Universiteit Gent - TELIN - DDCM
2019-2020

Workshop: Triggers, functies & views

1 Introductie

Tijdens de eerste oefeningenlessen richtten we onze aandacht vooral op het conceptueel en logisch ontwerp van een databank. Deze fases resulteerden in blauwdrukken die het databankontwerpprobleem op een respectievelijk abstracte en relationele wijze voorstellen. Tijdens de workshop 'Introductie PostgreSQL & Fysiek ontwerp' implementeerden we alle basiscomponenten van het logische ontwerp fysiek. Het resultaat is een werkende databank die gevuld kan worden met data (zie workshop 'Importeren van data') en waarop we tal van manipulaties (zie workshop 'Datamanipulatie') en analyses (zie SQL-lessen) kunnen doen.
Zoals jullie reeds weten zijn voorbeelden van basiscomponenten die we in een relationeel systeem vaak gebruiken tabellen, attributen, primaire en vreemde sleutels, `CHECK`-constraints,... De aandachtige student zal gemerkt hebben dat we hiermee niet alle gewenste functionaliteit kunnen implementeren. Daarom keren we in deze workshop even terug op het fysiek ontwerp en gaan we in detail kijken naar geavanceerdere concepten die ervoor zullen zorgen dat we in staat zijn om effectief alle gevraagde functionaliteit te implementeren in de databanklaag. Deze concepten worden functies, triggers en views genoemd.
Ook tijdens deze workshop zullen we werken met de 'Rolls Robin' voorbeeldopgave die reeds in de vorige oefeningenlessen en workshops aan bod kwam. Voor de volledigheid is het relationele databankschema van de 'Rolls Robin' databank terug te vinden in de Appendix bij deze workshop. Ook vinden jullie op Ufora (in de map van de workshop 'Datamanipulatie') een SQL-script dat alle statements bevat voor de (voorlopige) fysieke implementatie van deze databank. Zo kunnen jullie met een correcte databankimplementatie starten aan deze workshop.

2 Eerste stappen

Vooraleer we dieper ingaan op elk van de genoemde concepten, komen we eerst nog even kort terug op een aantal zaken uit de vorige workshops.
Ten eerste willen we jullie eraan herinneren dat alle info over het downloaden en installeren van PostgreSQL en alle gerelateerde tools op jullie eigen PC is samengevat in een handleiding op Ufora. Daarnaast is er ook een handleiding te vinden omtrent het gebruik van PostgreSQL op de labo-PC's. Vooraleer jullie starten met deze workshop is het aangeraden om het document dat voor jullie van toepassing is grondig door te nemen.
Als opstart voor deze workshop is een werkende fysieke implementatie van de `rollsrobin` databank nodig en/of een backup in de vorm van een SQL-script van deze databank. Het is dus niet noodzakelijk om een `rollsrobin` databank te hebben die reeds gevuld is met data. Meerbepaald zouden jullie de workshop 'Inleiding PostgreSQL & Fysiek ontwerp' en het deel omtrent aanpas- en verwijderbeperkingen in de workshop 'Datamanipulatie' reeds gemaakt moeten hebben. Mocht dit niet het geval zijn, kunnen jullie een volledig backup-script terugvinden in de map van de workshop 'Datamanipulatie' op Ufora. Dit script kan je gebruiken als voorbeeldoplossing van het fysieke ontwerp van de `rollsrobin` databank. Indien jullie dit script wensen te gebruiken is het aangeraden om de oplossing in detail door te nemen vooraleer verder te gaan, zodat je zeker bent dat je alle concepten uit de vorige workshops goed begrijpt.
Een fysieke implementatie van de `rollsrobin` databank bekomen jullie door deze backup in te laden (restore). Dit kan je doen via de commandolijn-applicatie `psql` met volgend commando:

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

Let wel op dat er reeds een lege `rollsrobin` databank moet bestaan op je lokale PostgreSQL-cluster. Deze dien je dus eerste zelf aan te maken.

> Zorg vooraleer verder te gaan dat je een volledig geïmplementeerde `rollsrobin` databank hebt aangemaakt op je lokale PostgreSQL-cluster.

Wanneer jullie kijken naar de huidige `rollsrobin` databank, zullen jullie merken dat alle basisrelaties (tabellen), attributen (kolommen), primaire en vreemde sleutels, `CHECK`-, `UNIQUE` en `NOT NULL`-constraints reeds geïmplementeerd zijn. Ook hebben

we reeds beperkingen afgedwongen in verband met het aanpassen en verwijderen van data. Bij het vergelijken van het logisch databankontwerp met het fysieke ontwerp, merken jullie dat er nog 2 beperkingen ontbreken in de fysieke implementatie. Dit zijn de beperkingen die stellen dat periodes van contracten van een werknemer en van registraties van een wagen niet mogen overlappen. Tijdens het logisch ontwerp hebben we reeds nagedacht over het gewenst gedrag met betrekking tot deze beperkingen bij het toevoegen van data in de tabel `contract` (resp. `registratieformulier`). Tijd om dit gedrag nu ook effectief fysiek te gaan implementeren...

3 Functies

Tot op dit moment was het mogelijk om bij het fysieke ontwerp van een databank alle beperkingen te implementeren bij het definiëren van de tabellen. Toen we bijvoorbeeld wouden afdwingen dat het dagtarief van een wagentype een positief getal moest zijn, konden we dit eenvoudigweg doen met behulp van een `CHECK`-constraint op de tabel `wagentype`. Simpele `CHECK`-constraints volstaan echter niet meer op het moment dat we beperkingen moeten leggen op data over verschillende tabellen en/of rijen heen. We hebben dus extra functionaliteit nodig om ervoor te zorgen dat er op geen enkele manier ongeldige data in de databank terechtkomt. PostgreSQL laat toe om (net als procedurele programmeertalen zoals C, Pascal,...) deze extra functionaliteit te gaan implementeren door middel van gebruikersgedefinieerde functies. Binnen deze functies is het mogelijk om heel wat procedurele elementen zoals controlestructuren, loops, variabelen,... te gebruiken. Aangezien deze elementen niet voorzien worden in het standaard PostgreSQL-dialect, voegt PostgreSQL de mogelijkheid toe om binnen een databank deze functies te schrijven in een procedurele programmeertaal. Voor deze workshop kiezen we voor de PL/pgSQL taal, die nauw verwant is met standaard SQL.
Op zich zou je 'moeilijkere' beperkingen, zoals het feit dat tijdsintervallen van contracten voor eenzelfde werknemer niet mogen overlappen, kunnen afdwingen vooraleer de data in de databank terechtkomt (bijvoorbeeld bij het creëren van de data of in het programma dat de data wegschrijft naar de databank), zodanig dat je dit soort beperkingen niet meer moet gaan afdwingen in de databanklaag zelf. Het definiëren van functies binnen de databanklaag in plaats van binnen de applicatielaag brengt echter heel wat voordelen met zich mee, namelijk

- een afname van het aantal oproepen vanuit de applicatie naar de databankserver, waardoor de performantie van de applicatie verbetert. De applicatie kan vertrouwen op de goede werking van de databank wanneer er data naar toe wordt geschreven;
- de herbruikbaarheid binnen meerdere applicaties (die mogelijks in andere pro-

grammeertalen geschreven zijn). Het is inderdaad beter om alles slechts één keer te moeten implementeren in de databank, i.p.v. de beperkingen opnieuw te implementeren in elke applicatie die data wil wegschrijven naar deze databank;

- een optimale benutting van de kosten gespendeerd aan de databanklaag en van de functionaliteit die het databankbeheersysteem aanbiedt (zeker wanneer er gebruik gemaakt wordt van betalende systemen).

Indien je dus de mogelijkheid hebt tot het definiëren van een functie in de databanklaag, wordt dit ook ten zeerste aangeraden.
In 3.1 en 3.2 gaan we dieper in op de definitie van functies in PostgreSQL en implementeren we als voorbeeld een eenvoudige teller. In Sectie 4 kijken we dan hoe we functies kunnen linken aan bepaalde gebeurtenissen door middel van triggers. Let op, we gaan niet in detail in op alle syntax die we gebruiken in de voorbeelden. Voor problemen, vragen of opmerkingen met betrekking tot de syntax kunnen jullie steeds terecht op `http://www.postgresqltutorial.com/`, in de documentatie van PostgreSQL of bij de assistenten.

3.1 Functiedefinitie

Om een functie te definiëren maak je standaard gebruik van het `CREATE FUNCTION` commando[1]. Dit commando heeft de volgende syntax.

```
CREATE FUNCTION function_name(p1 type,...,pn type)
    RETURNS type AS
$BODY$
DECLARE
    v1 type;
    ...
    vn type;
BEGIN
    --logic
END;
$BODY$
LANGUAGE language_name;
```

De verschillende delen die een functie (mogelijks) bevat zijn de volgende:

- De naam van de functie (naar keuze) wordt opgegeven na het `CREATE FUNCTION` commando.

[1] `https://www.postgresql.org/docs/current/sql-createfunction.html`

- Na de naam volgt er, tussen haakjes, een komma-gescheiden lijst van parameters die je wil meegeven aan de functie. Elke parameter heeft een naam en een datatype.
- Na het `RETURNS` sleutelwoord wordt het datatype opgegeven van de waarde die door de functie teruggegeven zal worden.
- Een oplijsting van lokale variabelen (elk met een lokale naam en een datatype) die gebruikt worden in de functie worden opgegeven na het `DECLARE` sleutelwoord.
- Tussen het `BEGIN` en `END` sleutelwoord volgt de eigenlijke functie-logica.
- Het `CREATE FUNCTION` commando wordt afgesloten met de definitie van de procedurele taal die gebruikt wordt binnen de functie. In het geval van PL/pgSQL geef je `plpgsql` op.

Tot slot zie je ook dat `$BODY$` geplaatst wordt rond de effectieve inhoud ('body') van de functie. De reden hiervoor is dat, wanneer je de inhoud op basis van PL/pgSQL definieert, dit als een string moet worden doorgegeven aan het `CREATE FUNCTION` commando. Aangezien het gebruik van enkele aanhalingstekens binnen de functie voor problemen kan zorgen, wordt het dubbele dollarteken als vervanger voor een enkel aanhalingsteken gebruikt. Tussen de dollartekens is het mogelijk om een optionele naam (naar keuze) mee te geven. In dit geval werd er gekozen voor de naam `BODY`.

3.2 De functie `increment`

Als voorbeeld zullen we een eenvoudige functie definiëren die een opgegeven waarde verhoogt met 1, het resultaat uitprint als info-bericht en dit resultaat ook teruggeeft. De code voor deze functie[2] is hieronder gegeven.

```
CREATE FUNCTION increment(value integer)
    RETURNS integer AS
$BODY$
DECLARE
    result integer;
    incrementer CONSTANT integer := 1;
BEGIN
    result := value + incrementer;
    RAISE INFO 'the result of this call is %', result;
    RETURN result;
```

[2] Dit is niet noodzakelijk de meest performante code.

```
END;
$BODY$
LANGUAGE plpgsql;
```

Deze functie zal dus een enkele parameter verwachten met de naam `value` van het `integer` datatype. Er zijn 2 lokale variabelen: `result` die het resultaat zal bevatten van de verhoging en `incrementer` die een waarde bevat om toe te voegen aan de waarde van `value`. Belangrijk om op te merken is dat `incrementer` gedefinieerd is als een constante en de waarde hiervan dus **niet** kan veranderen eens er een waarde aan toegekend is. In de functie wordt de waarde in de `result` variabele uitgeprint in een info-bericht (`RAISE INFO`) en teruggegeven (`RETURN result`).

Implementeer deze functie binnen de `rollsrobin` databank. Je kan dit doen door een query tool te openen binnen deze databank, de code hierin te typen en uit te voeren. Indien dit gelukt is zal je een bericht krijgen dat de query succesvol is uitgevoerd en je functie dus is geïmplementeerd.

3.3 Een functie oproepen

Net als uitvoering van standaard SQL functies, kan je een gebruikersgedefinieerde functie als volgt oproepen.

```
SELECT function_name(value_1,...,value_n)
```

De verschillende `value`-parameters moet je vervangen door effectieve parameterwaarden overeenkomstig aan de lijst van parameters gedefinieerd binnen de functie. Indien deze query succesvol uitvoert, zal je een resultatentabel te zien krijgen.

Test of de functie `increment` werkt en het verwachte resultaat teruggeeft door een aantal waarden te testen. Zie je ook het verwachte info-bericht onder “messages” in pgAdmin 4?

4 Triggers

We weten nu hoe we in PostgreSQL functies kunnen definiëren. Aan een functie op zichzelf hebben we echter niet genoeg om (complexe) beperkingen te voorzien in de `rollsrobin` databank. Met de introductie van het concept functies zijn we immers

in staat om bepaalde procedurele acties uit te voeren, maar we willen ook kunnen opleggen wanneer deze acties precies moeten worden uitgevoerd. Dit is met gewone functies niet mogelijk. Om dit probleem op te lossen introduceren we in deze sectie het concept 'trigger'. Een trigger voegt, net als een functie, ook bepaalde gebruikersgedefinieerde functionaliteit toe. Meer nog, een trigger maakt zelfs gebruik van functies. Het grote verschil met functies is dat triggers enkel opgeroepen worden indien er zich een bepaalde gebeurtenis (die vooraf gedefinieerd is) voordoet op een tabel (waaraan de trigger gekoppeld is). Indien er zich zo een gebeurtenis voordoet zal een trigger de functie die gekoppeld is aan deze trigger automatisch uitvoeren.
Triggers worden typisch gebruikt voor twee verschillende gevallen. Ten eerste worden ze gebruikt in een context waarbij een databank in parallel toegankelijk is voor meerdere applicaties en/of gebruikers en waarbij je bijvoorbeeld een geschiedenis (log) wil bijhouden van veranderingen binnen deze databank. Je kan dan een trigger definiëren die, iedere keer er een gebruiker gebruik maakt van de databank, de precieze acties van deze gebruiker wegschrijft naar een bestand. Hierdoor hoef je deze functionaliteit niet meer in de verschillende applicaties afzonderlijk te implementeren. Daarnaast worden triggers ook vaak gebruikt om complexe data-integriteitsregels af te dwingen (zoals beperkingen). En laat dit tweede geval nu exact zijn wat we nodig hebben...

4.1 Trigger functies

Een eerste stap in de definitie van een trigger is het implementeren van een trigger functie. Een trigger functie is zeer gelijkaardig aan een functie zoals gezien in Sectie 3, met het verschil dat een trigger functie nooit parameters zal hebben en altijd een waarde van het datatype `trigger` zal teruggeven. De syntax komt dus ook sterk overeen met de syntax voor de aanmaak van een functie en is hieronder weergegeven.

```
CREATE FUNCTION function_name()
    RETURNS trigger AS
...
LANGUAGE language_name;
```

4.2 Triggerdefinitie

Het definiëren van een trigger gebeurt door middel van het `CREATE TRIGGER` commando[3]. De (basis)syntax van dit commando is de volgende[4].

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

[4] De vierkante haakjes en accolades zijn geen deel van de syntax maar duiden respectievelijk een optioneel stuk en een keuze aan.

```
CREATE TRIGGER trigger_name
    {BEFORE | AFTER} {event [OR ...]}
        ON table_name
        [FOR [EACH] {ROW | STATEMENT}]
            EXECUTE PROCEDURE trigger_function_name;
```

De verschillende delen die een trigger-definitie omvat zijn de volgende:

- De naam van de trigger (naar keuze) wordt opgegeven na het `CREATE TRIGGER` commando.
- Een trigger functie kan opgeroepen worden voor (`BEFORE`) of na (`AFTER`) een gebeurtenis. In het geval een trigger functie wordt opgeroepen voor een gebeurtenis kan het mogelijks de gebeurtenis tegenhouden (wat de performantie ten goede kan komen). Indien een trigger functie na een gebeurtenis wordt opgeroepen kan het de aanpassingen die gebeurd zijn in de databank tijdens die gebeurtenis meteen gebruiken.
- `event` definieert de gebeurtenis waarop de trigger functie opgeroepen moet worden. Mogelijke gebeurtenissen zijn `INSERT` (toevoeging van data), `UPDATE` (aanpassing van data), `DELETE` (verwijderen van data) of `TRUNCATE` (leegmaken van volledige tabel). Het is ook mogelijk om meerdere gebeurtenissen op te geven door middel van concatenatie met `OR`.
- `table_name` geeft de tabel aan waarop de gebeurtenis moet plaatsvinden voor het uitvoeren van de trigger functie.
- `[FOR [EACH] {ROW | STATEMENT}]` bepaalt of de trigger op rijniveau of op statementniveau uitgevoerd wordt. Stel dat je bijvoorbeeld een aanpassing doet van 20 rijen tegelijk in de databank, dan zal een trigger op rijniveau 20 keer worden uitgevoerd en een trigger op statement niveau slechts 1 keer. Voor de eenvoudigheid zullen we altijd een trigger op rijniveau definiëren.
- Na het `EXECUTE PROCEDURE` sleutelwoord wordt de naam van de trigger functie gegeven die uitgevoerd moet worden.

4.3 De trigger `check_overlapping_registration`

Als voorbeeld zullen we een trigger implementeren die checkt of bij het toevoegen van een registratie aan de relatie `registratieformulier` er geen andere registratie bestaat voor dezelfde nummerplaat met een overlappende periode. Indien dit wel het geval is moet het databanksysteem een foutmelding genereren.
Vooraleer we overgaan naar de implementatie, willen we even nadenken over wat ‘overlappende periode’ nu juist betekent. Let op, we werken op het niveau van dagen en we moeten rekening houden met de beperkingen die reeds worden opgelegd

door de kandidaatsleutel(s) van de tabel `registratieformulier`. De sleutel dwingt namelijk af dat er voor eenzelfde nummerplaat, een registratie kan bestaan die begint en eindigt op dezelfde dag en er ook een nieuwe registratie begint op diezelfde (maar deze dan **na** deze dag moet eindigen). Dit moet ten allen tijde blijven gelden. Periodes kunnen op verschillende manieren met elkaar overlappen. De verschillende opties voor overlap zijn hieronder opgesomd[5]. Met `NEW` wordt er verwezen naar de data van de rij(en) die je wil toevoegen aan de tabel `registratieformulier`. Met `r` wordt er verwezen naar de rijen die reeds in de tabel `registratieformulier` zaten vooraleer de `INSERT` plaatsvond en waarvoor de nummerplaat gelijk is aan `NEW.nummerplaat`.

- De nieuwe registratie zit vervat in een lopende periode: `r.periode_begin` ≤ `NEW.periode_begin` < `r.periode_end` en `r.periode_begin` < `NEW.periode_end` ≤ `r.periode_end`
- De nieuwe registratie begint binnen een lopende periode en eindigt na deze lopende periode: `r.periode_begin` ≤ `NEW.periode_begin` < `r.periode_end` en `NEW.periode_end` > `r.periode_end`
- De nieuwe registratie eindigt binnen een lopende periode en begint voor deze lopende periode: `r.periode_begin` < `NEW.periode_end` ≤ `r.periode_end` en `NEW.periode_begin` < `r.periode_begin`
- De nieuwe registratie omvat een lopende periode: `NEW.periode_begin` < `r.periode_begin` ≤ `r.periode_end` < `NEW.periode_end`

De code[6] voor de implementatie van de trigger die checkt of er overlap is tussen registraties voor eenzelfde nummerplaat is te vinden op Ufora, in de map van deze workshop in een script met de naam `check_overlapping_registration.sql`. Als je de functie `check_overlapping_registration` bekijkt, zie je dat er vier queries worden uitgevoerd die checken of er rijen bestaan in de tabel `registratieformulier` die voldoen aan een van bovenstaande voorwaarden. Het aantal rijen dat voldoen wordt geteld en deze waarde wordt opgeslagen in een van de vier variabelen. Indien een van de variabelen een waarde heeft die groter is dan 0 wordt er een foutmelding opgegooid en wordt de toevoegoperatie afgebroken. Je ziet dat je op dezelfde manier als hierboven, met het `NEW` sleutelwoord kan verwijzen naar de rij(en) die je zal toevoegen. De trigger met de naam `check_overlapping_registration_trigger` zal steeds bij het toevoegen van data aan de tabel `registratieformulier` de functie `check_overlapping_registration` oproepen als check.

[5] Aangezien we op het niveau van dagen werken is de logica redelijk complex. Het doel van deze workshop is het leren implementeren van triggers. Daarom raden we aan om niet te lang stil te staan bij de verschillende mogelijkheden.

[6] Dit is opnieuw niet noodzakelijk de meest performante code.

Implementeer de trigger `check_overlapping_registration_trigger` in de `rollsrobin` databank. Zorg ervoor dat de trigger zo efficiënt mogelijk is (kopieer dus niet gewoon de code), maar wel nog steeds dezelfde voorwaarden controleert. Test of de beperkingen gedefinieerd in de trigger functie opgevangen worden door PostgreSQL door te proberen om ongeldige data toe te voegen. Verwijder hierna alle data opnieuw.

Een belangrijke opmerking is dat we in dit vak enkel redeneren over complexe beperkingen in de vorm van triggers met betrekking tot het **toevoegen** van data. Het aanpassen en verwijderen van data in de vorm van triggers moeten jullie niet in rekening brengen. Dit geldt voor zowel deze workshop alsook voor het project. Wees er jullie echter wel bewust van dat ook aanpassingen aan data of het verwijderen van data gepaard kunnen gaan met gewenst gedrag dat via een trigger moet worden opgevangen!

5 Views

We sluiten het fysieke databankontwerp af met het introduceren van views. Een view is eigenlijk niks anders dan een `SELECT`-query die je een naam hebt gegeven om hier later naar te kunnen verwijzen. Dit lijkt misschien in eerste instantie overbodig, maar een view heeft toch heel wat voordelen.

- Indien je een complexe query opbouwt, kan je (een deel van) de query vereenvoudigen door er een view voor te definiëren.
- Een view zal altijd up-to-date en consistent zijn met de onderliggende data. Dit zorgt er ook voor dat de berekening van **afgeleide attributen** vaak wordt gedaan met behulp van een view.
- Je hebt de mogelijkheid om permissies op niveau van views te definiëren. Zo kan je er bijvoorbeeld voor zorgen dat een gebruiker geen toegang heeft tot een tabel waarin gevoelige gegevens in worden opgeslagen, maar dat hij/zij wel toegang heeft tot een view waarbij maar een beperkt aantal kolommen worden geselecteerd uit de basisrelatie.

Belangrijk om op te merken is dat een view er niet voor zorgt dat de data die de aan de view gekoppelde query teruggeeft, fysiek opgeslagen zal worden[7]. Elke keer je de view oproept, zal de onderliggende query uitgevoerd worden en de data

[7] Behalve in het geval van `materialized` views.

dus opnieuw berekend worden. Het resultaat zal bestaan uit verwijzingen naar de fysieke data.

5.1 Viewdefinitie

Om een view aan te maken kan je gebruik maken van onderstaand commando.

```
CREATE VIEW view_name AS query;
```

Hierin vervang je `view_name` door een view-naam naar keuze en `query` door om het even welke correcte `SELECT`-query.

5.2 De view `totalcost`

Als voorbeeld gaan we een view definiëren die voor elke registratie de totale prijs die de klant moet betalen voor de registratie berekent. Dit zou je kunnen zien als een afgeleid attribuut van de tabel `registratieformulier`. De totale prijs van een registratie is gelijk aan de prijs van de wagen die een klant huurt vermenigvuldigd met het aantal dagen dat deze klant de wagen huurt. We kiezen `totalcost` als naam van deze view.

> Definieer de view `totalcost` die voor elke registratie de totale prijs die de klant moet betalen voor de registratie berekent.

Eenmaal een view is gedefinieerd kan je hiermee werken zoals een fysieke tabel. Om alle data die de view teruggeeft op te vragen kan je een eenvoudig `SELECT` statement schrijven van de vorm

```
SELECT * FROM view_name;
```

> Test of de data die de aan de view gekoppelde query teruggeeft, correct is.

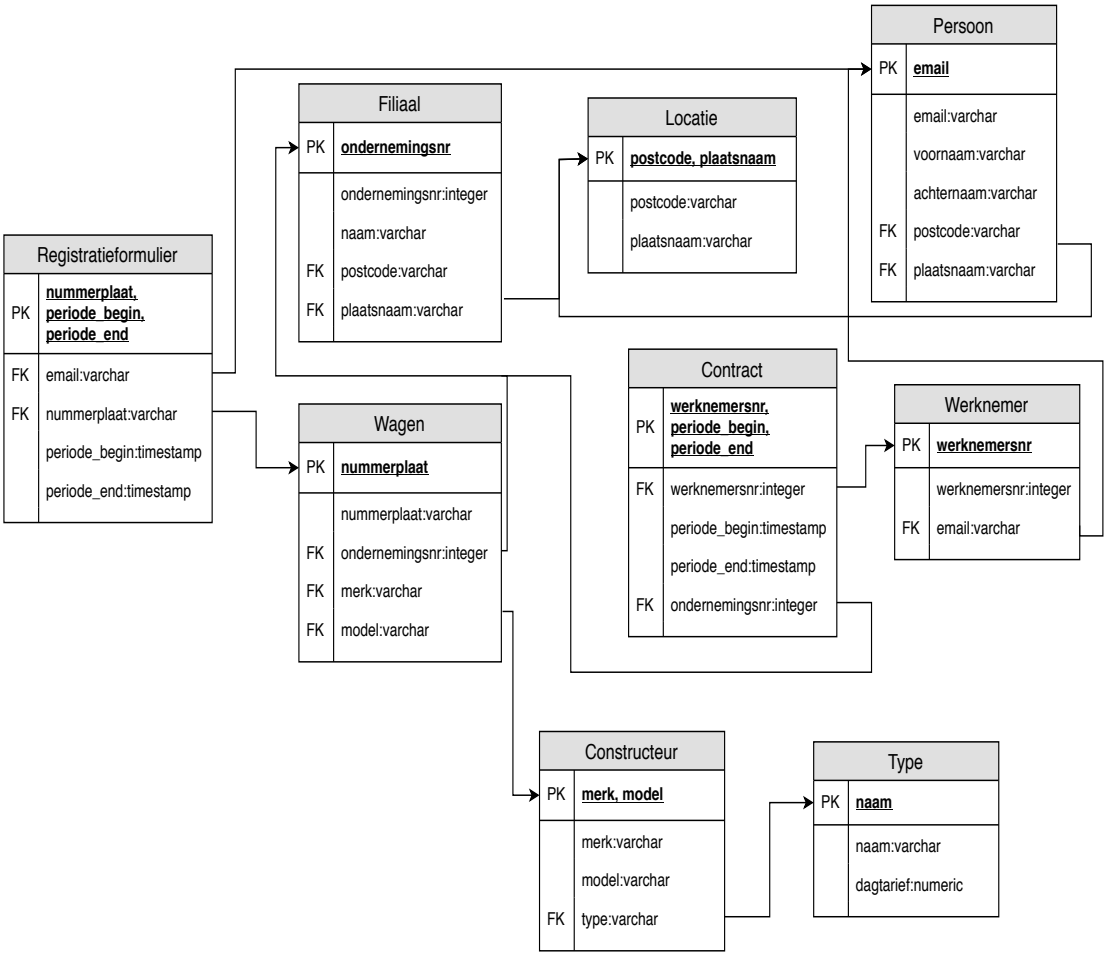
6 Tot slot

Implementeer de volledige `rollsrobin` databank fysiek. Voeg opnieuw alle data toe zoals gezien tijdens de workshop 'Data importeren'. Check of **alle vereisten** uit de informatievergaringsfase effectief geïmplementeerd zijn. Test grondig of **alle beperkingen** opgevangen worden door te proberen om ongeldige data toe te voegen. Maak via **pg_dump** een schema-backup (zonder data) van de **rollsrobin** databank waarin de database **niet** opnieuw wordt aangemaakt.

Indien je bovenstaande opdracht afgehandeld hebt, eindig je eindelijk met een volledige `rollsrobin` databank, die je als voorbeeld voor het project kan gebruiken. Ook de oplossingen in de vorm van SQL-scripts kunnen dienen als voorbeeld voor de scripts die jullie voor dit project moeten indienen. Na deze workshop zullen wij deze scripts bundelen in de map van deze workshop op Ufora.

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 weergegeven worden in dit schema worden onderaan opgelijst.



Extra beperkingen

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

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