public`), en dan onder Tables

verzameling een attribuut zou worden weggelaten, het niet meer gegarandeerd is dat iedere record uit de tabel een unieke combinatie van waarden heeft voor deze attributen.
In het logisch ontwerp identificeerden we `naam` als primaire sleutel voor de basisrelatie `type`. Om deze primaire sleutel toe te voegen aan de fysieke implementatie van de tabel, voeren we volgende instructie uit.

```
ALTER TABLE type ADD CONSTRAINT type_pkey
PRIMARY KEY (naam);
```

Zoals je kan zien past dit statement de tabel `type` aan door een primaire sleutelbeperking op het attribuut `naam` toe te voegen met de naam `type_pkey`.
Alhoewel we hierboven de aanmaak van een tabel en het toevoegen van een primaire sleutel aan deze tabel hebben uitgevoerd door middel van twee verschillende statements, kan je eigenlijk de primaire sleutel ook direct aan een tabel toevoegen bij de aanmaak ervan. Aangezien we de tabel `type` en haar bijhorende primaire sleutel hierboven al aanmaakten, moeten we deze eerst verwijderen vooraleer we de tabel opnieuw kunnen aanmaken. Een tabel verwijderen doe je als volgt.

```
DROP TABLE type;
```

Vervolgens maken we de tabel `type` en haar primaire sleutel opnieuw aan, ditmaal gebruik makend van slechts één enkel statement:

```
CREATE TABLE type
(
    naam varchar,
    dagtarief numeric,

    CONSTRAINT type_pkey PRIMARY KEY (naam)
);
```

Merk op dat het resultaat van bovenstaande instructie identiek is aan de tabel die we verkregen door de tabel en de primaire sleutel met meerdere statements aan te maken.
Een laatste manier om de tabel `type` aan te maken is door middel van de volgende verkorte notatie.

```
CREATE TABLE type
(
    naam varchar PRIMARY KEY,
    dagtarief numeric
);
```

Dit statement kan je enkel en alleen gebruiken als de primaire sleutel uit 1 attribuut bestaat.

4.3 Vreemde sleutels

Naast basisrelaties en primaire sleutels omvat het relationeel model een derde cruciale component, namelijk de vreemde sleutel. Dit stelt een verwijzing voor van een set aan attributen in een basisrelatie naar een set van attributen in een (al dan niet andere) basisrelatie en wordt gebruikt om relaties tussen data in verschillende tabellen weer te geven. Deze beperking dwingt af dat elke combinatie van attribuutwaarden die wordt toegevoegd onder de desbetreffende attributen van de refererende basisrelatie, ook aanwezig moet zijn in de gerefereerde basisrelatie. Bovendien moeten de attributen in de gerefereerde basisrelatie uniciteit afdwingen. Neem als voorbeeld de basisrelatie `constructeur`. Het attribuut `type` verwijst naar het attribuut `naam` in de basisrelatie `type`. In deze laatste basisrelatie dwingt dit attribuut inderdaad uniciteit af, aangezien `naam` de primaire sleutel vormt van de tabel `type`. Daarnaast mogen specifieke attribuutwaarden voor een record enkel voorkomen in de tabel `constructeur` als deze combinatie ook voorkomt in de gerefereerde basisrelatie `type`.
Stel bijvoorbeeld dat je een constructeur wil toevoegen aan de databank waarvoor het type wagen 'sportwagen' is, dan zorgt deze vreemde sleutel-beperking ervoor dat je de constructeur met deze typenaam enkel kan toevoegen als deze eerst al zijn toegevoegd in de tabel `type`.
Om dergelijk gedrag te bekomen, voorziet PostgreSQL functionaliteit voor het implementeren van vreemde sleutels. Met onderstaand statement wordt de tabel `constructeur` aangemaakt zoals voorgeschreven door het logisch ontwerp: drie verschillende attributen met corresponderend datatype, de atributenverzameling `(merk, model)` dat de primaire sleutel vormt, én tot slot het attribuut `type` dat een vreemde sleutel vormt die verwijst naar het attribuut `naam` in de tabel `type`.

```
CREATE TABLE constructeur
(
    merk varchar,
    model varchar,
    type varchar,

    CONSTRAINT constructeur_pkey PRIMARY KEY (merk, model),

    CONSTRAINT constructeur_type_fkey FOREIGN KEY (type)
    REFERENCES type (naam)
);
```

Ook voor vreemde sleutels die slechts uit één enkel attribuut bestaan is een kortere notatie mogelijk, die overeenkomt met de verkorte notatie geïntroduceerd in 4.2. Let wel op dat er expliciet moet worden meegegeven naar welk attribuut en naar welke basisrelatie de vreemde sleutel verwijst.

Tot slot is het belangrijk om even stil te staan bij de volgorde van de aanmaak van de basisrelaties. In bovenstaand voorbeeld hebben we eerst de tabel `type` gedefinieerd, en vervolgens pas de tabel `constructeur`. De omgekeerde volgorde zou niet mogelijk geweest zijn: aangezien `constructeur` een vreemde sleutel bevat die verwijst naar `type`, kan de tabel `constructeur` pas worden aangemaakt als de tabel `type` al bestaat.

4.4 Constraints

Bovenstaande informatie stelt je in staat een databank aan te maken die basisrelaties, primaire sleutels en vreemde sleutels bevat. Met deze componenten kan echter niet alle functionaliteit worden geïmplementeerd die door het logisch ontwerp wordt beschreven. Het logisch ontwerp bevat immers ook een aantal extra beperkingen, die een resem voorwaarden opleggen waaraan attribuutwaarden of combinaties aan attribuutwaarden moeten voldoen. Om een optimale, 100% correct functionerende databank te bekomen moeten deze dus op een of andere manier ook in de fysieke databank tot uiting komen. In databankterminologie heten dergelijke voorwaarden 'constraints'. Merk op dat primaire en vreemde sleutels eigenlijk ook constraints zijn aangezien deze ook voorwaarden opleggen op de waarden die attributen in een bepaalde basisrelatie al dan niet mogen aannemen. Daarnaast bestaan er echter nog andere soorten constraints, die in het vervolg worden besproken.

4.4.1 `UNIQUE`

Een `UNIQUE` constraint vereist dat voor een bepaalde set van attributen in een basisrelatie elke combinatie van attribuutwaarden die in de basisrelatie wordt opgeslagen uniek moet zijn. Dit is bijvoorbeeld het geval voor het attribuut `naam` in de basisrelatie `type`. Wanneer enkel een `UNIQUE` constraint wordt opgelegd aan een attribuut of verzameling van attributen, wordt echter wel toegestaan dat de attribuutwaarde(n) ervan `NULL`-waarden bevat. Dit zijn waarden die aangeven dat de exacte waarde van een bepaald attribuut voor een bepaald record niet gekend is. Het toevoegen van een `UNIQUE` constraint aan een reeds aangemaakte basisrelatie met naam `relatie` over de attributen $(\texttt{attr}_1,\ldots,\texttt{attr}_n)$ gebeurt in PostgreSQL met behulp van volgend statement.

```
ALTER TABLE relatie
ADD CONSTRAINT UC_relatie UNIQUE (attr1,...,attrn);
```

Merk op dat het toevoegen van een `UNIQUE` constraint echter ook opnieuw rechtstreeks kan gebeuren bij aanmaak van de basisrelatie, hetzij in verkorte notatie in het geval van een enkel attribuut, hetzij door toevoeging van een `CONSTRAINT` definitie aan het `CREATE TABLE` statement. Dit is vergelijkbaar met de aanmaak van primaire en vreemde sleutels.

4.4.2 NOT NULL

Een tweede soort voorwaarde legt op dat voor een bepaald attribuut iedere attribuutwaarde wel degelijk een waarde moet bevatten. Met andere woorden, de waarde NULL wordt voor dat attribuut niet aanvaard. Toevoeging van NOT NULL voor het attribuut attr horend bij de basisrelatie relatie die al werd aangemaakt kan als volgt.

```
ALTER TABLE relatie
ALTER COLUMN attr SET NOT NULL;
```

Dit statement zorgt dus voor een aanpassing aan de definitie van zowel de gegeven tabel alsook aan een kolom van deze tabel. Daarnaast kan je een NOT NULL constraint direct meegeven bij aanmaak van een basisrelatie door toevoeging van NOT NULL direct na het datatype van het betreffende attribuut in het CREATE TABLE statement.

Opmerking: Een primaire sleutel is in essentie niets anders dan een verzameling van attributen waarop automatisch een UNIQUE constraint is gedefinieerd en waarvoor op elk van de afzonderlijke attributen een NOT NULL constraint wordt opgelegd. Er kunnen in eenzelfde basisrelatie meerdere verzamelingen van attributen zijn waarvoor zowel een UNIQUE als een NOT NULL constraint gelden. Er kan echter slechts één hiervan gedefinieerd worden als primaire sleutel van de basisrelatie. De rest zijn alternatieve sleutels.

4.4.3 CHECK

Een laatste, veelgebruikte soort van constraint is de CHECK constraint. Dit soort constraint wordt gebruikt om het domein (mogelijke waarden) van een attribuut te limiteren. Een dergelijke constraint kan zowel gedefinieerd worden op een enkel attribuut als over meerdere attributen heen. Indien een CHECK constraint over een enkel attribuut wordt gedefinieerd, kan deze enkel beperkingen opleggen op de waarden die in deze kolom worden aangenomen.
Een voorbeeld uit de beperkingen opgelegd aan de Rolls Robin databank is de voorwaarde dat het dagtarief van een type wagen positief moet zijn. Deze beperking kan worden afgedwongen in de fysieke PostgreSQL databank met behulp van volgende CHECK constraint.

```
ALTER TABLE type ADD CHECK (dagtarief >= 0);
```

Wanneer je later zal proberen om wagentypes in te voeren met een strikt negatief dagtarief, zal deze constraint ervoor zorgen dat PostgreSQL een foutmelding zal opwerpen. CHECK constraints kunnen gebruikt worden om beperkingen te implementeren zolang het enkel betrekking heeft op attributen uit dezelfde basisrelatie

en zolang er bij de verificatie van een voorwaarde enkel data nodig zijn die behoren tot het record dat men wil opslaan. Het is bijvoorbeeld perfect mogelijk om met één enkele `CHECK` constraint de beperking te implementeren waarbij wordt geëist dat een startdatum altijd vroeger moet plaatsvinden dan een einddatum (zie tabellen `contract` en `registratieformulier`).
Net zoals bij de andere constraints moeten `CHECK` constraints niet per se na aanmaak van een tabel worden toegevoegd, maar kunnen deze ook direct bij aanmaak worden gedefinieerd. Bovendien is ook hier opnieuw de verkorte notatie mogelijk indien de `CHECK` constraint slechts betrekking heeft tot een enkel attribuut.

4.4.4 Views, stored procedures en triggers

Met behulp van de hierboven uitgelegde concepten zijn nu quasi alle beperkingen van het logisch ontwerp implementeerbaar op het fysiek niveau. In het logisch ontwerp worden echter ook nog enkele beperkingen geïdentificeerd die niet door een van de eerder uitgelegde concepten kunnen worden geïmplementeerd. Twee soorten beperkingen vallen hieronder.
De eerste soort heeft te maken met beperkingen die voortvloeien uit afgeleide attributen uit het conceptuele ontwerp. Deze kunnen worden gerealiseerd met behulp van views. Dit is een concept dat in een latere workshop aan bod zal komen. In het geval van de `rollsrobin` databank zijn er zo echter geen voorwaarden die moeten worden vervuld.
Daarnaast is er ook nog een tweede groep van ‘moeilijkere’ beperkingen, namelijk deze waarvan we in het logisch ontwerp enkel maar hun gewenst gedrag hebben onderzocht wanneer er data aan de databank werd toegevoegd. Een voorbeeld van een dergelijke beperking is dat het niet toegelaten is dat eenzelfde werknemer op hetzelfde tijdstip bij verschillende filialen werkt. Dit soort beperkingen valt inderdaad niet op te lossen met een eenvoudige `CHECK`-constraint. Wanneer je dit wil verifiëren bij invoering van een nieuw record in de `contract` tabel, heb je immers ook de data nodig die reeds in de tabel opgeslagen zit. Om dergelijke problemen op te lossen, heb je geavanceerdere concepten zoals functies en triggers nodig.
Voorlopig is het niet nodig om dit soort voorwaarden af te dwingen. In de workshop ‘Triggers, functies & stored procedures’ zullen we op deze specifieke voorwaarden dieper ingaan.

4.4.5 `serial`-datatype

Wanneer je in PostgreSQL een (artificiële) surrogaatsleutel (typisch een ID) wil implementeren (= een automatisch gegenereerd incrementerend getal) kan dit met het `serial`-datatype. Dit is eigenlijk geen écht datatype, maar het is een PostgreSQL-specifieke shortcut om aan te geven dat automatisch een unieke integer-waarde

moet gegenereerd worden voor elke rij van dit attribuut[5]. Het eigenlijke datatype van een '`serial`' attribuut zal dus integer zijn. Dit is belangrijk wanneer je later met een *vreemde sleutel* naar dit attribuut wil verwijzen. Het datatype van dit vreemde sleutel-attribuut moet dus integer zijn.

Vertaal alle geïdentificeerde basisrelaties, primaire sleutels, vreemde sleutels en (eenvoudige) constraints uit het logisch ontwerp van Rolls Robin naar een fysieke PostgreSQL implementatie.

5 User privileges

Als alles goed is, heb je op dit moment een praktisch bruikbare databank die voldoet aan quasi alle noodzakelijke vereisten. Deze databank zou vervolgens gebruikt kunnen worden door het bedrijf Rolls Robin. Rolls Robin wil natuurlijk niet dat iedereen zomaar toegang heeft tot de gegevens die in de databank zullen worden opgeslagen. Daarnaast zal er ook binnen het bedrijf zelf bepaald moeten worden wie welke rechten heeft op de databank. Zo kan de databank bijvoorbeeld gebruikt worden door filiaalmedewerkers (die willen interageren met de databank via een grafische interface). Zij zouden in staat moeten zijn om bijvoorbeeld nieuwe klanten of nieuwe verhuringen toe te voegen aan de databank. Anderzijds willen we niet dat een filiaalmedewerker per ongeluk alle data uit een bepaalde basisrelatie, of nog erger, uit de volledige databank kan verwijderen. Naast de filiaalmedewerkers zijn er echter ook nog andere profielen binnen het bedrijf, zoals bijvoorbeeld een databankbeheerder. Aangezien deze instaat voor het onderhoud van de databank en de voortdurende goede werking ervan, moet deze natuurlijk alles kunnen doen wat noodzakelijk is voor de databank. Je merkt dat we dus een authenticatiemechanisme nodig hebben, zodanig dat voor iedere connectie met de databank geweten is welk type gebruiker (rol) geconnecteerd is en wat hij/zij mag doen. Bovendien willen we aan iedere rol de specifieke acties en handelingen kunnen koppelen die deze rol mag uitvoeren.
Gelukkig heeft PostgreSQL een dergelijk authenticatiemechanisme. PostgreSQL werkt namelijk met zogenaamde 'database roles' of databankrollen. Een rol kan zowel een individuele gebruiker (bijvoorbeeld 'Jan Janssens') als een groep van gebruikers (bijvoorbeeld 'filiaalmedewerker') voorstellen. Een rol omvat een naam, een paswoord en een geassocieerde lijst aan rechten die een persoon die onder die rol is ingelogd kan uitvoeren op de databank. Bij de installatie van PostgreSQL wordt standaard één rol aangemaakt, namelijk de 'postgres' rol (standaard zonder

[5] `https://www.postgresql.org/docs/current/datatype-numeric.html#DATATYPE-SERIAL`

paswoord). Dit is ook hoe jij momenteel bent geconnecteerd met de databank, al is dit in de praktijk natuurlijk niet gewenst.
Om een nieuwe rol met naam ‘user’ met geassocieerd paswoord ‘password’ aan te maken volstaat volgend statement.

```
CREATE ROLE user WITH LOGIN PASSWORD 'password';
```

Wie de gebruikersnaam en het geassocieerde paswoord kent, kan dus vervolgens connecteren met de databank (bijvoorbeeld via psql, pgAdmin 4 of een ander gebruikersprogramma). Wat een gebruiker onder een bepaalde rol vervolgens kan doen met behulp van de databank hangt echter af van welke geassocieerde rechten (privileges) aan deze rol werden toegekend. Rechten kunnen zowel worden toegekend op het niveau van een databank, van een schema én van een tabel. De belangrijkste acties waarvoor rechten kunnen worden toegekend zijn de volgende.

- `SELECT`: laat toe dat de rol leest uit een bepaalde tabel.
- `INSERT`: laat toe dat de rol data toevoegt aan een tabel.
- `UPDATE`: laat toe dat de rol reeds aanwezige data in een tabel aanpast.
- `DELETE`: laat toe dat de rol data uit een tabel verwijdert.
- `TRUNCATE`: laat toe dat de rol alle data uit een bepaalde tabel verwijdert.
- `REFERENCES`: laat toe dat de rol een vreemde sleutel definieert. Om een vreemde sleutel aan te kunnen maken, moet dit recht zowel worden toegekend voor de tabel waarin de vreemde sleutel wordt gedefinieerd, als de tabel waarnaar de vreemde sleutel verwijst.
- `TRIGGER`: laat toe dat de rol een trigger aanmaakt op een bepaalde tabel.

Indien een rol een bepaalde tabel aanmaakt, worden alle bovenstaande rechten op deze tabel automatisch aan deze rol toegekend. Aangezien je gedurende dit practicum steeds hebt gewerkt onder de rol ‘postgres’, heeft deze rol dus alle bovenstaande rechten op de tabellen die je eerder aanmaakte in de `rollsrobin` databank. Nieuw aangemaakte rollen hebben standaard echter geen enkel recht.

Maak voor de `rollsrobin` databank een nieuwe rol met naam ‘test’ aan met bijhorend paswoord ‘testpw’. Maak vervolgens een nieuwe connectie met naam ‘test_connectie’ in pgAdmin 4 aan op ‘localhost:5432’, zoals eerder geleerd, waarbij je ditmaal niet inlogt als gebruiker ‘postgres’ maar als gebruiker ‘test’. Je hoeft je oude connectie hier niet voor te verwijderen!

Probeer vervolgens als ‘test’ gebruiker de data te lezen die in de tabel `filiaal` zijn opgeslagen. Dit doe je door rechts te klikken op de naam van de tabel - View/Edit Data - All rows. . .

Als alles goed ging zou je een foutmelding moeten krijgen. Dat is logisch aangezien de rol ‘test’ geen leesrechten heeft op de tabel `filiaal`! We moeten eerst leesrechten koppelen aan de ‘test’ rol vooraleer iemand die is ingelogd onder deze rol ook effectief de data kan lezen. Ga daarom terug naar de connectie die je hebt als ‘postgres’ gebruiker. Ken deze gebruiker leesrechten toe op de tabel filiaal als volgt:

```
GRANT SELECT ON filiaal TO test;
```

Met dit statement ken je dus `SELECT` rechten toe op de tabel `filiaal` voor de rol met naam ‘test’.

Ken leesrechten toe aan de rol ‘test’ op de tabel `filiaal`. Ga nu opnieuw naar je ‘test_connectie’ connectie en probeer nogmaals de data uit de tabel `filiaal` te lezen.

Als alles goed ging, zou pgAdmin nu een lege tabel moeten tonen. Dit is logisch, aangezien we nog geen data hebben opgeslagen in de databank. Je ziet echter dat je nu de structuur (de attributen en de datatypes) van de tabel wel degelijk kan bekijken, in tegenstelling tot bij de vorige poging.
Zoals hierboven getoond laat het `GRANT` statement dus toe om een specifiek recht op een bepaalde tabel te koppelen aan een rol. Alhoewel dit enkel werd voorgetoond voor leesrechten (`SELECT`), is het `GRANT` statement bruikbaar voor elk recht dat kan worden toegekend in PostgreSQL (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`,. . .). Indien je echter aan een bepaalde rol, alle mogelijke rechten op een bepaalde tabel wil toekennen, is dit ook mogelijk door uitvoering van één enkel statement.

```
GRANT ALL PRIVILEGES ON table_name TO role_name;
```

Tot slot is het ook mogelijk om alle rechten op alle tabellen in een welbepaald databankschema in één enkel statement toe te kennen met behulp van volgend statement.

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA schema_name TO role_name;
```

Stel dat je uiteindelijk iemand (een deel van) zijn rechten op een bepaalde tabel wil afnemen, dan kan dit ook. Hiervoor kan het REVOKE statement worden gebruikt als volgt.

```
REVOKE x ON table_name FROM role_name;
```

Hierbij staat `x` voor het recht dat je ongedaan wil maken.
Indien je een rol wil verwijderen kan dit met het volgende statement.

```
DROP ROLE role_name;
```

Let wel op dat je dan eerst alle rechten van deze gebruiker moet afnemen.

> Sluit de connectie 'test_connectie' af. Verwijder daarna vanuit de oorspronkelijk connectie de rol met naam 'test'.

Meer informatie in verband rollen en gebruikersrechten kan je vinden op `https://postgresql.org/docs/current/user-manag.html`.

6 Backup en restore

De databank die je hebt aangemaakt en de data die je er in de volgende workshops in zal laden worden opgeslagen in het permanente geheugen (d.i. de harde schijf). Dat maakt dat de betrouwbaarheid van de opslag rechtstreeks afhankelijk is van je hardware. Als je niet de juiste voorzorgsmaatregelen neemt, bestaat de kans dat op een dag je harde schijf faalt en je alle gegevens kwijt bent. Om dergelijke scenario's te vermijden en om de inwisselbaarheid van databanken over verschillende machines te vergroten, heeft PostgreSQL een backup-mechanisme voorzien.
Dit mechanisme vertaalt je huidige databank naar een lange reeks SQL-statements die nodig en voldoende zijn om de databank later te kunnen reconstrueren. Dit is niet alleen handig om een backup van je databank te hebben uit veiligheidsoverwegingen, maar ook om een databank van één machine naar een andere te kopiëren, zonder dat je daarvoor elke stap uit het fysiek ontwerp op de nieuwe machine moet herhalen en vervolgens alle data in de nieuwe databank opnieuw moet inladen.
In het vervolg zullen we tonen hoe je een backup maakt van je database en hoe je deze backup ook weer kunt inladen via de commandolijn.

6.1 Backup

Via de commandolijn-applicatie `pg_dump` is het mogelijk om een backup te nemen van je databank. Dit commando start net als psql een nieuw programma op dat kan uitgevoerd worden op de commandolijn. Let dus goed op dat je dit **niet** uitvoert binnen de `psql` omgeving. `pg_dump` schrijft alle commando's die noodzakelijk zijn om de PostgreSQL databank waarvan je een back-up wenst te nemen weg naar een .sql-bestand. Het backup-bestand bevat dus in feite niets anders dan een sequentie SQL commando's die men zou moeten uitvoeren om, vertrekkende van niets, te eindigen met een database zoals die is op het moment van de backup. `pg_dump` heeft heel wat verschillende parameters die je kan meegeven wanneer je het programma wil uitvoeren:

```
pg_dump
    --clean
    --create
    --if-exists
    --dbname rollsrobin
    --schema-only
    --username postgres
    --file rollsrobin.sql
```

Aan de optie `dbname` geef je de naam van de database mee waarvan je een backup wil maken. De opties `clean` en `if-exists` zorgen er voor dat het `DROP DATABASE IF EXISTS` commando wordt toegevoegd aan de .sql-backupfile. In dit geval wordt, indien er reeds een databank met de opgegeven naam bestaat, deze databank eerst verwijderd. De optie `create` zorgt ervoor dat in de backup ook het `CREATE DATABASE` commando wordt opgenomen. Deze maakt een nieuwe, lege databank met de gespecificeerde naam aan. Met de `schema-only` parameter geef je aan dat je enkel de definitie van de databank wil backuppen, en niet de data. De optie `username` geeft aan via welke gebruiker je een backup wilt maken. Na de `file` parameter geef je de relatieve padnaam op van het backup-bestand. Je zal zien dat je op een labo-PC enkel rechten hebt om deze weg te schrijven naar de H-schijf! In dit geval dient de relatieve padnaam die je opgeeft dus te starten met "h:/", waarna je de gewenste locatie waar je het backup-bestand wil opslaan kan specificeren. Let op dat je het commando en de opties op 1 lijn typt (dus zonder 'enters' tussen).

Maak een schema-backup van de `rollsrobin` databank waarin de database **niet** opnieuw wordt aangemaakt. Je moet dus zelf nadenken welke opties er moeten worden meegegeven aan het `pg_dump` commando.

6.2 Restore

Via de commandolijn-applicatie `psql` is het mogelijk om een backup van je database terug in te lezen (restore). Dit kan je doen met volgend commando.

```
psql
    --dbname rollsrobin_backup
    --username postgres
    --file rollsrobin.sql
```

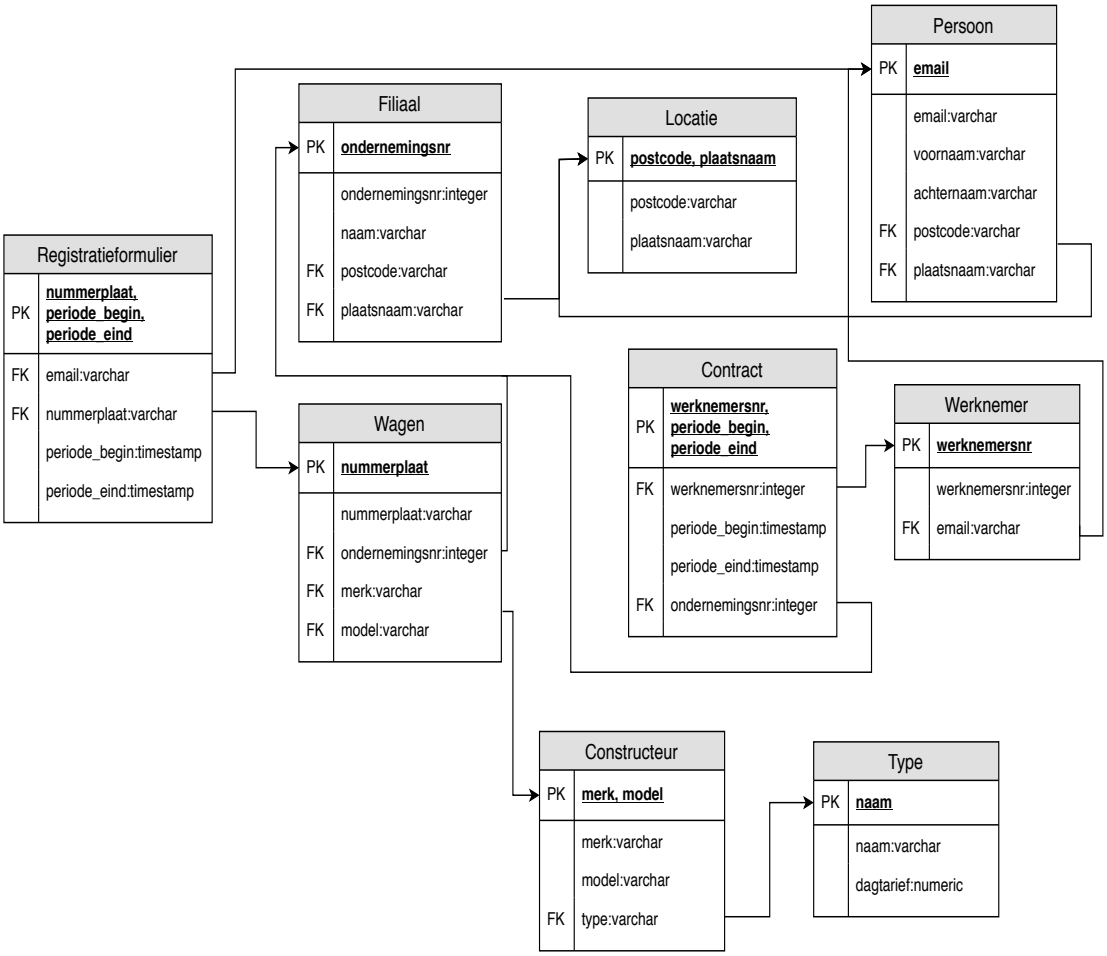
Dit commando is gelijkaardig aan het `pg_dump` commando. We gaan dus niet verder in op de betekenis van de opties. Let op, de `dbname` parameter kan je enkel gebruiken indien je backup-script geen databankaanmaak-statements bevat.

Maak via psql een tweede databank aan met naam `rollsrobin_backup`. Verlaat vervolgens het programma psql (via Ctrl + C) en restore de databank naar `rollsrobin_backup` waarvan je een backup hebt gemaakt in 6.1

.

Appendix: Logisch databankontwerp Rolls Robin

In onderstaande figuur vind je een schematische weergave van een mogelijk logisch ontwerp voor de Rolls Robin databank. Hierbij wordt iedere basisrelatie weergegeven door een rechthoek, die bovendien een oplijsting van alle attributen met bijhorend datatype bevat. Daarnaast worden de attributen die behoren tot de primaire sleutel bovenaan weergegeven, en worden vreemde sleutels voorgesteld door een pijl tussen de betreffende attribuutverzamelingen. Alle extra beperkingen die niet kunnen worden weergegeven in dit schema worden onderaan opgelijst.



Extra beperkingen

- Type:
  1. CHECK: dagtarief ≥ 0
- Registratieformulier:

1. CHECK: periode_begin ≤ periode_eind
2. Insert: controleer bij toevoegen van registratieformulier dat periode niet overlapt met periode van ander registratieformulier voor dezelfde nummerplaat.

- Contract:
  1. CHECK: periode_begin ≤ periode_eind
  2. Insert: controleer bij toevoegen van nieuw contract dat periode van contract niet overlapt met periode van ander contract voor hetzelfde werknemersnummer.