Universiteit Gent - TELIN - DDCM
2023-2024

Workshop: Fysiek ontwerp 2

1 Introductie

Tijdens de workshop ‘Fysiek ontwerp 1’ werden alle basiscomponenten van een relationele databank (tabellen, attributen, primaire en vreemde sleutels, `CHECK`-, `UNIQUE`- en `NOT NULL`-beperkingen...) in detail geïntroduceerd. Ook leerde je hoe je deze basiscomponenten fysiek kan implementeren. Het eindresultaat van deze workshop bestond uit een werkende databank die gevuld kan worden met data (zie workshop ‘Data importeren’) en die gebruikt kan worden door er manipulaties (zie later in de workshop ‘Datamanipulatie’) en analyses (zie SQL-lessen) op uit te voeren.
Waarschijnlijk zal je reeds opgemerkt hebben dat deze *basiscomponenten* niet voldoende zijn om alle gewenste functionaliteit te voorzien. Daarom keren we in deze workshop even terug naar het fysiek ontwerp en gaan we in detail in op de meer *geavanceerde* componenten van een relationele databank, genaamd functies, triggers en views. Deze componenten stellen ons, samen met de basiscomponenten, in staat om effectief alle gewenste functionaliteit in de databanklaag te implementeren.
Tenslotte willen we nog meegeven dat we opnieuw zullen werken met de `voetbal` databank die reeds tijdens de vorige oefeningenlessen uitgebreid geïntroduceerd en gebruikt werd. Voor de volledigheid is het relationele databankschema van de `voetbal` databank terug te vinden in de Appendix van deze workshop. Ook vinden jullie op Ufora (in de map van de workshop ‘Fysiek ontwerp 1’) een SQL-script dat alle statements bevat voor de (basis) fysieke implementatie van deze databank. Zo kunnen jullie met een correcte databankimplementatie starten aan deze workshop, hoewel je meer zal leren als je deze workshop maakt vertrekkende van je eigen oplossing.

2 Eerste stappen

Deze sectie dien je enkel door te nemen indien je nog niet (volledig) klaar bent met de vorige workshops of indien je wil vertrekken van onze modeloplossing. Indien je hier meer dan 15 minuten aan spendeert, vraag dan hulp aan een van de assistenten.

Vooraleer we uitgebreid kennis zullen maken met functies, triggers en views, komen we eerst nog even kort terug op een aantal zaken uit de vorige workshops.
Ten eerste willen we jullie er aan herinneren dat alle informatie over het downloaden en installeren van PostgreSQL en alle gerelateerde tools op jullie eigen PC is samengevat in een handleiding op Ufora. Vooraleer jullie starten met deze workshop is het aangeraden om deze handleiding na te lezen, en PostgreSQL en pgAdmin 4 te installeren indien je dit nog niet eerder deed. Als opstart is er namelijk een PostgreSQL databank, waarin alle basiscomponenten van de `voetbal` databank fysiek geïmplementeerd zijn, of een backup in de vorm van een SQL-script van deze databank nodig. Het is dus niet noodzakelijk om een `voetbal` databank te hebben die reeds gevuld is met data. Meerbepaald zouden jullie de workshops 'Inleiding PostgreSQL' en 'Fysiek ontwerp 1' reeds gemaakt moeten hebben. Mocht dit niet het geval zijn, kunnen jullie een volledig backup-script (met naam `voetbal_basic.sql`) terugvinden in de map van de workshop 'Fysiek ontwerp 1' op Ufora. Dit script kan je gebruiken als voorbeeldoplossing van het (basis) fysiek ontwerp van de `voetbal` databank. Indien jullie dit script wensen te gebruiken, is het aangeraden om de oplossing in detail door te nemen vooraleer verder te gaan en nog eens na te gaan of je alle concepten uit de vorige workshops goed begrepen hebt.
Een fysieke implementatie van de `voetbal` databank bekomen jullie door deze backup in te laden (restore). Dit kan je doen door middel van de commandolijn-applicatie `psql` met het volgende commando (plaats het volledige commando op 1 lijn).

```
psql
    --host=127.0.0.1
    --port=5432
    --dbname=voetbal
    --username=postgres
    --file=voetbal_basic.sql
```

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

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

Wanneer jullie kijken naar de huidige `voetbal` databank zullen jullie inderdaad merken dat alle basisrelaties (tabellen), attributen (kolommen), primaire en vreemde sleutels, `CHECK`-, `UNIQUE`- en `NOT NULL`-beperkingen reeds geïmplementeerd zijn. Bij het vergelijken van het relationeel databankschema van deze databank (zie Appendix) met het huidige fysiek ontwerp zullen jullie echter merken dat er momenteel nog drie beperkingen niet geïmplementeerd zijn. Dit zijn

1. het aantal toeschouwers die een wedstrijd bijwonen kan de capaciteit van het stadion van de thuisclub niet overschrijden,
2. een club kan slechts 1 wedstrijd per datum spelen, en
3. het totaal aantal doelpunten dat gelieerd is aan een wedstrijd kan niet groter zijn dan de som van de scores van de thuis- en uitclub op het einde van deze wedstrijd.

In tegenstelling tot de beperkingen die we reeds geïmplementeerd hebben, zijn er, wanneer je wil verifiëren of data voldoen aan deze beperkingen, (andere) data nodig die opgeslagen worden in verschillende tabellen of in verschillende rijen van een tabel. Tijdens de ontwerpsfase van deze databank is er reeds nagedacht over het gewenst gedrag met betrekking tot deze beperkingen bij het toevoegen van data in de corresponderende tabellen (respectievelijk `wedstrijd` en `doelpunt`). Tijd om deze beperkingen nu ook fysiek te gaan implementeren.

3 Functies

Tot nog toe was het mogelijk om bij het fysiek ontwerp van een databank alle beperkingen te definiëren bij het aanmaken van de tabellen. Om, bijvoorbeeld, af te dwingen dat de capaciteit van elk stadion strikt positief moet zijn, konden we eenvoudigweg een `CHECK`-beperking definiëren die inwerkt op het attribuut `capaciteit` in de tabel `stadion`. `CHECK`-beperkingen volstaan echter niet meer op het moment dat er meer geavanceerde beperkingen afgedwongen moeten worden op data. Typisch zijn dit beperkingen waarvoor (i) data uit andere tabellen, of (ii) data uit andere rijen van dezelfde tabel nodig zijn om te controleren of ze effectief voldaan zijn. We hebben dus extra functionaliteit nodig om ervoor te zorgen dat er op geen enkele manier ongeldige data in de databank terechtkomen. PostgreSQL voorziet (net als elke procedurele programmeertaal, zoals C, Pascal. . .) deze extra functionaliteit in de zin van gebruikersgedefinieerde functies. Binnen deze functies is het mogelijk om heel wat procedurele elementen (bv. 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[1] taal, die nauw verwant is met SQL.
Merk op dat je in principe ook alle (eenvoudige en geavanceerde) beperkingen zou kunnen implementeren in de applicatielaag van software die gebruik maakt van een databank in plaats van in de databanklaag van deze software. In dat geval zou je kunnen controleren of data voldoen aan deze beperkingen vooraleer ze naar de databank weggeschreven worden (bv. bij het creëren van data of in het programma dat data wegschrijft naar de databank). Het definiëren van beperkingen en functies binnen de databanklaag zelf brengt echter heel wat voordelen met zich mee. Hieronder zijn een aantal van deze voordelen opgesomd.

- Een afname van het aantal oproepen vanuit de applicatie(laag) naar de databankserver, waardoor de performantie van de applicatie sterk kan verbeteren. De applicatie kan vertrouwen op de goede werking van de databank wanneer er data naartoe worden geschreven.
- De herbruikbaarheid van eenzelfde databank over meerdere applicaties (die mogelijks in verschillende programmeertalen geschreven zijn). Het is inderdaad beter om alles slechts één keer te implementeren in de databank in plaats van de beperkingen opnieuw te moeten implementeren in elke applicatie die deze databank wil gebruiken.
- 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).

We raden dus aan om beperkingen en functies steeds in de databanklaag te definiëren, indien je hiertoe de mogelijkheid hebt.
In Sectie 3.1 en 3.2 gaan we dieper in op de definitie van functies in PostgreSQL en implementeren we als voorbeeld een eenvoudige tellerfunctie. In Sectie 4 kijken we dan hoe we (de uitvoering van) functies kunnen linken aan bepaalde gebeurtenissen in de databank door middel van triggers. Let op, we gaan niet in detail in op alle componenten (en bijhorende syntax) die gebruikt kunnen worden bij het implementeren van functies. Voor verdere problemen, vragen of opmerkingen kunnen jullie steeds terecht op `https://www.postgresqltutorial.com/`, in de documentatie van PostgreSQL[2] of bij de assistenten.

3.1 Functiedefinitie

Om een functie te definiëren maak je typisch gebruik van een `CREATE FUNCTION` statement[3]. Dit statement heeft standaard de volgende syntax.

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

```
CREATE FUNCTION functienaam(p1 datatype,...,pn datatype)
    RETURNS datatype AS
$BODY$
DECLARE
    l1 datatype;
    ...
    lm datatype;
BEGIN
    --functielogica
END;
$BODY$
LANGUAGE programmeertaal;
```

De verschillende onderdelen die een functie (mogelijks) bevat zijn hieronder opgelijst.

- De naam van de functie (naar keuze) wordt opgegeven na het sleutelwoord `CREATE FUNCTION`.
- Na de naam volgt er, tussen haakjes, een door komma's gescheiden lijst van parameters die je wil meegeven aan de functie. Elke parameter heeft een (unieke) naam en een datatype.
- Na het sleutelwoord `RETURNS` wordt het datatype opgegeven van de waarde die door de functie teruggegeven zal worden.
- De lokale variabelen (elk met een naam die uniek is binnen de functie en een bijhorend datatype) die gebruikt worden in de functie worden opgelijst na het `DECLARE` sleutelwoord.
- Tussen het `BEGIN` en `END` sleutelwoord volgt de eigenlijke functielogica.
- Het `CREATE FUNCTION` statement wordt afgesloten met de definitie van de procedurele programmeertaal die gebruikt wordt om de logica te implementeren. 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 een functie. De reden hiervoor is dat, wanneer je PL/pgSQL gebruikt om de logica te implementeren, deze inhoud als een string moet worden doorgegeven aan het `CREATE FUNCTION` statement. Aangezien het gebruik van enkele aanhalingstekens in deze inhoud dan voor problemen kan zorgen, wordt het dubbele dollarteken als vervanger voor een enkel aanhalingsteken gebruikt. Tussen de dollartekens is het mogelijk om een naam (optioneel en naar keuze) mee te geven. In dit geval is er gekozen voor de naam `BODY`.

3.2 De functie `increment`

Als voorbeeld zullen we een eenvoudige functie implementeren waarin een opgegeven waarde verhoogd wordt met 1, het resultaat wordt uitgeprint als info-bericht en deze waarde ook wordt teruggegeven als 'return value'. De code voor deze functie wordt hieronder gegeven[4].

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

Deze functie verwacht dus, als input, een enkele parameter met naam `value` van het datatype `integer`. Er zijn 2 lokale variabelen: een variabele met naam `result` die het resultaat zal bevatten en een variabele met naam `incrementer` die de waarde bevat om op te tellen bij de waarde van parameter `value`. Belangrijk om op te merken is dat `incrementer` gedefinieerd is als een constante en de waarde hiervan niet meer kan veranderen van zodra er een waarde aan deze parameter is toegekend. Tijdens uitvoering van de functie wordt de waarde van de variabele `result` uitgeprint in een info-bericht (`RAISE INFO ...;`) en op het einde wordt deze waarde teruggegeven als 'return value' (`RETURN result;`).

Implementeer de functie met naam `increment` binnen de `voetbal` databank. Je kan dit doen door een query tool te openen binnen deze databank, de code hierin te typen en uit te voeren.

Indien de functie met naam `increment` succesvol werd geïmplementeerd, zou je een bericht moeten zien dat aangeeft dat het `CREATE FUNCTION` statement zonder problemen is afgerond. De functie kan nu worden opgeroepen.

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

3.3 Een functie oproepen

Een gebruikersgedefinieerde functie met naam `functienaam` en waarden $v_1,...,v_n$ voor de gegeven parameters $p_1,...,p_n$ kan je, net als ingebouwde functies, als volgt oproepen.

```
SELECT functienaam(v1,...,vn);
```

Indien deze query succesvol werd uitgevoerd, zou je een resultatentabel moeten zien.

> Controleer of de functie met naam `increment` succesvol uitvoert en het verwachte resultaat teruggeeft door de functie met een aantal verschillende parameterwaarden op te roepen. Zie je ook het verwachte info-bericht onder 'messages' in pgAdmin 4?

4 Triggers

Hierboven leerde je hoe je in PostgreSQL functies kan definiëren. Functies alleen volstaan echter niet om (complexe) beperkingen te leggen op data in een databank. Ze stellen ons wel in staat om bepaalde acties uit te voeren in een databank, maar we willen nu eenmaal ook kunnen vastleggen wanneer deze acties precies moeten worden uitgevoerd. Om dit probleem op te lossen introduceren we in deze sectie triggers. Een trigger voegt, net als een functie, ook bepaalde gebruikersgedefinieerde functionaliteit toe. Meer nog, een trigger maakt zelfs gebruik van (trigger)functies. Het grote verschil met gewone functies is dat triggerfuncties enkel opgeroepen worden indien er zich een bepaalde gebeurtenis (die vooraf vastgelegd is) voordoet in de tabel waaraan de trigger gekoppeld is. Indien er zich zo een gebeurtenis voordoet, zal een trigger de bijhorende functie automatisch uitvoeren.
Triggers kunnen in verschillende gevallen nuttig zijn. Neem, bijvoorbeeld, een databank die door meerdere applicaties en/of gebruikers in parallel kan worden gebruikt. In zo'n context zou het handig kunnen zijn om een geschiedenis (log) bij te houden van welke operaties er door bepaalde applicaties/gebruikers zijn uitgevoerd. Je zou in dit geval een trigger kunnen definiëren die ervoor zorgt dat alle operaties die worden uitgevoerd door een bepaalde applicatie/gebruiker weggeschreven worden naar een extern bestand. Daarnaast kunnen triggers ook gebruikt worden om te controleren of data in een databank steeds voldoen aan een verzameling (complexe) integriteitsregels (of beperkingen), ook na het toevoegen, aanpassen of verwijderen van (andere) data. En laat dit tweede geval nu exact zijn wat we nodig hebben.

4.1 Triggerfuncties

Als je een trigger wil implementeren, is het belangrijk om eerst de bijhorende triggerfunctie aan te maken. De syntax om dit te doen is zeer gelijkaardig aan de syntax om een gewone functie aan te maken (zie Sectie 3.1). De enige verschillen zijn dat een triggerfunctie nooit parameters zal hebben en altijd een waarde van het datatype `trigger` zal teruggeven. Hieronder wordt, voor de volledigheid, de pseudocode gegeven waarmee je een triggerfunctie kan aanmaken.

```
CREATE FUNCTION functienaam
    RETURNS trigger AS
...
LANGUAGE programmeertaal;
```

4.2 Triggerdefinitie

Het definiëren van een trigger gebeurt door middel van het `CREATE TRIGGER` statement[5]. De (basis)syntax van dit statement is de volgende[6].

```
CREATE TRIGGER triggernaam
    {BEFORE | AFTER} {gebeurtenis [OR ...]} ON basisrelatie
    [FOR [EACH] {ROW | STATEMENT}] EXECUTE PROCEDURE functienaam;
```

De verschillende onderdelen die een trigger (mogelijks) bevat zijn hieronder opgelijst.

- De naam van de trigger (naar keuze) wordt opgegeven na het `CREATE TRIGGER` statement.
- Een triggerfunctie kan opgeroepen worden voor (`BEFORE`) of na (`AFTER`) een gebeurtenis. In het geval een triggerfunctie opgeroepen wordt voor een gebeurtenis kan het mogelijks de gebeurtenis nog tegenhouden (wat de performantie ten goede kan komen). Indien een triggerfunctie opgeroepen wordt na een gebeurtenis kan het de aanpassingen die gebeurd zijn in de databank tijdens die gebeurtenis meteen gebruiken. Let wel op, de keuze om een triggerfunctie op te roepen voor of na een gebeurtenis kan impact hebben op de functielogica.
- `gebeurtenis` definieert de gebeurtenis die ervoor zal zorgen dat de triggerfunctie automatisch opgeroepen wordt. Mogelijke gebeurtenissen zijn `INSERT` (toevoeging van data), `UPDATE` (aanpassing van data), `DELETE` (verwijderen van data) of `TRUNCATE` (leegmaken van een volledige tabel). Het is ook mogelijk om meerdere gebeurtenissen op te geven door middel van concatenatie met `OR`.

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

[6] De vierkante haakjes, verticale strepen en accolades zijn geen deel van de syntax maar duiden optionele onderdelen en keuzes aan.

- `basisrelatie` geeft de naam van de basisrelatie waarin zich de gebeurtenis moet voordoen zodat de triggerfunctie automatisch wordt uitgevoerd.
- `[FOR [EACH] {ROW | STATEMENT}]` bepaalt of de trigger op rijniveau of op statementniveau uitgevoerd moet worden. Stel dat je bijvoorbeeld een aanpassing doet aan 20 rijen met 1 statement, dan zal een trigger op rijniveau 20 keer worden uitgevoerd en een trigger op statementniveau slechts 1 keer. Voor de eenvoudigheid zullen we in dit vak steeds een trigger op rijniveau definiëren.
- Na het `EXECUTE PROCEDURE` sleutelwoord wordt de naam van de bijhorende triggerfunctie die uitgevoerd moet worden meegegeven.

4.3 De trigger `check_doelpunten_trigger`

Als voorbeeld zullen we een trigger implementeren die, bij het toevoegen van een doelpunt aan de basisrelatie `doelpunt`, het totaal aantal doelpunten dat gelieerd is aan de wedstrijd waarvoor het doelpunt wordt toegevoegd controleert. Meerbepaald dient dit aantal steeds kleiner dan of gelijk aan de som van het aantal thuis- en uitdoelpunten dat gelieerd is aan deze wedstrijd te zijn. Indien dit niet het geval is, moet het databanksysteem een foutmelding genereren en de toevoegoperatie afbreken[7].
Vooraleer we overgaan naar de implementatie, willen we eerst even nadenken over hoe we dit nu best kunnen aanpakken. Ten eerste hebben we het aantal thuisdoelpunten en uitdoelpunten dat gelieerd is aan de wedstrijd waarvoor er een extra doelpunt toegevoegd zal worden nodig. Deze waarden kunnen eenvoudig opgevraagd worden door een `SELECT`-query uit te voeren die inwerkt op de basisrelatie `wedstrijd`. Het resultaat van deze opvraging kan opgeslagen worden in twee lokale variabelen met naam `score_thuis` en `score_uit`. Ten tweede willen we ook weten hoeveel doelpunten er reeds zijn toegevoegd voor deze wedstrijd. Deze waarde kan dan weer eenvoudig opgevraagd worden door een `SELECT`-query uit te voeren die inwerkt op de basisrelatie `doelpunt`. Het resultaat hiervan kan op zijn beurt opgeslagen worden in een lokale variabele met naam `huidig_aantal_doelpunten`. Tot slot kunnen we controleren of de som van het aantal thuis- en uitdoelpunten (`score_thuis + score_uit`) strikt groter is dan de waarde van de variabele `huidig_aantal_doelpunten`. Indien dit niet het geval is, zorgt de toevoeging van een extra doelpunt ervoor dat de data niet meer voldoen aan de gestelde beperking. Idealiter moet PostgreSQL dan een foutmelding opwerpen en moet de toevoegoperatie worden afgebroken. Indien dit wel het geval is, moet de nieuwe rij worden toegevoegd.

[7] In principe zou ook het bijhorende wedstrijdevent verwijderd moeten worden, maar voor de eenvoudigheid volstaat het hier om enkel het toevoegen van foutieve data tegen te houden.

De code[8] voor de implementatie van de trigger die gebruikt kan worden ter verificatie van deze beperking is terug te vinden op Ufora in de map van deze workshop, meerbepaald in een script met naam `check_doelpunten.sql`. De definitie van deze trigger (met naam `check_doelpunten_trigger`) stelt dat de bijhorende triggerfunctie (met naam `check_doelpunten`) steeds uitgevoerd zal worden *voor* het toevoegen van een rij aan de tabel `doelpunt`. Als je de triggerfunctie `check_doelpunten` bekijkt, zie je dat de aanpak die we hierboven hebben voorgesteld nauwgezet wordt gevolgd. Belangrijk om op te merken is dat het laatste statement `RETURN NEW;` ervoor zorgt dat de rij die toegevoegd zou moeten worden ook effectief wordt toegevoegd indien er eerder geen foutmelding werd opgeworpen door PostgreSQL. Vergeet niet om dit statement op het einde van elke triggerfunctie te plaatsen, anders zal er nooit een rij worden toegevoegd. Daarnaast kan je zien dat er naar de rij, waarvoor momenteel de triggerfunctie wordt uitgevoerd, verwezen kan worden met het sleutelwoord `NEW`. Dit sleutelwoord moet je dus zien als een verwijzing naar de rij die als volgende zal worden toegevoegd. Deze verwijzing bevat voor elk attribuut uit de tabel `doelpunt` een waarde die ook aangenomen wordt door de toe te voegen rij. Deze waarde kan je eenvoudig opvragen door middel van het sleutelwoord `NEW.<attribuut>`, waarbij je `<attribuut>` vervangt door de naam van het attribuut. Stel, bijvoorbeeld, dat de triggerfunctie uitgevoerd wordt voor toevoeging van een doelpunt met volgende data.

- thuisclub: AA Gent
- datum: 2023-10-25
- nr: 1
- beschrijving: kopbal
- spelerid: 15

In dit geval zal de variabele `NEW.thuisclub` de waarde 'AA Gent' bevatten en zal de variabele `NEW.spelerid` de waarde 15 bevatten.

> Bekijk en interpreteer nauwgezet de triggercode die gegeven is in het script met naam `check_doelpunten.sql`. Test of deze trigger effectief correct is geïmplementeerd door te proberen om ongeldige data toe te voegen aan de tabel `doelpunt`. Verwijder hierna alle toegevoegde data opnieuw.

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

Implementeer een trigger en bijhorende triggerfunctie ter verificatie van de beperking die stelt dat het aantal toeschouwers die een wedstrijd bijwonen de capaciteit van het stadion van de thuisclub niet kan overschrijden. Test opnieuw of de trigger werkt zoals verwacht.

Implementeer een trigger en bijhorende triggerfunctie ter verificatie van de beperking die stelt dat een club slechts 1 wedstrijd per datum kan spelen. Test opnieuw of de trigger werkt zoals verwacht.

Een belangrijke opmerking is dat we in dit vak enkel redeneren over (complexe) beperkingen in de vorm van triggers die uitgevoerd worden voor/na het *toevoegen* van data. Het verifiëren van beperkingen voor/na het aanpassen en verwijderen van data moeten jullie dus niet in rekening brengen. Dit geldt zowel voor deze workshop alsook voor het project. Wees er jullie echter wel van bewust dat ook het aanpassen en verwijderen van data ook gepaard kan gaan met ongewenst gedrag dat via een trigger zou moeten worden opgevangen. Het is, bijvoorbeeld, nog steeds mogelijk om het aantal thuis- of uitdoelpunten dat gelieerd is aan een wedstrijd aan te passen, zodat de hierboven beschreven beperking niet meer voldoet.

5 Views

We sluiten het fysiek databankontwerp af met het introduceren van views. Een view is eigenlijk niks anders dan een `SELECT`-query met een naam. Deze naam kan je, na het aanmaken van de view, gebruiken om herhaaldelijk de bijhorende `SELECT`-query op te roepen zonder dat je de query elke keer opnieuw moet schrijven. Dit lijkt misschien in eerste instantie overbodig, maar een view heeft toch ook heel wat andere voordelen.

- Bij het opbouwen van een complexe `SELECT`-query 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 een view vaak wordt gebruikt voor de berekening van *afgeleide data/attributen*.
- Je hebt de mogelijkheid om privileges te definiëren op niveau van views. Zo kan je er, bijvoorbeeld, voor zorgen dat een gebruiker geen toegang heeft tot

een (volledige) tabel waarin gevoelige gegevens worden opgeslagen, maar dat hij/zij wel toegang heeft tot een view die maar een beperkt aantal kolommen opvraagt uit deze tabel of afgeleide data berekent op basis van de data in deze tabel.

Belangrijk om op te merken is dat een view er niet voor zorgt dat de data die door de bijhorende `SELECT`-query worden opgevraagd, fysiek opgeslagen worden[9]. Elke keer je een view oproept zal deze query uitgevoerd worden en het resultaat opnieuw berekend worden. Dit resultaat zal bestaan uit interne verwijzingen naar de fysieke data.

5.1 Viewdefinitie

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

```
CREATE VIEW viewnaam AS select-query;
```

Hierin vervang je `viewnaam` door een naam voor deze view (naar keuze) en `select-query` door en bijhorende `SELECT`-query.

5.2 De view `winnaar_view`

Als voorbeeld gaan we een view definiëren die, per wedstrijd, aangeeft welke club de wedstrijd heeft gewonnen (of, met andere woorden, welke club meeste goals heeft gescoord) en wat het doelsaldo (absolute verschil tussen het aantal goals gemaakt door de winnende club en de verliezende club) was. We kiezen `winnaar_view` als naam voor deze view en willen dat de resultatentabel minstens bestaat uit de kolommen `thuisclub` (`varchar`), `uitclub` (`varchar`), `datum` (`date`), `winnaar` (`varchar`) en `doelsaldo` (`integer`). Indien beide clubs evenveel goals hebben gescoord, moet de kolom `winnaar` de waarde ‘Gelijkspel’ bevatten en moet de kolom `doelsaldo` de waarde 0 bevatten.

Definieer de view `winnaar_view`.

Eenmaal een view is gedefinieerd kan je hiermee werken zoals een fysieke tabel. Om alle data op te vragen die een view met naam `viewnaam` teruggeeft, kan je een eenvoudige `SELECT`-query schrijven van de vorm

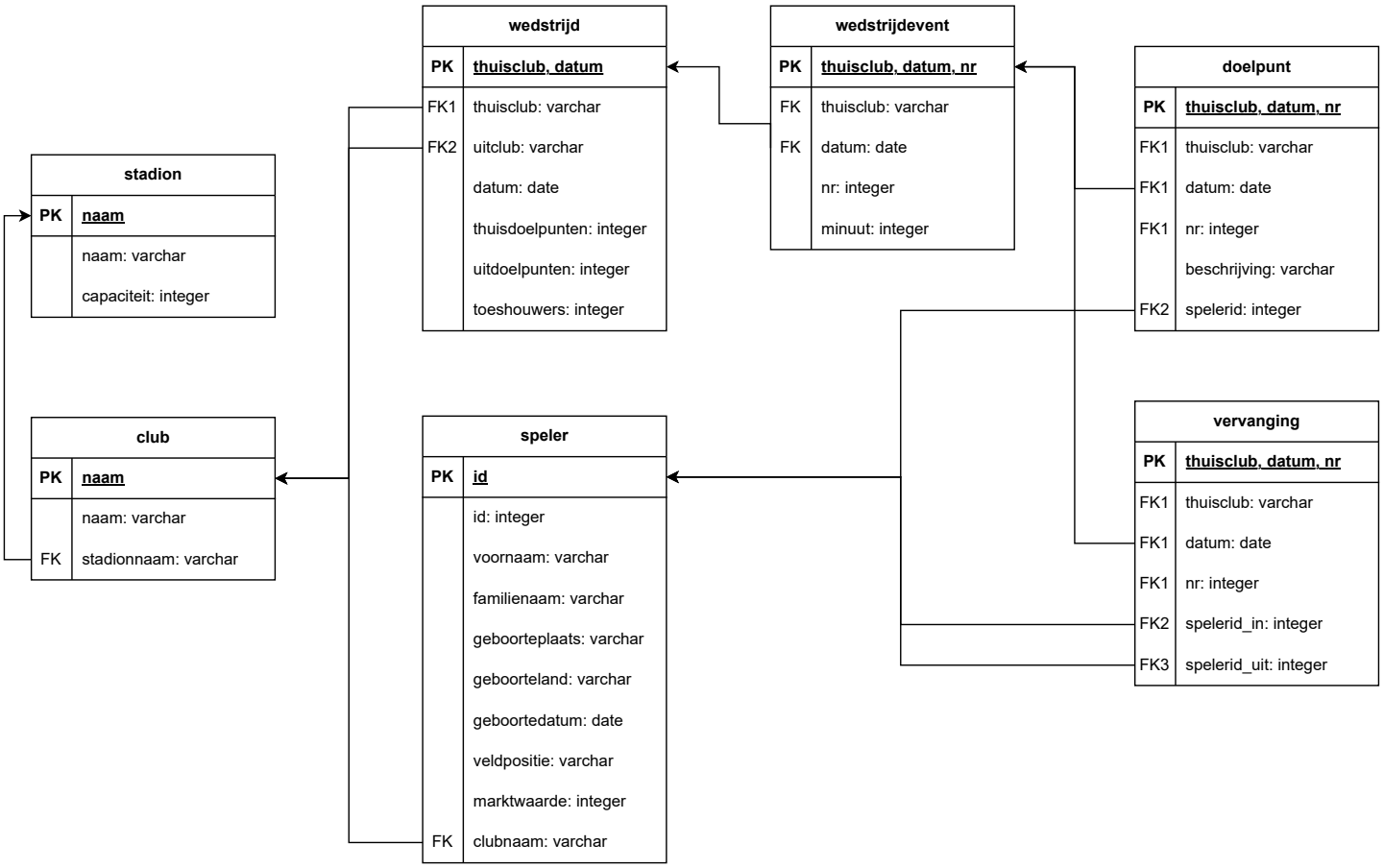
```
SELECT * FROM viewnaam;
```

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

Daarnaast kan je, net als alle andere (fysieke) tabellen, een view ook in eender welke andere (meer complexe) `SELECT`-query gebruiken.

Test of de data die worden teruggegeven door de view met naam `winnaar_view` correct zijn.

Appendix: Relationeel schema `voetbal` databank



In bovenstaande figuur vind je het relationeel databankschema van de `voetbal` databank. Hierbij wordt iedere basisrelatie weergegeven door een rechthoek, die bovendien een oplijsting van alle attributen met bijhorende datatypes bevat. Daarnaast worden de attributen die behoren tot de primaire sleutel (PK) bovenaan weergegeven, en worden vreemde sleutels (FK) voorgesteld door een pijl tussen de betreffende attribuutverzamelingen. Alle extra beperkingen die niet kunnen worden weergegeven in dit schema, worden hieronder opgelijst.

Extra beperkingen

- `stadion`:
  - check: capaciteit > 0
- `speler`:
  - optioneel: voornaam, geboorteplaats, geboorteland, geboortedatum, veldpositie, marktwaarde
  - check: veldpositie ∈ {‘Goalkeeper’, ‘Defender’, ‘Midfield’, ‘Attack’}, marktwaarde ≥ 0
- `wedstrijd`:
  - optioneel: toeschouwers
  - uniek: {uitclub, datum}
  - check: thuisdoelpunten ≥ 0, uitdoelpunten ≥ 0, toeschouwers ≥ 0, thuisclub ≠ uitclub
  - controleer bij toevoeging dat het aantal toeschouwers niet groter is dan de capaciteit van het stadion van de thuisclub
  - controleer bij toevoeging dat een club slechts 1 wedstrijd per datum speelt
- `wedstrijdevent`:
  - check: nr ≥ 1, minuut ≥ 0, minuut ≤ 120
- `doelpunt`:
  - controleer bij toevoeging dat het totaal aantal doelpunten dat gelieerd is aan deze wedstrijd niet groter is dan de som van de scores van de thuis- en uitclub op het einde van deze wedstrijd
- `vervanging`:
  - check: speler_in ≠ speler_uit