Database Development

Als een systeem data voor langere tijd moet opslaan dan zal hiervoor hoogstwaarschijnlijk een database gebruikt worden. Alhoewel moderne applicaties via een Object Relational Mapping (ORM) model het gebruik abstraheren voor de ontwikkelaar is het belangrijk om te begrijpen hoe een database functioneert.

In deze cursus leer je met een database werken, leer je ontwerp modellen overzetten naar database modellen en bestudeer je ook databases die niet een relationeel model gebruiken.

1. Werken met een Database

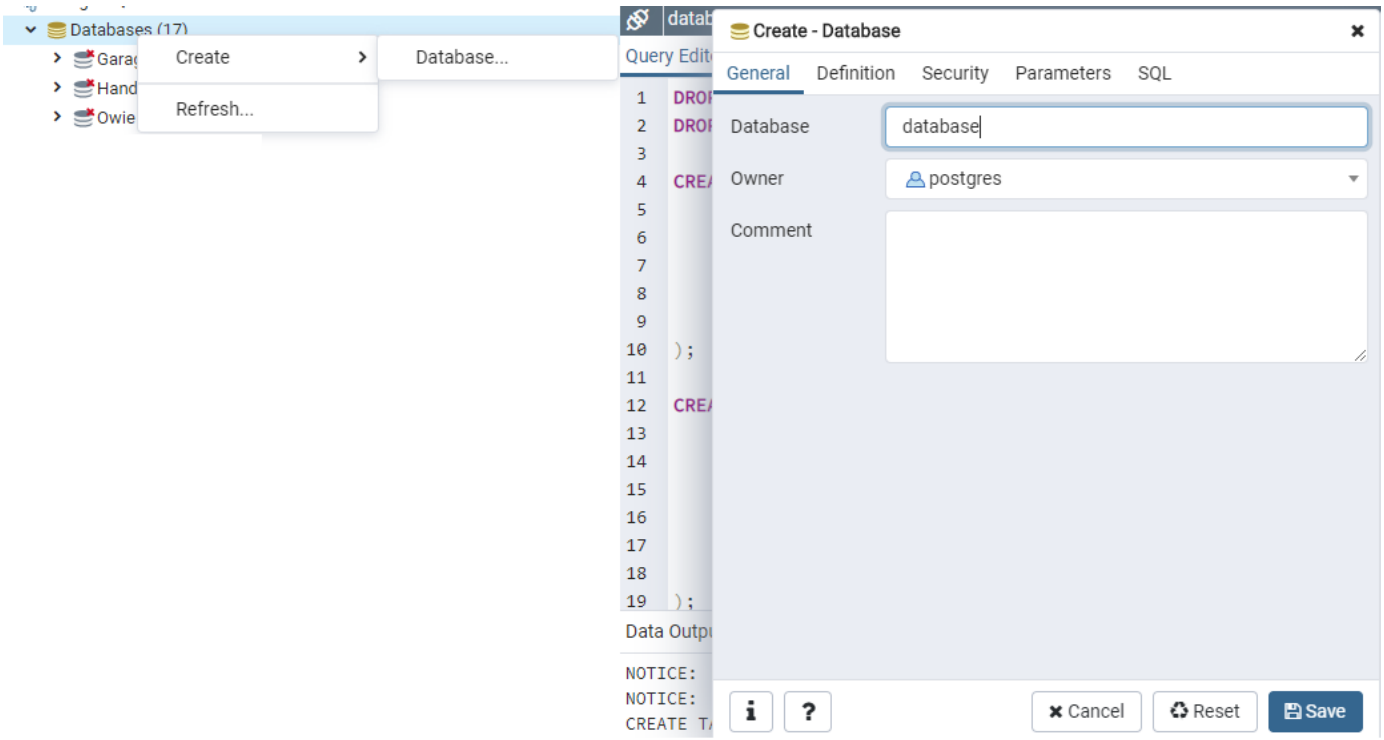
Een database is een geweldig medium om data in op te slaan. Voor nu focussen we op het gebruik van Relationele databases, dat wil zeggen een database waar relaties tussen data in het model worden gedefinieerd.

Om te starten ga je leren hoe je data kan bevragen, updaten en verwijderen. In de voorbeelden wordt er vanuit gegaan dat je deze queries uitvoert op een PostgreSQL (https://www.postgresql.org/) database. Deze kan je op elk gewenst systeem installeren en via het programma pgAdmin (https://www.pgadmin.org/) bevragen.

2. Voorbeeld tabellen

Om te beginnen is het handig om alvast een schema te hebben. Later leer je deze zelf maken, maar voor nu kun je onderstaande kopiëren naar jouw postgresql omgeving en uitvoeren. Deze tabellen zijn een begin van een lesplanningsysteem waarbij docenten, cursussen en studenten aan elkaar gekoppeld worden.

Door dit systeem incrementeel op te bouwen leer je tabellen maken, relaties leggen en data te modelleren. Voor nu is het belangrijk dat deze queries worden uitgevoerd op een PostgreSQL schema. De naam van het schema mag je zelf bepalen, via de `pgAdmin` interface kun je zelf een schema aanmaken.



Kopieer vervolgens onderstaande queries naar het SQL venster en voer deze uit.

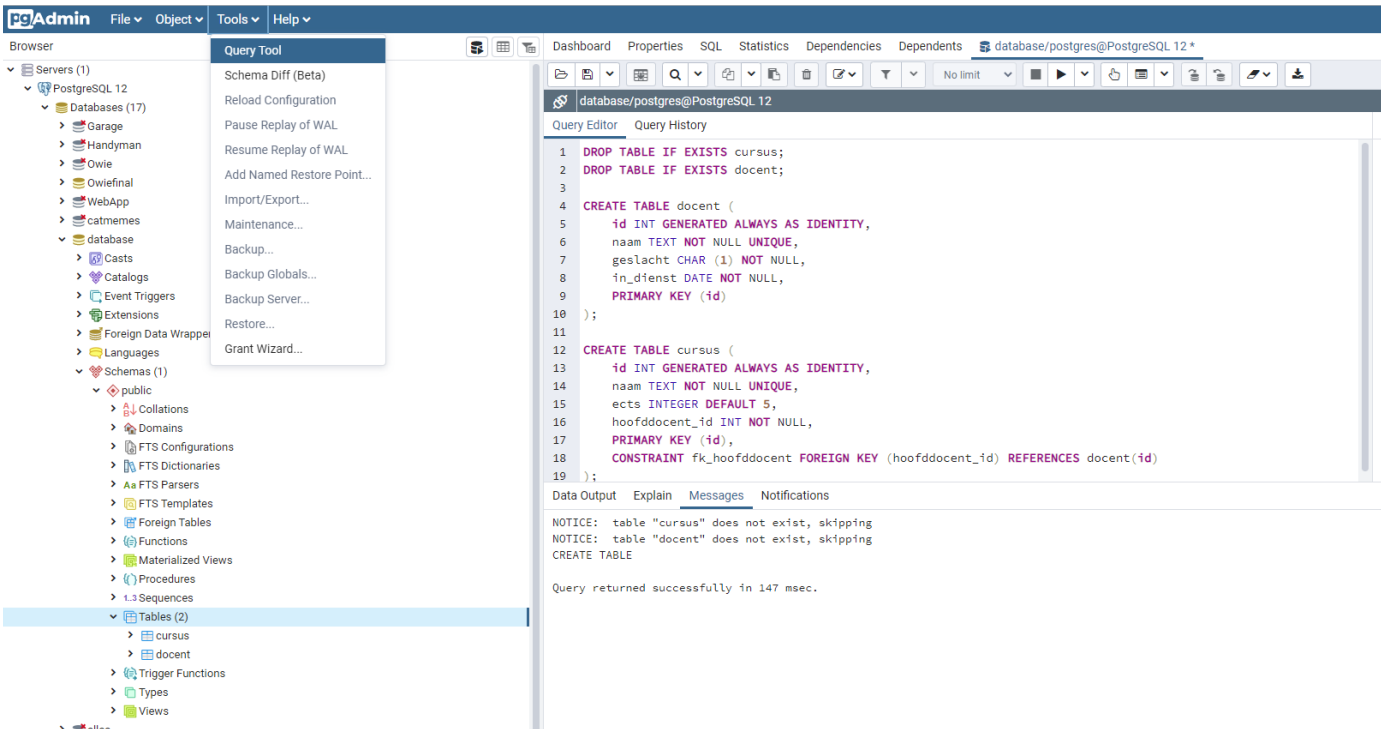
```
DROP TABLE IF EXISTS docent_cursus;
DROP TABLE IF EXISTS cursus;
DROP TABLE IF EXISTS docent;

CREATE TABLE docent (
    id INT GENERATED ALWAYS AS IDENTITY,
    naam TEXT NOT NULL UNIQUE,
    geslacht CHAR (1) NOT NULL,
    in_dienst DATE NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE cursus (
    id INT GENERATED ALWAYS AS IDENTITY,
    naam TEXT NOT NULL UNIQUE,
    ects INTEGER DEFAULT 5,
    hoofddocent_id INT NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT fk_hoofddocent FOREIGN KEY (hoofddocent_id) REFERENCES docent(id)
);

CREATE TABLE docent_cursus (
```

```
    docent_id INT,
    cursus_id INT,
    CONSTRAINT fk_docent FOREIGN KEY (docent_id) REFERENCES docent(id),
    CONSTRAINT fk_cursus FOREIGN KEY (cursus_id) REFERENCES cursus(id)
);
```



Wanneer je bovenstaande queries hebt uitgevoerd, je hoeft ze nu nog niet te begrijpen, zul je zien dat de volgende tabellen zijn aangemaakt in de database:

- cursus
- docent
- docent_cursus

Deze tabellen kun je vinden door links in het menu je database uit te klappen, en vervolgens het volgende klikpad te volgen: Schemas -> public -> Tables. Met deze tabellen ga je leren data in de database te plaatsen, te selecteren, te bewerken en te verwijderen.

2.1 Database Dialecten

Elke database heeft een eigen implementatie van de SQL standaard, SQL 99 / ANSI 99. Dit betekent dat elke database ongeveer dezelfde wijze hanteert van het schrijven van SQL queries, maar dat er wel degelijk verschillen zijn per database.

De verschillen vind je voornamelijk in de data types; bijvoorbeeld de ene database kent het type `TEXT` voor grote stukken tekst en de ander gebruikt `LONGTEXT`. Ook verschillen databases in de manier hoe ze relaties en beperkingen definiëren, bijvoorbeeld hoe een unieke beperking opgelegd kan worden.

Dit is natuurlijk een beetje onhandig als je net begint met databases. Daarom is het van belang om eerst te focussen op 1 database systeem, in dit geval PostgreSQL. Binnen de `Java` wereld is het gewoon om een zogenaamd ORM systeem te gebruiken. Dit kan `jpa` of `hiberante` zijn. De ORM abstraheert de verschillen tussen de databasen weg met een algemene set aan instructies. Tijdens de cursus zul je vanuit jouw code met een ORM gaan werken. Voor nu is dat echter niet van belang en zullen we focussen op het bevragen van de database door middel van *standaard* SQL.

2.2 Data toevoegen

Een database is niets als er geen data in staat. In de vorige sectie heb je tabellen aangemaakt, in deze sectie leer je data toevoegen.

Om data toe te voegen wordt een `INSERT INTO` instructie gegeven. De formele definitie van deze instructie is als volgt:

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
```

Dit ziet er natuurlijk een beetje gek uit. Hierboven is dan ook de `grammer` voor de instructie afgebeeld; de exacte definitie zoals de software van PostgreSQL het zal lezen. Over het algemeen zul je altijd de `grammer` manier van schrijven terugvinden in de documentatie. Laten we kijken wat hier nu eigenlijk staat.

De regel begint met een aantal losse woorden: `INSERT INTO`. Dit betekent dat om gegevens in te voeren de instructie moet beginnen met de woorden `INSERT INTO`. Hoofdletters zijn niet belangrijk, maar de conventie is wel om database woorden in hoofdletters te schrijven en namen van tabellen en kolommen in kleine letters.

Vervolgens staat er `table`. Het woord is in kleine letters en geeft aan dat hier verwacht wordt dat de naam van een tabel, die door de gebruiker is bedacht, wordt ingevoerd. In ons voorbeeld zijn er 2 tabellen waaruit we kunnen kiezen; `docent`, `cursus`.

Na de `table` wordt er iets optioneels verwacht. De `[...]` geven aan dat de waarde tussen de haken optioneel is op deze positie. In dit geval is het optioneel om `( column [, ...] )` in te voeren. Als er dus een waarde komt, wordt er verwacht dat het begint met een `(`, gevolgd door een kolom naam. In ons voorbeeld zou `naam` een correcte kolomnaam zijn voor 2 tabellen. Het aantal kolom namen wat verwacht wordt is variabel; `[, ...]` geeft aan dat het mogelijk is meerdere kolomnamen (meer van hetzelfde) op te geven door deze te scheiden met een `,`. Zo'n lijst van elementen wordt ook wel een `tuple` genoemd.

Dan komt er een verplicht blok, aangegeven met een `{`. Hierin staat dat er 2 mogelijkheden zijn; `DEFAULT VALUES` geeft aan dat de database voor de rest niets verwacht en zal proberen de kolommen te vullen met standaard waardes, als dat mogelijk is. In ons voorbeeld is dat niet mogelijk omdat er niet bij elke kolom een standaard waarde staat. Vervolgens wordt er aangegeven dat er `VALUES` kan worden ingevoerd.

De `VALUES` instructie zal het meest gebruikt worden. Net als bij de kolommen wordt deze gevolgd met een lijst tussen haken`(...)`. Als waardes kunnen hier `expression` elementen worden opgegeven. Voor nu is het voldoende om te weten dat `expression` of een waarde is zoals `'Nick'` of `5`, of een `SELECT` querie. Deze ken je nog niet, maar wordt in volgende sectie uitgelegd.

Phew. Je hebt `grammer` leren lezen. Dit is waardevol omdat de documentatie van databases vol staat met allerlei `grammer`. Nu kun je dus naar de documentatie pagina van elke type instructie gaan en een goed idee hebben wat er van je verwacht wordt.

Laten we nu deze kennis toepassen door data toe te voegen aan de voorbeeldtabellen. Allereerst voegen we enkele docenten toe aan de `docent` tabel.

```
INSERT INTO docent (naam, geslacht, in_dienst)
     VALUES ('Arjen', 'M', '1-apr-2019');
INSERT INTO docent (naam, geslacht, in_dienst)
     VALUES ('Nick', 'M', '1-may-2020');
```

Zoals je ziet staan er 2 `INSERT INTO` instructies die beide een regel toevoegen aan de database. Als je goed kijkt naar de `grammar` dan zal je zijn opgevallen dat na de `tuple` voor `VALUES` er een `[, ...]`, oftewel meer van hetzelfde, staat. Dit betekent dat we bovenstaande SQL instructies kunnen samenvoegen naar het volgende.

```
INSERT INTO docent (naam, geslacht, in_dienst)
     VALUES ('Arjen', 'M', '1-apr-2019'),
            ('Nick', 'M', '1-may-2020');
```

In beide voorbeelden is het je wellicht wel opgevallen dat in de lijst van kolommen geen `id` staat. Deze kolom is een `GENERATED` waarde en kan door de database zelf worden bepaald. Het is dan ook niet nodig om deze zelf aan te geven.

Probeer zelf nog een docent toe te voegen. In september van 2020 is Nova als docent in dienst gekomen.

Het was ook mogelijk om tijdens de `INSERT INTO` instructie in de `VALUES` een `SELECT` uit te voeren. Alhoewel je nog niet weet hoe een `SELECT` werkt gaan we deze wel gebruiken in het voorbeeld. Wat hier belangrijk is om te weten is dat door de `SELECT` de waarde van de `id` kolom wordt opgehaald.

In de tabel `cursus` staat een verwijzing naar de hoofddocent door middel van de kolom `hoofddocent_id`. Van te voren kunnen we niet weten wat de waarde van de `id` van een bepaalde rij zal zijn. In onderstaande voorbeeld voegen we een cursus toe van 5 studiepunte waarbij de hoofdocent `Nick` is. De waarde van `id` voor `Nick` is onbekend, maar kunnen we achterhalen door een `SELECT` instructie op te geven.

```
INSERT INTO cursus (naam,ects,hoofddocent_id)
     VALUES ('Database', 5, (SELECT id FROM docent WHERE naam='Nick'));
```

Hetzelfde komt voor in de tabel `docent_cursus` waar enkel verwijzingen naar andere tabellen in staan. Door voor beide kolommen een `SELECT` uit te voeren om de `id` te achterhalen kan `Arjen` gekoppeld worden als docent voor de cursus `Database`.

Er staat nu data in onze database en je hebt geleerd hoe je de `grammar` voor SQL instructies kan lezen.

Probeer zelf nog enkele cursussen toe te voegen, kijk naar de leerlijnomschrijving voor Backend en voeg de cursussen toe aan de database. Je moet uitkomen op een totaal van 30 ECTS.

2.3 Data opvragen

Het `grammar` voor een `SELECT` instructies is als volgt.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
```

In de vorige sectie heb je al geleerd hoe je de `grammer` kan lezen. Het gedeelte na `SELECT` ziet er erg ingewikkeld uit, dus laten we deze nader begijken. `SELECT` wordt gevolgd door een optioneel blok waarbij `ALL` of `DISTINCT` kan worden aangegeven. Als `DISTINCT` wordt gekozen kan daarachter weer een optioneel blok worden geplaatst met `ON`. Voor nu zijn beide opties niet heel relevant, maar zullen later worden toegelicht.

De optionele onderdelen worden gevolgd met een verplicht onderdeel `* | expression`, wat zoveel zegt als alles of een subselectie van de data. En dat is het voor de verplichte onderdelen.

Probeer maar eens de query `SELECT 1` of `SELECT 'Arjen'`. Je zult zien dat dan de waardes `1` of `Arjen` terug komen. Heel handig natuurlijk, maar wij hebben tabellen waar de iets uit willen halen. Daar komt het optionele `FROM` blok naar voren. Daarmee geef je aan dat je data uit een tabel wil halen. Probeer onderstaande query eens.

```
SELECT * FROM docent;
```

Het resultaat zal in ieder geval deze data bevatten. De `id` kolom kan andere nummers bevatten, afhankelijk van hoe vaak je iets geinsert hebt.

```
 id | naam  | geslacht | in_dienst
----+-------+----------+------------
  1 | Arjen | M        | 2019-04-01
  2 | Nick  | M        | 2020-05-01
```

Gaaf he? Maar wat als dit een tabel is met duizenden rijen, hoe zou je de output kunnen beperken?. Je hebt het al gebruikt in de `INSERT INTO` instructies; de `WHERE` instructie. Hiermee kun je aangeven dat je bepaalde onderdelen van de tabel wilt zien die voldoen aan een conditie. Laten we kijken welke rijen de naam `Arjen` bevatten.

```
SELECT * FROM docent WHERE naam='Arjen';
```

Er komen nu enkel rijen terug met de naam `Arjen`. Leuk, maar wat als we willen weten welke docenten de letter 'c' in de naam hebben? het volgende SQL instructie zal niet het gewenste resultaat hebben.

```
SELECT * FROM docent WHERE naam='c';
```

Het resultaat is nu 0 rijen groot. Wat we nodig hebben zijn wildcards. In de SQL taal is een wildcard een `%` en kunnen we deze gebruiken als we in de `WHERE` een `condition` kunnen opgeven. De `condition` verbonden aan een wildcard is altijd `LIKE`. Dus feitelijk vragen we de database: geef mij alle rijen uit de table decent waar de naam ongeveer c is.

```
SELECT * FROM docent WHERE naam LIKE '%c%';
```

Probeer zelf de rijen uit cursus te halen waar de letter 'a' in zitten. Als je verschillende cursussen hebt toegevoegd zou je Java en Database moeten terugkrijgen, wellicht zelfs meer.

2.4 Data bewerken

We hebben in onze tabellen nu al wat data staan. Als het goed is heb je ook verschillende docenten toegevoegd. In de initiele data zijn de docenten enkel met hun voornaam in de tabel gezet. Dat is natuurlijk prima, totdat er een docent bij komt met een naam die al in de tabel staat.

In zo'n situatie is het nodig om data te bewerken. Het is over het algemeen niet een oplossing om alle data weg te gooien en opnieuw in te voeren. Gelukkig is er de `UPDATE` instructie waarmee data bewerkt kan worden.

De syntax, of `grammar`, voor de `UPDATE` instructie is als volgt.

```
UPDATE [ ONLY ] table [ * ] [ [ AS ] alias ]
    SET { column = { expression | DEFAULT } |
          ( column [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
    [ FROM fromlist ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
```

Ook nu ziet het er weer ingewikkelder uit dan dat het daadwerkelijk is. Bij een `UPDATE` dien je aan te geven welke `table` je wil updaten, geef je via `SET` aan welke kolommen moeten worden veranderd en met een `WHERE` clausule kun je aangeven welke conditie waar moet zijn om de update door te voeren. Laten we eens een voorbeeld uitwerken.

De huidige database bestaat uit de volgende data:

```
 id | naam  | geslacht | in_dienst
----+-------+----------+------------
  1 | Arjen | M        | 2019-04-01
  2 | Nick  | M        | 2020-05-01
```

Om de rij met `id` 1 te updaten van Arjen naar Arjen Wiersma schrijven we de volgende SQL query:

```
UPDATE docent SET naam='Arjen Wiersma' WHERE id=1;
```

Het resultaat zal zijn dat rij 1 wordt bewerkt en de naam wordt geupdate. Maar stel dat in plaats van Arjen Wiersma het eigenlijk Rebecca Wiersma had moeten zijn en bij het invoeren dus de compleet verkeerde naam en geslacht waren ingevoerd?

Probeer zelf de regel van Arjen te veranderen naar Rebecca Wiersma met het juiste geslacht en dat allemaal in 1 SQL query.

```
UPDATE docent SET naam='Rebecca Wiersma', geslacht='V' WHERE id=1;
```

Het resultaat van deze bewerking is:

```
 id |      naam       | geslacht | in_dienst
----+-----------------+----------+------------
  2 | Nick            | M        | 2020-05-01
  1 | Rebecca Wiersma | V        | 2019-04-01
```

Laten we dit voorbeeld weer snel veranderen, anders komt de rest van de tekst hierna niet meer goed.

```
UPDATE docent SET naam='Arjen Wiersma', geslacht='M' WHERE id=1;
```

2.5 Data verwijderen

Net als data toevoegen, kunnen we natuurlijk ook data verwijderen uit een database.

De syntax, of `grammar`, voor de `DELETE` instructie is als volgt .

```
DELETE FROM [ ONLY ] table [ * ] [ [ AS ] alias ]
    [ USING usinglist ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
```

Net als de `UPDATE` instructie is de `DELETE` instructie redelijk eenvoudig. Je bent nu al enigszins gewend om `grammar` te lezen.

Laten we proberen docenten te verwijderen. De instructies voor `DELETE` geven aan dat enkel de naam van de tabel een verplichte waarde is. Laten we de meest simpele versie eerst proberen.

```
DELETE FROM docent;
```

Dit geeft een gigantisch lange error.

```
ERROR:  update or delete on table "docent" violates foreign key constraint "fk_hoofddocent" on table "cursus"
DETAIL:  Key (id)=(2) is still referenced from table "cursus".
```

Feitelijk staat hier dat een delete van de tabel `docent` de relatie met de tabel `cursus` kapot zou maken. Oftewel, er is een docent gekoppeld als hoofddocent bij een cursus. De opdracht slaagt dus niet en er wordt niets verwijderd.

De database zal er alles aan doen om te zorgen dat de data in het systeem zo juist mogelijk blijft.

Probeer zelf eens alle inhoud uit de tabel `cursus` te verwijderen. Welke relatie zorgt er nu voor dat deze opdracht niet lukt?

Om wel data te verwijderen is het dus nodig dat de rij geen relatie of afhankelijkheid heeft. Stel dat in de database een cursus stond met de naam Python. In het nieuwe curriculum is besloten dat er geen Python meer wordt gegeven en dus kan de cursus weg.

```
 id |   naam   | ects | hoofddocent_id
----+----------+------+----------------
  1 | Database |    5 |              2
  2 | Python   |    5 |              1
```

Allereerst moet dan de koppeling tussen docenten en de cursus verwijderd worden. In de koppel tabel `docent_cursus` kunnen we alle koppelingen met de cursus verwijderen. Omdat er geen afhankelijkheid met deze tabel is (deze tabel is immers afhankelijk van andere tabellen, en niet andersom) worden alle koppelingen tussen docenten en de cursus verwijderd.

```
DELETE FROM docent_cursus WHERE cursus_id=2;
```

Vervolgens kan de cursus zelf verwijderd worden.

```
DELETE FROM cursus WHERE id=2;
```

Je hebt gezien hoe data uit de database verwijderd kan worden. Door de foutmeldingen goed te analyseren kan worden achterhaald welke relaties verwijderd moeten. Als er geen afhankelijkheden meer zijn van de rij die verwijderd wordt kan de data uit de database verwijderd worden.

3. Select met Joins

3.1 Introductie

Bij het opvragen van data uit de `cursus` tabel is het je wellicht opgevallen dat je onderstaande resultaat krijgt.

```
 id |   naam   | ects | hoofddocent_id
----+----------+------+----------------
  1 | Database |    5 |              2
```

Wie is eigenlijk hoofddocent 2? Het zou veel handiger zijn als er naast de naam van de cursus de naam van de hoofddocent zou staan zodat je weet waar je naar toe kan met al je vragen. Dit wordt gedaan door middel van `Join` statements. De syntax voor een `JOIN` is als volgt.

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
```

In deze syntax zijn T1 en T2 tabellen. Na een paar keer lezen betekent het dus dat er verschillende `JOIN` instructies mogelijk zijn. van een simpele `T1 JOIN T2 ON ...` tot aan `T1 LEFT OUTER JOIN T2 ON ...`. Laten we een aantal variaties doorlopen om te kijken wat het gewenste resultaat geeft.

De JOIN geeft aan dat de waardes uit de 2 tabellen met elkaar verbonden moeten worden. Deze verbinding gebeurt op basis van de `boolean_expression` achter de `ON`. Als voor een regel de `boolean_expression` op TRUE uit komt dan zal de combinatie worden gekoppeld in de output. Laten we het meest simpele eerst proberen.

```
SELECT * FROM cursus JOIN docent ON true;
```

Hierbij vragen we de database om elke rij uit cursus op te halen, deze te koppelen met de docent tabel waarbij de conditue `true` altijd waar zal zijn. Dus voor elke rij uit cursus zal elke rij van docent worden gekoppeld. Bekijk de output van deze query:

```
 id |   naam   | ects | hoofddocent_id | id | naam  | geslacht | in_dienst
----+----------+------+----------------+----+-------+----------+------------
  1 | Database |    5 |              2 |  1 | Arjen | M        | 2019-04-01
  1 | Database |    5 |              2 |  2 | Nick  | M        | 2020-05-01
```

Dit is overduidelijk fout. Arjen is geen hoofddocent voor Database. Dit komt omdat de conditie die de database controleert altijd de waarde `true` gebruikt en dus altijd waar is. Wat we eigenlijk willen vragen is: *geef mij alle rijen uit de tabel cursus en voeg daarbij de rij uit docent waarbij hoofddocent_id van cursus gelijk is aan de id van docent*. Dat is een hele mond vol, maar wel een hele specifieke instructie.

```
SELECT * FROM cursus JOIN docent ON cursus.hoofddocent_id=docent.id;
```

De SQL query is al bijna net zo lang als de uitgeschreven vraag, maar nu wordt er dus voor elke rij in de tabel `cursus` gekeken of de `hoofddocent_id` uit cursus overeenkomt met de `id` uit `docent`. Alleen bij de rij `2` in de docent tabel is dit waar.

```
 id |   naam   | ects | hoofddocent_id | id | naam | geslacht | in_dienst
----+----------+------+----------------+----+------+----------+------------
  1 | Database |    5 |              2 |  2 | Nick | M        | 2020-05-01
```

Maar wat als we de vraag willen omdraaien? In plaats van te weten welke docent bij een cursus hoort, kunnen we ook vragen welke cursus bij welke docent hoort. Laten we de tabellen eens omdraaien.

```
SELECT * FROM docent JOIN cursus ON docent.id=cursus.hoofddocent_id;
```

Het zal je niet verbazen, maar het geeft preciest hetzelfde resultaat. Alleen dan
met de docent eerst en dan de cursus.

```
 id | naam | geslacht | in_dienst  | id |   naam   | ects | hoofddocent_id
----+------+----------+------------+----+----------+------+----------------
```

```
  2 | Nick | M        | 2020-05-01 |  1 | Database |    5 |             2
```

Standaard zal de `JOIN` alleen resultaten laten zien als de conditie waar is. Dus als er een koppeling is tussen de docent en de cursus. Maar wat als de vraag is om ook docenten te laten zien die nog niet zijn aangewezen als hoofddocent?

De `JOIN` heeft ook de mogelijkheid om met deze situatie om te gaan. Dit is de `OUTER JOIN`. Dat wil zeggen dat als de conditie waar is, dan worden de rijen aan elkaar gekoppeld, en anders wordt voor de andere tabel een lege rij toegevoegd. Maar hoe weet de database nou welke kan een lege rij moet krijgen? Daarvoor wordt aan jou gevraagd om aan te geven of het de `LEFT` (linker) of `RIGHT` (rechter) kant van de vergelijking is die niet leeg mag zijn.

Dat is wel even een hersenkraker, maar laten we een voorbeeld nemen. Onderstaande query is dezelfde als bij het vorige voorbeeld, maar dan met `LEFT OUTER` toegevoegd.

```
SELECT * FROM docent LEFT OUTER JOIN cursus ON docent.id=cursus.hoofddocent_id;
```

Het resultaat is:

```
 id | naam  | geslacht | in_dienst  | id |   naam   | ects | hoofddocent_id
----+-------+----------+------------+----+----------+------+----------------
  2 | Nick  | M        | 2020-05-01 |  1 | Database |    5 |              2
  1 | Arjen | M        | 2019-04-01 |    |          |      |
```

Het is nu dus duidelijk dat docent Arjen nog geen vak heeft waar hij hoofddocent is.

Probeer zelf de vraag om te draaien, welke docent hoort bij welke cursus? Het je hier dan een `RIGHT JOIN` of een `LEFT JOIN` nodig om de volgende output te krijgen?

```
 id |   naam   | ects | hoofddocent_id | id | naam  | geslacht | in_dienst
----+----------+------+----------------+----+-------+----------+------------
  1 | Database |    5 |              2 |  2 | Nick  | M        | 2020-05-01
    |          |      |                |  1 | Arjen | M        | 2019-04-01
```

Probeer zelf met de cursussen en docent die je hebt toegevoegd te experimenteren.

3.2 Koppeltabellen

Je hebt tot nu toe geleerd hoe je data toevoegt en hoe je data opvraagt. Je hebt ook geleerd hoe je dit met joins kan doen.

In ons model hebben we gezegd dat we cursussen en docenten hebben en dat elke cursus een hoofddocent heeft. De enige echte vraag die nog rest is welke docent geeft welke cursus, want de hoofddocent kan het natuurlijk niet allemaal alleen.

Als we bij de docent een veld zouden maken die verwijst naar de cursus dan kan een docent maar 1 cursus geven. Als we bij de cursus een veld maken die verwijst naar de docent dan zou nooit een andere docent de cursus kunnen geven. Dit lijkt een onmogelijke situatie.

Probeer zelf voordat je verder leest even na te denken over deze situatie. Hou zou jij docenten aan cursussen koppelen.

Op het moment dat de vraag wordt `hoe kan ik docenten aan cursussen koppellen?` dan merk je al dat je over meervoud praat. Aan beide kanten van de relatie heb je meerdere entiteiten. We praten hier dan ook over een `N-to-N` of `many-to-many` relatie.

Deze vorm van relaties zie je veel in systemen en er is dan ook een goede oplossing voor. We maken simpelweg nog een tabel, die docenten en cursussen aan elkaar koppelt. Onderstaande SQL maakt deze tabel. Ook hier wordt elk `_id` gebruikt in een foreign key constraint, oftewel de waarde in deze kolom moet bestaan in een andere tabel.

Naast de `id` van docent en cursus bevat deze tabel geen data van zichzelf. Het dient enkel om meerdere rijen uit de andere tabellen aan elkaar te koppelen.

```
DROP TABLE IF EXISTS docent_cursus;

CREATE TABLE docent_cursus (
    docent_id INT,
    cursus_id INT,
    CONSTRAINT fk_docent FOREIGN KEY (docent_id) REFERENCES docent(id),
    CONSTRAINT fk_cursus FOREIGN KEY (cursus_id) REFERENCES cursus(id)
);
```

Op basis van de kennis uit het *Data Toevoegen* onderdeel kunnen we in deze tabel toevoegen dat Arjen het vak Database geeft.

```
INSERT INTO docent_cursus (docent_id, cursus_id)
     VALUES ((SELECT id FROM docent WHERE naam='Arjen'),
             (SELECT id FROM cursus WHERE naam='Database')),
            ((SELECT id FROM docent WHERE naam='Nick'),
             (SELECT id FROM cursus WHERE naam='Database'));
```

Met deze `INSERT INTO` query zeggen we dat beide docenten Arjen en Nick in staat zijn om de cursus Database te geven.
Als je kijkt naar de rijen in de tabel zal je nu het volgende zien.

```
 docent_id | cursus_id
-----------+-----------
         1 |         1
         2 |         1
```

Probeer zelf verschillende docenten toe te voegen aan de cursussen. Wellicht wil je ook eens les krijgen van Nova of Nick of wellicht heb je andere docenten toegevoegd. Wat gebeurt er als je een docent invoert die je niet hebt toegevoegd aan de docent tabel?

Laten we kijken of we met al onze kennis nu een lijst van vakken kunnen maken waarbij duidelijk wordt welke docenten deze vakken kunnen geven.

Probeer zelf met de kennis van SELECT en JOIN een lijst te maken van vak en docent.

Een eerste poging is om de tabel docent en docent_cursus met elkaar te verbinden.

```
SELECT * FROM docent JOIN docent_cursus ON docent_cursus.docent_id=docent.id;
```

Het resultaat mag er zijn, maar geeft alleen `cursus_id` en niet de naam van de cursus.

```
 id | naam  | geslacht | in_dienst  | docent_id | cursus_id
----+-------+----------+------------+-----------+-----------
  1 | Arjen | M        | 2019-04-01 |         1 |         1
  2 | Nick  | M        | 2020-05-01 |         2 |         1
```

Om de naam van de cursus er bij te krijgen is de `cursus` tabel nodig. Laten we er simpelweg nog een join bij zetten en kijken wat het resultaat is.

```
SELECT * FROM docent JOIN docent_cursus ON docent_cursus.docent_id=docent.id JOIN cursus ON cursus.id = docent_cursus.cursus_id;
```

Dat wordt wel een hele lange string. Wellicht kunnen we het iets anders formatteren om leesbaarder te worden.

```
SELECT *
  FROM docent
  JOIN docent_cursus ON docent_cursus.docent_id=docent.id
  JOIN cursus ON cursus.id = docent_cursus.cursus_id;
```

Dat wordt al beter. Als we deze query analyseren vragen we dus om alle rijen te laten zien van docent en bij elke rij de rijen uit docent_cursus weer te geven waarbij de `docent_id` overeenkomt met de `id` uit docent en dan ook nog om alle rijen weer te geven vanuit cursus waar de `id` van cursus overeenkomt met de `cursus_id` uit docent_cursus. Wow. Wat een vraag. Laten we eerst kijken naar het resultaat.

```
 id | naam  | geslacht | in_dienst  | docent_id | cursus_id | id |   naam   | ects | hoofddocent_id
----+-------+----------+------------+-----------+-----------+----+----------+------+----------------
  1 | Arjen | M        | 2019-04-01 |         1 |         1 |  1 | Database |    5 |              2
  2 | Nick  | M        | 2020-05-01 |         2 |         1 |  1 | Database |    5 |              2
```

Ja! Het is wel een antwoord op de vraag. Nu valt het je wellicht ook op dat we de query niet helemaal juist hebben gelezen. Tot nu toe is er gezegd dat `SELECT *` alle rijen weergeeft, maar dat is eigenlijk niet waar. Het betekent *laat alle kolommen van de rij zien*.

Eigenlijk willen we al deze kolommen niet zien. De vraag is om aan te geven welke docenten welke vakken kunnen geven. Dan zou het wel fijn zijn om bijvoorbeeld alleen de kolommen `naam` uit docent en `naam` uit cursus en het aantal `ects` te laten zien.

In plaats van *alle kolommen* willen we dus maar een beperkt aantal kolommen laten zien. In plaats van * kunnen we simpelweg een lijst van kolomnamen invoeren. Laten we het proberen.

```
SELECT naam, naam, ects
  FROM docent
  JOIN docent_cursus ON docent_cursus.docent_id=docent.id
  JOIN cursus ON cursus.id = docent_cursus.cursus_id;
```

Deze aanpassing levert niet het juiste resultaat op. Er komt een foutmelding: `ERROR: column reference "naam" is ambiguous`. Ambiguous wil zeggen dat er meerdere kolommen zijn met dezelfde naam. Dat klopt ook want beide cursus en docent hebben deze kolom. We moeten dus aangeven uit welke tabel welke kolom komt. De kolom `ects` komt slechts uit 1 tabel en kan zonder tabelnaam gebruikt worden.

```
SELECT docent.naam, cursus.naam, ects
  FROM docent
  JOIN docent_cursus ON docent_cursus.docent_id=docent.id
  JOIN cursus ON cursus.id = docent_cursus.cursus_id;
```

Het is echter verstandig om rechtlijnig te zijn en dezelfde methode aan te houden en dus voor ook `ects` de tabelnaam op te geven.

```
SELECT docent.naam, cursus.naam, cursus.ects
  FROM docent
  JOIN docent_cursus ON docent_cursus.docent_id=docent.id
  JOIN cursus ON cursus.id = docent_cursus.cursus_id;
```

Het resultaat is als volgt.

```
 naam  |   naam   | ects
-------+----------+------
 Arjen | Database |    5
 Nick  | Database |    5
```

Perfect! Nou ja, bijna. De kolomnamen zijn nog steeds hetzelfde. Dit kan voor verwarring zorgen. Gelukkig kunnen we kolommen hernoemen met de instructie `AS naam` achter de kolom naam. Laten we dat eens proberen.

```
SELECT docent.naam AS docent, cursus.naam AS cursus, cursus.ects
  FROM docent
  JOIN docent_cursus ON docent_cursus.docent_id=docent.id
  JOIN cursus ON cursus.id = docent_cursus.cursus_id;
```

Het resultaat is als volgt:

```
 docent |  cursus  | ects
--------+----------+------
 Arjen  | Database |    5
 Nick   | Database |    5
```

Perfect! Je hebt geleerd hoe je met joins meerdere tabellen op verschillende wijze aan elkaar kan koppelen. Hiermee is het mogelijk om van een genormaliseerd model een eenduidig overzicht te maken van relevante data.

4. RRM & Create

4.1 Relationele Modellen

In de vorige hoofdstukken hebben we gewerkt met een aantal tabellen, maar hebben we de relatie tussen deze tabellen nog niet onderzocht en hoe deze gemodelleerd dienen te worden.

```
Docent (id, naam, geslacht, in_dienst)
```

De naam van de tabel is ook de naam van de relatie en de namen van alle kolommen van de tabel Docent staan tussen de haakjes beschreven. In het relationele model noem je de kolommen attributen. Een rij uit de tabel heet een tupel in een relatie.

| relatie | tabel | | attribuut | kolom | | tupel | rij |

Van ieder attribuut kan het domein vastgelegd worden. Dit betekent dat van ieder attribuut wordt vastgelegd wat de mogelijke waarden van dat attribuut mogen zijn.
Bijvoorbeeld in de relatie Docent:

- `naam` is een text veld
- `geslacht` is een 1 character code
- `in_dienst` is een datum

Het domein geeft dus aan uit welke verzameling waarden een keuze gemaakt mag worden voor dat attribuut. Je kunt een datatype, bijvoorbeeld integer, ook als een domein zien.

In het relationele model kan een attribuut per tupel precies één waarde hebben. Iedere tupel in de relatie Docent heeft dus één invulling voor naam, geslacht en in_dienst.

Een domein kan voor meer dan een attribuut gebruikt worden.
In het relationele model worden vaak de begrippen intentie en extensie gebruikt.

- Intentie wil zeggen: de beschrijving, de definitie van een relatie, de regels die voor de relatie gelden.
- Extensie betekent: de verzameling tupels van een relatie met de mogelijke waarden in de tupels.

De intensie van de relatie Docent is dus:

```
Docent (id, naam, geslacht, in_dienst)
```

De extensie van de relatie Docent is dus:
| 1 | Arjen | M | 01-04-2019 |

Een relatie wordt gedefinieerd als een verzameling tupels. Volgens de definitie van een verzameling zijn alle elementen van een verzameling verschillend, dus moeten alle tupels van een verzameling ook verschillend zijn. Dit betekent dat het niet kan voorkomen dat er twee tupels zijn met precies dezelfde waarden voor alle attributen. De volgende extensie is dus onmogelijk:

| 1 | Arjen | M | 01-04-2019 | | 1 | Arjen | M | 01-04-2019 |

Of anders gezegd, de rijen (tupels) van een tabel zijn uniek.

Dit betekent dus dat er altijd een attribuut is dat een tupel kan identificeren. Vaak is het zo dat er meer dan een attribuut is dat een tupel kan identificeren. Deze identificerende attributen worden kandidaatsleutels genoemd. Uit deze kandidaatsleutels wordt een primaire sleutel gekozen. In de relatie Docent zijn (gezien de extensie) id, naam, `in_dienst` en geslacht kandidaatsleutel. Met een waarde voor een van deze attributen kan precies één tupel geïdentificeerd worden.

Als primaire sleutel is id gekozen, ook omdat praktisch gezien een database een numeriek `id` zelf kan aanmaken en beheren. In het relationele model geef je aan dat een attribuut een primaire sleutel is door dit attribuut te onderstrepen.

```
Docent (_id_, naam, geslacht, in_dienst)
```

Een primaire sleutel en dus ook de kandidaatsleutel kan uit meer dan één attribuut bestaan. Dit wordt echter alleen toegepast als dat echt nodig is. In docent kun je kiezen voor een combinatie van id en naam als primaire sleutel. Met beide attributen kun je een tupel identificeren. Naam is echter niet nodig, met code kun je volstaan. De primaire sleutel moet zo klein mogelijk zijn, dit wordt wel de minimaliteitseis genoemd.

De laatste sleutel in het relationele model is de vreemde sleutel of de foreign key. Deze sleutel wordt gebruikt om gegevens uit verschillende relaties te combineren. Neem bijvoorbeeld de relaties Docent en Cursus:

```
Docent (id, naam, geslacht, in_dienst, salaris)
Cursus (id, naam, ects, /hoofdocent/)
```

`hoofddocent` in `Cursus` is een vreemde sleutel (vreemde sleutels worden dus cursief weergegeven of als je schrijft: dan onderstrepen met een onderbroken streep). Met behulp van `hoofddocent` in `Cursus` kan in Docent de naam van de hoofddocent opgezocht worden die bij deze code hoort. In het relationele model staat ook altijd aangegeven waar de vreemde sleutel naar verwijst en of null wel of niet is toegestaan.

Er gelden twee belangrijke regels voor een relationeel model naar aanleiding van de primaire sleutel en de vreemde sleutel:

- de entiteitsintegriteitsregel
- referentiële integriteitsregel

Entiteitsintegriteitsregel wil zeggen dat een primaire sleutel nooit NULL mag zijn en ook niet dubbel mag vóórkomen. Dit is gezien het bovenstaande logisch aangezien je met een primaire sleutel een tupel moet kunnen identificeren.
Referentiële integriteitsregel wil zeggen dat een vreemde sleutel altijd verwijst naar een primaire sleutel en altijd een waarde heeft, die ook voorkomt als primaire sleutel in de relatie waar de vreemde sleutel naar verwijst of (als dat mag) NULL is.

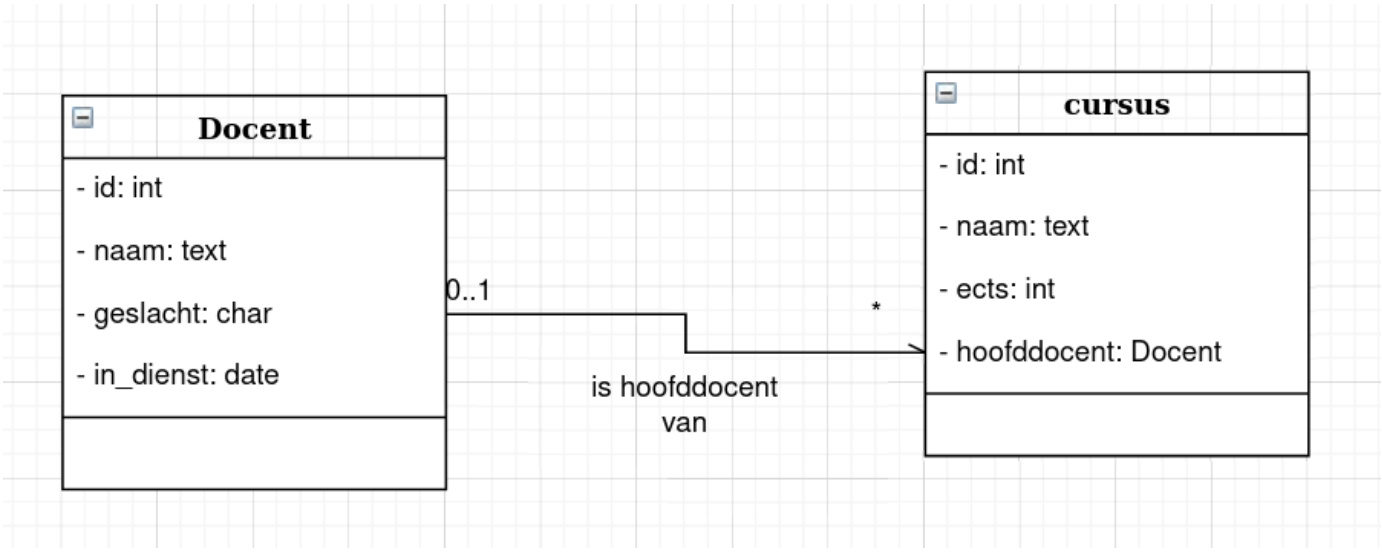
Opdracht

Werk een aantal nieuwe entiteiten en relaties uit. Er zijn nu docenten en curussen, maar nu zijn er nog `Student` en `Resultaat` entiteiten nodig.

4.2 Naar een relationeel model

Een klassendiagram wordt gebruikt om op een schematische manier weer te geven welke klassen voor een (te ontwerpen) systeem nodig zijn. In dit diagram staan niet alleen de klassen en de associaties tussen de klassen, maar ook de eigenschappen van die klassen. Deze eigenschappen zijn attributen en operaties.

Een klassendiagram wordt gebruikt bij een object georiënteerde ontwerpmethode en het is natuurlijk voor de hand liggend om als voor een database gekozen wordt voor de opslag van de objecten, dat dan gekozen wordt voor een object georiënteerd DBMS. In deze cursus wordt echter gewerkt met een relationeel DBMS. In een relationele database komen geen objecten voor. Dit betekent dat er slechts een deel van het klassendiagram vertaald kan worden naar de relationele database, namelijk alleen de klassen met de attributen en de associaties tussen de klassen. Bekijk onderstaand klassendiagram:



In dit klassendiagram staan twee klassen afgebeeld: Docent en Cursus.
Het vertalen van het klassendiagram naar een relationeel model gaat als volgt:

De eerste stap

Iedere klasse in het klassenmodel wordt een relatie in het relationele model.
Er zijn dus twee relaties:

```
Docent (id, naam, geslacht, in_dienst)
Cursus (id, naam, ects, hoofddocent)
```

De tweede stap

Elke relatie in het relationele model heeft een primaire sleutel.
Bepaal per relatie de kandidaatssleutels en kies hieruit een primaire sleutel.

```
Docent (_id_, naam, geslacht, in_dienst)
Cursus (_id_, naam, ects)
```

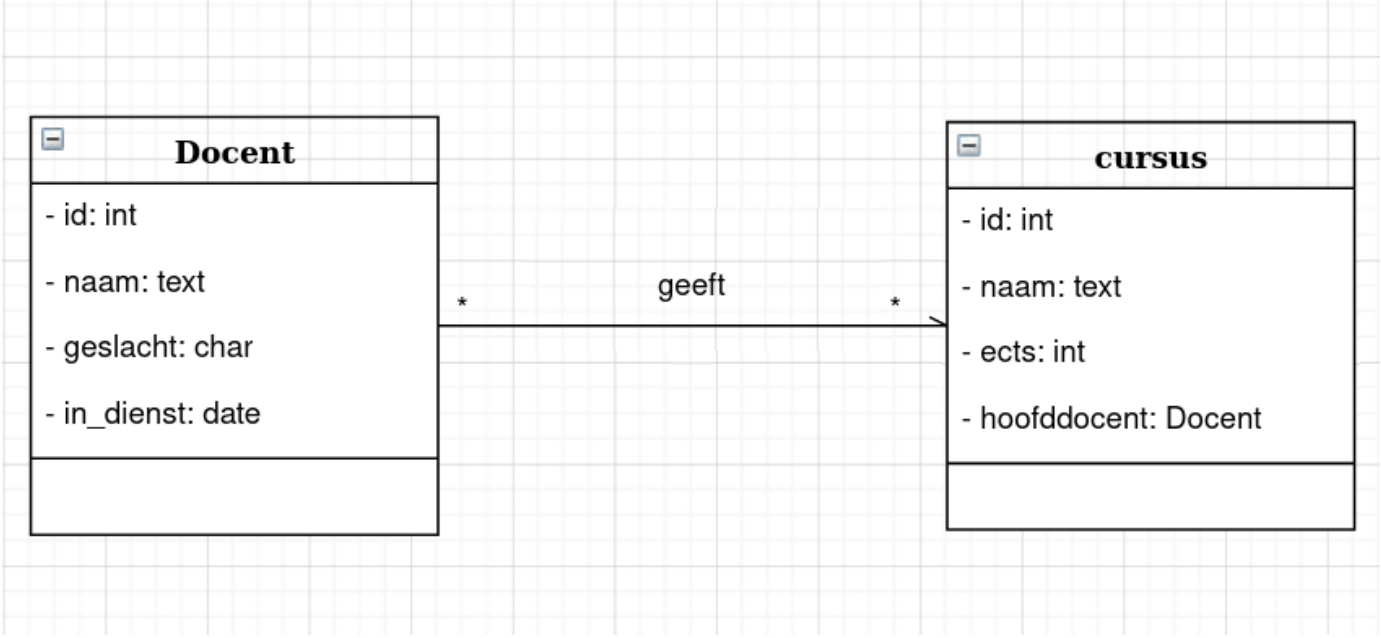
De derde stap

Vervolgens moeten de verbanden tussen de relaties gelegd worden. De associaties uit het klassenmodel moeten ook herkenbaar zijn in het relationele model.

De vertaling naar het relationele model is afhankelijk van het soort associatie.

Bij iedere 1-op-veel associatie (dus een associatie die aan de ene kant 1 of 0..1 heeft en aan de andere kant een * of 1..* als multipliciteit heeft (zoals hierboven aangegeven), wordt de primaire sleutel van de 1-kant extra opgenomen aan de veel kant. Dit extra attribuut is de vreemde sleutel. In dit voorbeeld: er bestaat een 1-op-veel associatie tussen Docent en Cursus. Docent is de 1-kant, Cursus de veel kant. `hoofddocent_id` wordt dus extra opgenomen bij Cursus en is een vreemde sleutel. Daarnaast kun je aangeven of voor deze vreemde sleutel NULL is toegestaan. Uit het model blijkt dat een cursus één of geen hoofddocent heeft. NULL is dus toegestaan.

```
Cursus (_id_, naam, ects, /hoofddocent_id/)
```



Bij iedere veel-op-veel associatie (dus een associatie die aan beide kanten bijvoorbeeld een * als multipliciteit heeft, wordt een nieuwe relatie gemaakt. Deze nieuwe relatie heeft als attributen de primaire sleutels van de klassen van de veel-op-veel associatie. Deze primaire sleutels zijn samen de primaire sleutel in de nieuwe relatie en zijn ook vreemde sleutel. Hierbij zijn uiteraard NULL waarden niet toegestaan: NULL voor een primaire sleutel kan niet. In dit voorbeeld: er bestaat een veel-op-veel associatie tussen Docent en Cursus: dit levert een nieuwe relatie op.

Deze noemen we Docent_Cursus en de relatie ziet er als volgt uit:

```
Docent_Cursus (/docent_id/, /cursus_id/)
```

`docent_id` is een vreemde sleutel en verwijst naar het `id` van Docent. Hetzelfde geldt voor `cursus_id` voor de tabel `Cursus`.

Bij iedere 1-op-1 associatie (dus een associatie die aan beide kanten bijvoorbeeld een 1 of 0..1 als multipliciteit heeft, wordt de primaire sleutel van één van de twee opgenomen als vreemde sleutel bij de ander. Heb je te maken met een 0- of 1-associatie dan neem je de primaire sleutel op aan de 0-kant. Heb je te maken met precies 1-op-1, dan kun je zelf een keuze maken.

Oefening

Stel een relationeel model op waar je het bestaande model uitbreidt:

- Docent
- Cursus
- Student
- Eindopdracht waarin het cijfer van een student voor een cursus wordt bepaald

Als extra:

- Voeg een Examinator toe aan de eindopdracht en implementeer het 4 ogen principe (beide docent en examinator geven een cijfer) toe.

De uitwerkingen kun je hier bekijken.

5. Non relationele databases

Naast de relationele databases wordt in de praktijk ook gebruik gemaakt van NoSQL-databases

Wat is hier het verschil tussen de twee varianten SQL en NoSQL? Waarom was het nodig een alternatief voor de relationele database te hebben? Voor welke doelen is een relationele database geschikt en voor welke doelen juist niet? Op al deze vragen gaan we in dit hoofdstuk in.

In de voorgaande hoofdstukken hebben we de structuur van relationele databases gezien. We hebben gelezen in welke branches deze databases gebruikt kunnen worden. Het relationele aspect van de SQL-databases moet inmiddels helder zijn voor aan dit hoofdstuk wordt begonnen. Indien dit nog niet het geval is, is het uiterst raadzaam om de voorgaande opdrachten te maken.

Het achterliggende idee van NoSQL-databases bestaat al zeer lang, zo lang zelfs dat we terug naar de tijd kunnen waar de eerste computer uitgevonden werd. Toen werd het niet aangeduid als een NoSQL-database, maar het idee is hetzelfde. Bovendien was de techniek niet zo uitgebreid als nu. De NoSQL data opslagtechniek (het idee) werd destijds gebruikt voor nogal specifieke gebieden zoals hiërarchische directory of het opslaan van gebruikersnamen en wachtwoorden. Na verloop van tijd werden SQL- databases uitgevonden als opslagtechniek. Deze techniek had sterke functies zoal query technieken, transactie beheer, enzovoorts. Hierdoor werden SQL-databases bijna overal voor gebruikt, want er was voor bijna elke taak wel een functie. De populariteit van SQL-databases had als gevolg dat er veel minder interesse en dus ook minder doorontwikkeling was in een opslag-techniek die veel minder kon (Tiwari, 2011).

Na verloop van tijd, door een toename van SQL-databases die in gebruik genomen zijn door bedrijven, werd al snel duidelijk dat het brede functie-aanbod in combinatie met een grote hoeveelheid data (meestal een paar Terabyte [Prof NoSQL]) van SQL-databases ook het grote nadeel is. Dit komt doordat de distributie van SQL-databases complex blijkt te zijn. Bovendien is het lastig, in een gedistribueerde SQL-database omgeving, om gebruik te maken van het brede functie aanbod van SQL-databases. Deze problemen worden zichtbaar in de vorm van inefficiënt afhandelen van opdrachten en parallel uitvoeren van opdrachten, schaalbaarheid en niet te vergeten de kosten (Orielly, 2010).

Vooral bedrijven als Google, Amazon en Yahoo merkten de eerder omschreven gevolgen al snel op. Google had als doel om een schaalbare infrastructuur te implementeren, die parallel opdrachten met grote hoeveelheid data kon uitvoeren. De ontwikkelingen voor zo'n DBMS begonnen in 2004. In 2006 bracht Google een wetenschappelijk artikel hierover uit. Hierin werd onder andere duidelijk dat het DBMS van Google niet relationeel, maar wel schaalbaar is (Liu, sd). Dit artikel, samen met een aantal andere artikelen, wekte zeer grote interesse op in de open source community. Al snel kwam Lucene uit, een alternatief op Google's BigTable, die door de open source community ontwikkeld is. Niet lang daarna stapte een van de Lucene ontwikkelaars over naar Yahoo om daar ook, samen met andere ontwikkelaars, een soortgelijke variant te ontwikkelen (Tiwari, 2011).

Binnen een jaar, samen met Google's wetenschappelijke artikel over BigTable, werden niet relationele en schaalbare database management systemen zeer populair. In 2007 bracht ook Amazon een soortgelijke DBMS uit, genaamd Dynamo, en ook Amazon publiceerde een wetenschappelijk artikel hierover (Tiwari, 2011; Kai, april 2010; Strauch, 2010).

Bedrijven als Facebook en andere open source community' s besloten ook om niet relationele en schaalbare databasemanagement systemen te ontwikkelen, die goed kunnen omgaan met het parallel afhandelen van opdrachten. Pakketten als Cassandra en CouchDB zijn hiervan het resultaat.

In 2009 werd er een conferentie gehouden over niet relationele databases waar de term NoSQL weer volop in gebruik genomen werd als aanduiding voor niet-relationele databases, die fundamenteel anders zijn dan relationele databases, zoals Google's BigTable en Amazon's Dynamo. Inmiddels wordt de term in de IT-wereld gebruikt als "No SQL" of "Not Only SQL". Over beide termen valt te discussiëren, maar zoals voormalig Rackspace werknemer Eric Evans het heeft gezegd: "The whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for."

NoSQL-databases worden in verband gebracht met grote datasets die flexibel toegankelijk moeten zijn, waarbij het horizontaal schalen van de database niet een zeer complexe taak moet worden. (Tiwari, 2011). Een NoSQL-database is eigenlijk geen product of een techniek op zichzelf. Het wordt tegenwoordig meer gebruikt als een richtlijn hoe een DBMS er uit moet zien, waarbij onder andere de volgende punten belangrijk zijn:

- Het DBMS moet geen performance last ondervinden bij een grote dataset (meestal een paar Terabyte) met veel (parallelle) opdrachtaanvragen.
- Het DBMS moet makkelijk (en goedkoop) geschaald kunnen worden.
- Grote datasets moeten efficiënt opgeslagen worden.

Dit zijn dan ook een aantal gebieden waar NoSQL sterk in is. De term NoSQL-database representeert inmiddels een set NoSQL DBMS-pakketten, die aan de hierboven genoemde eisen voldoen. Uiteraard worden algemene concepten (over data opslag en bewerking) natuurlijk altijd wel gedeeld binnen variërende NoSQL-databases, zoals het feit dat ze niet relationeel zijn (Tiwari, 2011).

We zien dat steeds meer bedrijven, met webapplicaties en in het gebied van sociale media toepassingen, een groei hebben in hun databanken en dat er een grotere hoeveelheid data opgeslagen moet worden. Naast deze zaken groeien ook nog eens het aantal parallelle lees- en schrijfoperaties, die de database moet uitvoeren. Deze bedrijven beginnen door de groei van hun databanken problemen te ondervinden op de eerder beschreven gebieden, waar NoSQL sterk in is. Hierdoor neemt de populariteit van NoSQL- databases toe in verschillende sectoren. Bedrijven zoals Facebook, Twitter en Google, waar de nadruk ligt op de gegevens die door de gebruiker gegenereerd en opgeslagen worden, hebben NoSQL-databases al een geruime tijd in gebruik. Vaak heeft het bedrijf zelf, intern, een eigen NoSQL-databasemanagementsysteem ontwikkeld. (Tiwari, 2011; Cattel, 2010; Pady, 2011).

Het is lastig om exacte kenmerken van een NoSQL-databasemanagementsysteem aan te duiden, omdat elk NoSQL- databasemanagementsysteem met zijn eigen ontwikkelde kenmerken komt en gebaseerd is op een andere architectuur.

Over het algemeen komt het erop neer, dat het brede aanbod van NoSQL-databasemanagementsystemen de volgende kenmerken heeft.

- Er is geen databaseschema.
- NoSQL werkt met verschillende datamodellen (document, key value, column store).
- Er zijn geen JOIN operaties.
- NoSQL is niet relationeel.
- Vaak worden niet alle ACID (Atomair, Consistent, Isolated, Durable)-regels en kenmerken gegarandeerd.
- NoSQL maakt gebruik van het CAP (Consistent, Availability, Partition tolerance)-theorema.
- Horizontaal schalen is flexibel.
- NoSQL is geoptimaliseerd voor grote dataopslag en hoge performance.
- Databaseopslag kan verdeeld worden.
- Database kan flexibel worden geclusterd.
- De SQL-taal is vaak in een ander vorm geïmplementeerd.

(Cattel, 2010; Yunhua, 2011, Strauch, 2010)

Bij het verticaal schalen van de database wordt de databaseserver uitgebreid middels verschillende hardware elementen, bijvoorbeeld extra geheugen, snellere harde schrijven, enzovoorts. Bij het horizontaal schalen van de database worden meer databaseservers ingezet en worden de verschillende databaseservers (computers) aan elkaar gekoppeld, die als het ware als één database (server) functioneren. NoSQL legt de nadruk dan ook op het horizontaal schalen (Cattel, 2010).

De schema's zijn vervangen door bijvoorbeeld collecties of kolommen. Elk database binnen een NoSQL database management systeem heeft een datamodel. Het datamodel geeft grotendeels weer hoe de data structuur is, zo slaat het ene bedrijf data in documenten binnen een verzameling op en een ander bedrijf in kolommen.

NoSQL-databases zijn niet relationeel, daardoor kunnen geen JOIN operaties uitgevoerd worden. Het niet relationeel zijn van de database is een van de redenen waarom NoSQL-databases (met een dataset van een paar Terabyte) een hoge performance kennen (Strauch, 2010).

Voordat we beginnen moeten we mededelen dat iedere uitgever van een databasemanagementsysteem zijn eigen functionaliteiten implementeert in een bestaand Data Model. De verschillende pakketten, die gebruik maken van eenzelfde datamodel, hebben wel globale lijnen, die hieronder worden beschreven.

Bij een Document Store systeem worden de data georganiseerd in documenten (niet te verwarren met bijvoorbeeld Word documenten). Elk document kan attributen hebben en vaak worden deze documenten opgeslagen in een collectie. Een database kan weer bestaan uit verschillende collecties. Deze documenten worden (vaak) opgeslagen en ook opgevraagd middels (B)JSON (Binary Java Script Object Notation)-objecten. Elk document heeft ook altijd een eigen identificerend attribuut. Standaard wordt dit altijd zelf gegenereerd en binnen een database zal een gegenereerd identificerend attribuut slechts één keer voorkomen.

Document Stores ondersteunen over het algemeen ook een secundaire index (naast de index voor het identificerend attribuut) en kunnen verschillende objecten opslaan zoals documenten, muziekbestanden, enzovoorts. Relaties binnen Document Stores worden meestal opgelost middels een boomstructuur. Dit wordt gedaan door het bijeenvoegen van een object of een array binnen een document of door het handmatig toevoegen van een refererend identificerend attribuut aan het document. Cascade opties zullen natuurlijk niet werken, want referentiële integriteit wordt niet ondersteund. Het bijeenvoegen van een object of een array wordt geadviseerd bij Document Store modellen (Plugge, Membrey, & Hawkins, 2010; Strauch, 2010; Cattel, 2010; Yunhua, et al., 2011).

ACID-transacties worden over het algemeen bij document stores niet ondersteund. Er zijn wel een aantal pakketten die via een omweg ACID-transacties op een bepaald niveau ondersteunen (CouchDB).

Enkele NoSQL-pakketten die gebruik maken van een datamodel, dat gebaseerd is op het Document Store, zijn:

- MongoDB
- CouchDB

Key Value Store Model
Key Value datamodellen zijn eenvoudig en kunnen vergeleken worden met een (hash)map in Java. In principe heb je een Key en een Value, waarbij de Key gedefinieerd kan worden door een externe gebruiker of applicatie. Met dit model kunnen clients data aanvragen middels een Key. Hierdoor kunnen deze modellen dus makkelijk geschaald en gedistribueerd worden, omdat elke Key/Value-combinatie onafhankelijk is van een andere Key/Value-combinatie.

De Key wordt bij het gros van het aanbod van Key/Value NoSQL pakketten opgeslagen als String en de Value als Blob. Naast deze voorwaarden is het aan de programmeur, hoe hij deze parst, enzovoorts. Key Value modellen richten zich vaak op schaalbaarheid ten koste van de consistentie van de ACID-theorie. Verder verwaarloost het model ook aggregeerfuncties.

De Key-waarde heeft een gelimiteerd bereik op het gebied van grootte in bytes. Deze limitatie is minder van toepassing voor het Value-waarde (Strauch, 2010).
Enkele NoSQL-pakketten, die gebruik maken van een datamodel en gebaseerd zijn op het Key Value Store Model, zijn:

- Dynamo
- Project Voldemort

Column Oriented of Extensible Record Store Model
Het Column Oriented Store Model wordt vaak ook de Extensible Record Store Model genoemd. Dit model lijkt erg op de Key Value Store. De Column Oriented Store kan ook gezien worden als multidimensionale Maps, waar Key Value Stores tweedimensionale Maps zijn.

Ten eerste, de Column Oriented Store werkt met kolommen. De kolom bevat bijvoorbeeld waardes en de kolom heeft ook waardes. Nu kunnen we een oneindig aantal kolommen maken, wat het onoverzichtelijk zou maken.

Ten tweede kunnen we kolommen (samen met hun waardes) groeperen, dit wordt een Column Family genoemd. Vanuit een relationeel standpunt kunnen deze Column Family's als tabellen gezien worden.

Ten derde is er ook nog de Super Column Family, waarbij we Column Family's kunnen groeperen.

Deze Column Oriented Store ondersteunt over het algemeen secundaire indexen. Secundaire indexen worden geplaatst op de waardes van kolommen, dus niet op de kolom zelf. Tevens is er geen referentiële integriteit aanwezig. De Column Oriented Store werkt het beste met gedenormaliseerde data. Deze data kunnen opgeslagen worden in groepen middels de (Super) Colomn Famliy structuur (Orielly, 2010).Enkele NoSQL-pakketten, die gebruik maken van een datamodel en gebaseerd zijn op het Column Oriented of Extensible Record Store, zijn:

- Google Big Table
- Cassandra

Hieronder worden een aantal populaire NoSQL database pakketten opgesomd:
HBase

- http://hbase.apache.org/
- Wordt onder andere gebruikt door: Facebook, Yahoo

Redis

- http://redis.io/
- Wordt onder andere gebruikt door: Craigslist

Cassandra

- http://cassandra.apache.org/
- Wordt onder andere gebruikt door: Facebook, Twitter

Project Voldermort

- http://project-voldemort.com/
- Wordt onder andere gebruikt door: LinkedIn

MongoDB

- http://www.mongodb.org/
- Wordt onder andere gebruikt door: GitHub, LexisNexis

CouchDB

- http://couchdb.apache.org/
- Wordt onder andere gebruikt door: Apple, BBC

Neo4J

- http://neo4j.org/
- Wordt onder andere gebruikt door: Adobe, Kamer van Koophandel