Datababases voor BIN

Voorwoord

Dit dictaat is geschreven om studenten BioInformatica van de Hanze Hogeschool Groningen te helpen met hun eerste wankele schreden bij het werken met relationele databases. Omdat het bedoeld is om in combinatie met de lessen gebruikt te worden ontbreekt de nuance hier en daar en is het verre van volledig. Alle opbouwende kritiek is vanzelfsprekend van harte welkom.

Arne Poortinga

Inhoudsopgave

1. Over dit blok

In dit vak leer je hoe je om moet gaan met realtionele databases. Omdat dit vak je drie EC oplevert en er veertien contactblokken @ 90 minuten ingeroosterd zijn wordt er verwacht dat je, buiten de lessen, zeven uur per week in dit vak zou moeten investeren om het te halen. Dit kan natuurlijk enorm variëren van persoon tot persoon. Het vak wordt beoordeeld aan de hand van een individueel computertentamen.

Voorkennis van het werken met de command line en tekst editors onder Linux, het zelfstandig lezen van documentatie en een basaal begrip van wiskundige verzamelingen wordt verondersteld

De leerdoelen van dit vak zijn samengevat in de toetsmatrijs op de volgende pagina’s.

Over dit blok

Leeruitkomsten	Kennis vragen	Inzicht vragen	Creatie/ontwerp	Punten
'De student is zelfstandig in staat om specifieke velden uit één tabel op te halen en deze te sorteren.'			1	2%
'De student is zelfstandig in staat om specifieke velden uit twee of meer tabellen op te halen middels de join- en/of de 'where-syntax' en deze eventueel te sorteren.'			2	8%
'De student is zelfstandig in staat om, gebruik makend van de group-by/having clausule, standaard sql-functies, subqueries, wildcards en het combineren van voorwaarden, queries op te stellen die willekeurige vragen over de opgeslagen data beantwoorden.			5 tot 6	30%
'De student is zelfstandig in staat om specifieke velden van bepaalde records te wijzigen en specifieke records toe te voegen dan wel te verwijderen.'			1 tot 2	10%
'De student is zelfstandig in staat			1 tot 2	10%

om veld- en tabeldefinities te manipuleren middels scripts c.q.middels de terminal.				
'De student is zelfstandig in staat om tabellen en foreign keys tussen tabellen te manipuleren middels scripts c.q.middels de terminal.			1 tot 2	10%
De student kan op basis van informatiebehoeften zelfstandig een database ontwerpen, of een bestaand ontwerp aanpassen, zodat alle noodzakelijke gegevens opgeslagen kunnen worden en de database voldoet aan de Boyce-Codd normaalvorm.			1 tot 2	30%

2. Tijdschema

week	**stof**
1	bestudeer hoofdstukken 1 t/m 5 en maak de opgaven
2	bestudeer hoofdstuk 6 en maak de opgaven
3	bestudeer hoofdstukken 7 en 8 en maak de opgaven
4	proeftentamen SQL
5	bestudeer hoofdstukken 9 en 10 en maak de opgaven
6	proeftentamen ontwerpen
7	vragen/extra oefeningen

3.Gegevens, data en informatie

'Data' is in het latijn het meervoud van 'datum'. 'Datum' is afgeleid van 'dare' (geven) en betekent dus niets anders dan 'gegeven'.
De begrippen 'gegevens' c.q. 'data' en 'informatie' worden vaak door elkaar gebruikt, maar zijn, als je het goed bekijkt, geen volledig identieke begrippen. Gegevens zijn niet meer dan dat, kale gegevens en hebben als zodanig weinig betekenis. Pas door gegevens met elkaar te combineren of te vergelijken krijgen ze betekenis en worden ze dus tot informatie. Dit lijkt misschien een onderscheid dat geen verschil maakt, maar hoe reageer je zelf op de volgende uitspraak: 'Hij weegt vijftig kilo'?
Waarschijnlijk denk je nu iets als 'Ja en? Wat moet ik daar nou mee? Wie is “hij”?'
Met andere woorden je vraagt om meer gegevens zodat je ze kunt combineren tot iets waar je wel wat mee kunt. Als “hij” bijvoorbeeld een volwassen vent van twee meter blijkt te zijn, zul je hem al snel aanraden zich te laten onderzoeken op een lintworm en vooral meer te eten, maar gaat het om een driejarige peuter dan zul je waarschijnlijk iets heel anders zeggen en als je te horen krijgt dat “hij” een zak aardappelen is die een trap opgesjouwd moet worden dan weet je dat je even bij de buren moet gaan kijken of je daar niet bent.

Dataopslag

Voordat je gegevens met elkaar kunt combineren zul je ze eerst moeten verzamelen en daarna zien te bewaren. Dat kan op losse kaartjes in een kaartenbak, maar als de hoeveelheden gegevens toenemen dan gaat een dergelijk systeem wel erg veel bomen kosten en bovendien zul je altijd zien dat meerdere gebruikers allemaal op hetzelfde moment dezelfde gegevens willen gebruiken. En dat geeft ruzie. Bovendien is het niet erg snel of makkelijk om specifieke gegevens uit een dergelijk systeem op te halen. Dat kan makkelijker en beter met een computer.
Een van de meest eenvoudige oplossingen zou kunnen zijn de gegevens in aparte, losse files in het bestandssysteem op te slaan en ze afhankelijk van de gewenste informatie te groeperen in directories. Dat een dergelijk systeem maar beperkt bruikbaar is, hoop ik aan de hand van het volgende voorbeeld duidelijk te maken.
Denk bijvoorbeeld aan een klantensysteem: als een klant belt, wil je snel de status van een bepaalde order op kunnen zoeken. Een oplossing hiervoor zou kunnen zijn: elke order in een apart bestand, met als naam het ordernummer, op te slaan. Je maakt een directory 'orders' met daarin al die bestanden en klaar is Kees. Totdat aan het einde van de maand de financiële afdeling de rekeningen wil gaan versturen. Die hebben er geen zin in elke order apart te gaan bekijken of die al betaald is en dus wordt het systeem gereorganiseerd. De openstaande orders komen in een andere subdirectory dan de afgehandelde orders en om het de financiële boys helemaal makkelijk te maken, worden de orders die langer dan twee maanden open staan in een aparte directory gezet, zodat daar makkelijk een boze brief achteraan gestuurd kan worden.
Maar nu gaat klantenservice weer zeuren dat het systeem niet werkt omdat bellende klanten vaak niet weten of, laat staan wanneer, hun eigen (externe) boekhouder een order betaald heeft en zij niet steeds op drie plaatsen naar informatie willen zoeken. Dat kost te veel tijd en dan worden de bellende klanten weer boos. Dan kun je er voor kiezen dat elke afdeling z'n eigen copie van de gegevens krijgt. Maar dan gaan allebei de afdelingen steigeren omdat ze veranderde gegevens op meer plaatsen bij moeten werken. Zo zal bijvoorbeeld de boekhouding ook in het systeem van de klantenservice een order op 'betaald' moeten zetten.
Het zal duidelijk zijn dat een dergelijk systeem, waarbij de structuur van de gegevens bepaald wordt door de gevraagde informatie, behalve in enkele specifieke gevallen, niet lang goed werkbaar blijft.

Al die copieën van de zelfde gegevens kosten veel opslagruimte en het wordt steeds lastiger om de gegevens in al die copieën aan elkaar gelijk te houden. Denk maar aan een marketingafdeling die leuke acties gaat verzinnen als 'tien procent korting voor klanten die de laatste drie maanden elke maand voor meer dan een bepaald bedrag besteld en betaald hebben'. Dan loop je snel tegen de grenzen van een dergelijk systeem aan.

Een goed databeheersysteem zou flexibel genoeg moeten zijn om de opgeslagen data telkens op andere manieren te combineren zonder ze telkens opnieuw op te slaan c.q. zonder de onderliggende opslagstructuur daarvoor aan te moeten passen.
Een oplossing hiervoor kan in de wiskunde gevonden worden. Meer specifiek in de relationele algebra (a.k.a. verzamelingenleer) waar je vorig jaar kennis mee gemaakt hebt bij het vak wiskunde. Een kenmerk van een relationele database is dat gegevens in tabellen opgeslagen worden, met in elke tabel alleen maar gegevens van dezelfde soort. In een studentenbeheersysteem krijg je dus een tabel met daar in alle studenten en een aparte tabel met hun resultaten. Hier wordt later op terug gekomen.

Programma's die gegevens beheren heten DataBase Managament Systemen (DBMS). Een subgroep hiervan, die gebaseerd zijn op de relationele algebra, heten 'Relational DataBase Management Systems' (RDBMS). De combinatie van de door een RDBMS beheerde gegevens en het RDBMS zelf heet 'relationele database'. Dit dictaat gaat over het hoe en waarom van relationele databases zodat je daar zelf op een goede manier mee om kunt gaan. Omdat relationele databases veel meer gebruikt worden dan andere soorten databases, zijn 'database' en 'relationele database' bijna synoniemen geworden. In dit dictaat wordt met database dan ook, behalve als het duidelijk anders aangegeven wordt, een relationele database bedoeld.

Relationele databases zijn niet zaligmakend. Omdat de basis van elke relationele database gevormd wordt door het ‘schema’ (welke gegevens er worden opgeslagen en in welke tabellen) zijn ze tamelijk inflexibel. Verder zorgt het gebruiken en afdwingen van het volgen van een schema voor de nodige overhead. Voor toepassingen waarbij een schema in de weg zit omdat het te star is, of niet nodig is omdat de opgeslagen data een simpele structuur hebben (zoals bijvoorbeeld tijdsreeksen) kunnen andere databasemanagementsystemen vaak een betere keuze zijn.

4. De (R)DBMS, de droom van elke IT-er

Een DBMS, alias DataBase Manager, is een programma dat de opslag van en de toegang tot de gegevens in een database regelt. In tegenstelling tot menselijk management doet de database manager feilloos waar hij voor bedoeld is.

Database managers zijn er in soorten en maten. Bekende namen zijn bijvoorbeeld SQL-server, Oracle, SyBase, Ingress, Progress, Access, ... Met Python wordt SQLite meegeleverd. Op je desktop zijn drie databasemanagers beschikbaar, OpenOffice.org Base, sqlite3 en MariaDB, een fork van mySQL. De laatste is de database manager die jullie gaan gebruiken. Al deze managers spreken (een dialect van) SQL.

Omgang met de database manager

De database manager leeft op een databaseserver en beheert alle data die hem op de correcte manier aangeboden worden. Dat kunnen meerdere losse sets aan gegevens zijn. Binnen de context van een database manager wordt zo'n set aan bij elkaar horende gegevens ook een database genoemd. Een betere naam is echter **schema.** Schema is ook de formele benaming van een database ontwerp. Op het hoe en waarom van schema's komen we later terug.

inloggen en algemene omgangsvormen

Omdat hij z'n verantwoordelijkheid serieus neemt verifiëert de database manager altijd wie hem een opdracht geeft. Daarom begint een database sessie altijd met inloggen. Je geeft de manager je inlognaam, je wachtwoord en (eventueel) het schema dat je wil benaderen en als alles klopt kun je de RDBMS vragen om gegevens op te halen of andere opdrachten uit te voeren.
Omdat een DBMS geen oneindig aantal sessies tegelijkertijd aan kan, log je na afloop van een sessie weer uit.

Voor de communicatie met de database manager die jouw database beheert gebruiken we een CLI-tool genaamd `mysql`. (`mariadb` mag je ook gebruiken, dit is op onze systemen een link naar m`ysql`). Zoals bij (bijna) elke commandline tool kun je middels opties en parameters het gedrag van die tool beïnvloeden. Zie hiervoor ook de bijbehorende man-pagina.

Om in te loggen open je een terminal en tikt daarin de naam van die tool (`mysql`) gevolgd door de opties voor de user (`-u`), het te gebruiken schema (`-D`), de server waar de databasemanager op 'leeft' (`-h`), voor dit vak `mariadb.bin`) en het wachtwoord (`-p`):

```
mysql -u <je login> -D <je login met een hoofdletter> -h mariadb.bin -p
<return>
```

Tik nu je wachtwoord in en je hebt contact met de databasemanager.

Het is handig als je een `alias` (zie de relevante man-page) aanmaakt met daarin je loginnaam, het schema, de databaseserver en de optie waarmee je aangeeft dat het wachtwoord nog ingegeven moet worden. Je wachtwoord sla je natuurlijk nergens op.
Een alternatief kan het gebruik van een `.my.cnf` bestand zijn (zie de mySQL documentatie).

Het is handig om te weten dat veel database managers, waaronder mySQL, een opdracht pas herkennen als die afgesloten is door een puntkomma.
Het commando om uit te loggen is 'exit'. Om uit te loggen tik je dus

```
exit;<return>
```

Een van de eigenaardigheden van de mySQL datamanager, namelijk dat hij er op staat dat een statement met een puntkomma beëindigd wordt, heb ik al genoemd. In dat kader is het ook handig om te weten dat hij er op staat dat strings tussen enkele quotes staan. Binnen een string kun je soms wildcards gebruiken; '%' komt overeen met de reguliere expressie '.*' en '_' met de reguliere expressie '.'.
Weliswaar worden door veel programmeurs verzoeken aan de mySQL datababase manager in upper case gedaan, maar dat is louter folklore. Hij herkent ook lower cased of mixed case verzoeken.

scripts

Als je een serie sql-opdrachten achter elkaar zet in een tekstbestand heb je een script. Je kunt de mySQL datababase manager verzoeken de statements in dit script uit te voeren door het script als invoer te laten gebruiken door de mysql client. In z'n meest kale vorm ziet die redirectie er als volgt uit:

```
mysql < pad_naar_mijn_sql_script
```

Een alternatief is dat je een pipe gebruikt:

```
cat pad_naar_mijn_sql_script | mysql
```

Natuurlijk kun je bij bovenstaande voorbeelden ook allerlei opties toevoegen, wat je in de praktijk (als je geen .my.cnf of alias gebruikt om in te loggen) dus ook zult doen.
Een alternatieve manier om een script uit te voeren is eerst in te loggen in de mysql client en dan het script uit te voeren. Zie hiervoor de help functie van de client.

Zoals alle scripts/programma's moet ook een sql-script voorzien zijn van commentaar. Alles wat op een regel na een dubbele dash (--) komt en alles wat over meerdere regels tussen /* en */ staat wordt gezien als commentaar.

documentatie en tutorials

Omdat je in de praktijk altijd toegang zult hebben tot documentatie is het belangrijk dat je die leert lezen. Dat betekent dus dat je moet begrijpen hoe een en ander werkt. Het betekent ook dat het zinloos is om allerlei syntax *zonder* dat je het begrijpt uit je hoofd gaat leren of SQL-statements (queries) zonder ze te begrijpen ergens anders vandaan haalt. De details van de syntax heb je trouwens altijd tot je beschikking als je in de commandline tool zit. Als je ingelogd bent heb je namelijk de beschikking over een aantal in de cli-tool ingebouwde commandos. Een ingebouwd commando dat je veel zult gebruiken is `quit` (of zijn synoniem `exit`). Verreweg de belangrijkste is mijns inziens echter `help`.
Met gewoon `help` krijg je een overzicht van alle beschikbare commandos, wil je meer weten over de syntax van een bepaald commando dan tik je

```
help <commando>;<return>
```

Bijvoorbeeld:

```
help insert;<return>
```

Alle informatie over de gebruikte database manager is beschikbaar op https://mariadb.com/kb/en/mysql-command-line-client/ en https://mariadb.com/kb/en/documentation/ . Natuurlijk mag je tijdens de lessen ook andere documentatie gebruiken, maar tijdens het tentamen heb je alleen de beschikking over de ingebouwde documentatie in de mySQL client zelf en de man pages.

Om de officiële documentatie goed te kunnen gebruiken is het misschien niet noodzakelijk maar wel erg makkelijk om inzicht te hebben in het hoe en waarom van SQL. Behalve deze handleiding met bijbehorende oefeningen is er op het internet van alles te vinden. Aangeraden sites zijn:

youtube.com
www.tutorialspoint.com/mysql
http://www.mysqltutorial.org

Opdrachten:

1. maak een alias waarmee je op de database server in kunt loggen
2. bestudeer op https://dev.mysql.com/doc/refman/8.0/en/manual-conventions.html hoe de documentatie vorm gegeven is.
3. Maak jezelf iets meer bekend met de command line client tool (man mariadb of https://mariadb.com/kb/en/mysql-command-line-client/
4. zorg dat je weet hoe je in de command line tool hulp kunt vragen (https://mariadb.com/kb/en/help-command/)
5. bestudeer de documentatie op https://mariadb.com/kb/en/set-password/ en verander je wachtwoord in iets wat je wel kunt onthouden.
6. Voer het script dat in `studenten.sql` staat uit en
7. bekijk welke tabellen er gemaakt zijn met het commando `show tables;` en bestudeer de syntax hiervan in de documentatie. (https://mariadb.com/kb/en/show/)
8. zorg dat je weet wat voor datatypen er beschikbaar zijn (https://mariadb.com/kb/en/data-types/)
9. Vraag in de klas als iets onduidelijk is!

5. SQL

SQL is het acronym van Structured Query Language. Het is de taal die begrepen wordt door de meeste database managers. In tegenstelling tot de talen die je tot nu toe geleerd hebt, is SQL geen 3e, maar een 4e generatie taal. Dat wil zeggen dat je alleen maar hoeft te vertellen *wat* je wil en niet meer *hoe* het gedaan moet worden.

De taal valt in twee delen uiteen. Een deel voor het beschrijven hoe de gegevens er uit zien en wat de gewenste tabelstructuur is, de Data Definition Language (**DDL**) en een deel voor het manipuleren van de opgeslagen gegevens, de Data Manipulation Language (**DML**). Van beide deeltalen zal in dit dictaat genoeg behandeld worden om je in staat te stellen de online documentatie van mySQL te lezen, maar ook niet veel meer! Ten eerste heeft het weinig zin informatie die al makkelijk beschikbaar is nog eens over te tikken. Ten tweede zul je toch ook zelf moeten leren documentatie van computertalen te lezen en te begrijpen Oefen dat dus ook!

Behalve specifieke DDL en DML statements kent SQL operatoren voor rekenkundige bewerkingen (optellen, aftrekken etc.), de normale vergelijkingen (groter dan, kleiner dan, ...) en de normale logische operatoren (and, or, not).
Elke database manager kent z'n eigen basale datatypen en functies. De `show` functionaliteit zul je bijvoorbeeld niet in elke database manager tegen komen.

Nog even een paar begrippen die je nu nodig hebt, maar pas later in de syllabus toegelicht worden:

- record: de gegevens over een ding in een database, bijvoorbeeld de basisgegevens van een student. Je kunt een record ook zien als een rij of regel in en tabel.
- Attribuut: ook wel veld genoemd. Dit kun je zien als een kolom uit een tabel die een onderdeel van een record bevat. Voornaam is bijvoorbeeld een attribuut van een student, net zoals studentnummer. Een attribuut is altijd van een bepaald type. Als je bijvoorbeeld een attribuut hebt dat de temperatuur in ºK weergeeft, dan is dit altijd een numeriek veld met een waarde groter dan nul.
- primary key: dit is het veld of de combinatie aan velden die een record identificeert. Meestal is dit een numeriek veld.
- Foreign key: dit is een veld of een combinatie van velden die verwijst c.q. verwijzen naar de primary key in een andere tabel.

DDL

Voordat je iets anders kunt doen met een database, zul je eerst de database manager moeten vragen tabellen te maken waar de gegevens in opgeslagen kunnen worden.
Je zult, als je een tabel wil laten aanmaken ook details over die tabel, zoals de naam, de namen en de datatypen van de kolommen, de primary key en eventuele foreign keys mee willen geven.
Als je in de online reference manual van mySQL gaat kijken, dan zie je onderstaande syntax:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_option] ...
```

Items tussen blokhaken zijn optioneel, de ellipsis geeft een herhaling aan en de pipe een keuze. Als er een keuze uit een groep gemaakt moet worden staat de groep tussen accolades. de keywords staan in hoofdletters (N.B. de manager snapt het ook als je de opdracht in kleine letters geeft!) , de woorden in kleine letters moet je door iets anders vervangen.

SQL doet overigens de S uit z'n naam alle eer aan. Als je bijvoorbeeld naar de create statement kijkt waarmee je tabellen en dergelijke aanmaakt, dan valt op dat die bestaan uit een keyword (`create`) gevolgd door wat je wilt dat de database manager voor je aanmaakt. Als je op deze logische manier naar de taal kijkt kost het leren er van veel minder inspanning dan wanneer je alles maar botweg in je hoofd probeert te stampen.

Kijk je wat verder op de pagina, dan zie je dat `create_definition` en `table_option` ook gedefiniëerd worden.

```
create_definition:
    col_name column_definition
  | [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...)
        [index_type]
  | {INDEX|KEY} [index_name] [index_type] (index_col_name,...)
        [index_type]
  | [CONSTRAINT [symbol]] UNIQUE [INDEX|KEY]
        [index_name] [index_type] (index_col_name,...)
        [index_type]
  | {FULLTEXT|SPATIAL} [INDEX|KEY] [index_name] (index_col_name,...)
        [index_type]
  | [CONSTRAINT [symbol]] FOREIGN KEY
        [index_name] (index_col_name,...) reference_definition
  | CHECK (expr)

column_definition:
    data_type [NOT NULL | NULL] [DEFAULT default_value]
      [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
      [COMMENT 'string'] [reference_definition]
```

Onder alle syntax beschrijvingen volgt nog een toelichting.

Als je bijvoorbeeld een relatie naar voorbeeld van tabel 1 aan zou willen laten maken, dan begin je bij het begin van de pagina. Optionele items negeren we nu zoveel mogelijk. De woorden `TEMPORARY` en `IF NOT EXISTS` vervallen dus. `tbl_name` staat voor tabelnaam (duh) dus vervangen we dat door 'soorten'.

Het begin van je statement wordt dus:

```
CREATE TABLE soorten
```

Dit deel moet, tussen haakjes, gevolgd worden door één of meer 'create_definitions'. Volgens de pagina bestaat een create_definition uit een 'col_name' samen met een 'column_definition' of een boel andere meuk. Het zal wel duidelijkzijn dat als je een kolom aan wil maken, wat je normaliter als eerste doet, je de daarvoor de combinatie van col_name en column_definition moet hebben. Als je wat verder kijkt, dan zie je ook dat een column_definition in ieder geval bestaat uit het gewenste datatype.
Omdat beide velden deel uitmaken van je primary key mogen ze niet null zijn (we komen hier later op terug). Dat geef je aan met NOT NULL. Ook geef je aan wat de primary key van de tabel moet zijn.
De statement voor het aanmaken van een soortentabel die er uit ziet zoals in tabel 2 (uit hoofdstuk 9) ziet er dus als volgt uit:

```
CREATE TABLE soorten (
          genusnaam VARCHAR(30) NOT NULL,
          speciesnaam VARCHAR(30) NOT NULL,
          PRIMARY KEY (genusnaam, speciesnaam)
);
```

Let er op dat extra whitespace voor de manager geen betekenis heeft en dat het statement afgesloten moet worden door een puntkomma.

Het keyword `CONSTRAINT` geeft aan dat er een eis , een beperking, waar de data aan moet voldoen geldt. Een dergelijke 'eis' kunnen bijvoorbeeld een pk (primary key)of een fk (foreign key)zijn. Omdat het optioneel is en verder niets toevoegt aan het aanmaken van de pk, is het hier weggelaten.

Na deze inleiding zou je verder zelf in staat moeten zijn om statements te schrijven waarmee je de structuur van een tabel aanpast (`alter table`) of een tabel verwijdert (`drop table`).

Als je wil controleren wat er in jouw schema staat dan kan dat met de volgende statement:

```
mysql> show tables;
```

Je krijgt dan alle tabellen die in jouw schema staan te zien. Wil je de tabelstructuur van de soorten tabel bekijken, dan geef je de volgende opdracht:

```
mysql> show columns from soorten;
```

of:

```
mysql> describe soorten;
```

Opdrachten:

1. bestudeer de syntax van 'show' (https://mariadb.com/kb/en/show/) en zorg er voor dat je de code voor het bekijken van aanwezige tabellen op kunt vragen
2. bestudeer de code van `studenten.sql` en de documentatie van de de statements die daar in staan. Zorg er voor dat je de documentatie kunt lezen en dat je de statements zelf ook kunt gebruiken.
3. Voer het script `studenten.sql` nogmaals uit. Als het goed is krijg je een foutmelding. Pas het script zo aan dat het gaat werken.
4. Maak een script om onderstaande tabel in je database te krijgen. Het veld id moet de primary key worden, gebruik hiervoor een auto_increment. Je kunt studenten.sql als voorbeeld gebruiken, maar bestudeer wel de documentatie van de statements die je gebruikt en zorg dat je begrijpt wat je doet!

id	genusnaam
1	Anas
2	Alces
3	Homo

Tabel 1: Genera

6. DML

Het manipuleren van gegevens bestaat uit het opslaan (`insert`), het aanpassen (`update`), verwijderen (`delete`) en raadplegen (`select`) van gegevens.
Je moet bij al deze statements aangeven op welke tabel de statement van toepassing is. Manipuleer je bestaande gegevens, dan zul je ook aan moeten geven welk record je wil bewerken. Dat doe je met de '**where**'-clausule. Als een record voldoet aan de voorwaarden die in deze clausule staan, dan wordt de statement op dit record uitgevoerd.
Je moet je wel terdege realiseren dat SQL op dataverzamelingen werkt. Met andere woorden *àlle* records die aan de where-clausule voldoen worden beïnvloed door de statement! For- of while loops zoals je die kent uit andere talen zitten al impliciet in elke SQL statement ingebakken!

De insert-, update- en delete statements zijn heel recht toe recht aan en zullen hier ook niet verder besproken worden, bestudeer die dus zelf. De `select` statement is een stuk complexer en zal dus wel deels toegelicht worden.

select

Met het keyword '`select`' vraag je de database manager informatie op het scherm te printen. Je kunt de mySQL datababase manager bijvoorbeeld vragen hoe laat het is. De mySQL functie `current_time()` geeft de tijd terug en door het resultaat van die functie te laten selecteren print de database manager het resultaat uit in de terminal.

```
mysql> select current_time();
+----------------+
| current_time() |
+----------------+
| 13:14:42       |
+----------------+
1 row in set (0.00 sec)
```

Je zou ook op kunnen tellen;

```
mysql> select 1 + 1;
+-------+
| 1 + 1 |
+-------+
|     2 |
+-------+
1 row in set (0.00 sec)
```

Zoals je ziet geeft de database manager keurig een tabelletje (een relatie!) terug. In de voorbeelden bestaat de relatie maar uit een enkele kolom met een enkele waarde.

Je kunt de naam van de kolom in het resultaat aanpassen door de `as` clausule (let op de enkele quotes).

```
mysql> select current_time() as 'het is nu';
+-----------+
| het is nu |
+-----------+
| 13:23:55  |
```

```
+-----------+
1 row in set (0.00 sec)
```

Meestal zul je de select clausule gebruiken om de database manager te vragen om alleen specifieke kolommen weer te geven. Bijvoorbeeld de query

```
select naam from studenten;
```

laat alleen de kolom ‘naam’ uit de tabel ‘studenten’ zien, terwijl de query

```
select voorletters, naam from studenten;
```

de inhoud van de kolommen ‘voorletters’ en ‘naam’ laat zien. Wil je alle kolommen zien, dan gebruik je als wildcard de asterisk.
Bijvoorbeeld met de query

```
mysql> select * from studenten;
```

Behalve kolomnamen kan de select clausule ook de functies `COUNT(attribuuttype)`, `COUNT(*)`, `SUM(attribuuttype)`, `AVG(attribuuttype)`, `MAX(attribuuttype)` en `MIN(attribuuttype)` bevatten.

Zoek zelf uit wat deze functies doen, je hebt die kennis nodig!

From

Als je informatie uit een bepaalde tabel wil halen, dan zul je de database manager moeten vertellen uit welke tabel de informatie moet komen, maar ook welke kolommen er geselecteerd moeten worden. Wil je alle kolommen zien, dan gebruik je als wildcard de asterisk.
Bijvoorbeeld met de query

```
mysql> select * from studenten;
```

krijg je alle informatie die in alle records van de soorten-tabel staat te zien. Pas hiermee op! Bij sommige databases kan dit in de miljoenen records lopen!

Where

Vaak wil je alleen maar de gegevens van een bepaalde student zien. Dan moet je een voorwaarde stellen waaraan de records moeten voldoen.
De volledige query kan er dus als volgt uit zien:

```
mysql> select * from studenten where naam = 'Jansen';
```

In bovenstaand voorbeeld staat een voorbeeld van een eenvoudige voorwaarde, maar omdat wat achter de where komt een geldige booleaanse expressie moet zijn, kun je met behulp van de standaard booleanse sleutelwoorden (`and`, `not`, `or`) hier arbitrair complexe combinaties van voorwaarden neerzetten.

IS

In de where clausule zet je, zoals gezegd, de voorwaarden waar de te selecteren records aan moeten voldoen. Vaak zullen dat gewone vergelijkingen zijn die je met logische operatoren kunt combineren. Omdat bij de ontwikkeling van SQL veel wiskundigen betrokken zijn geweest, is het hier en daar wat rechtlijniger dan veel andere talen. Omdat null betekent 'niet gedefiniëerd' mag je geen directe vergelijking maken met deze waarde. een vergelijking als

```
naam = null
```

mag dus niet want het =-teken betekent 'is gelijk aan' en dat impliceert kennis van de waarde van het item aan de rechterkant van de vergelijking en die is niet gedefinieerd! De oplossing voor de problemen die hier uit voorkomen is de IS-operator. De voorwaarde dat de waarde van het veld 'naam' niet gedefinieerd is, luidt in correct SQL:

```
naam IS null
```

LIKE

Om dezelfde reden mag je als je gebruik wilt maken van wildcards de normale vergelijkingsoperatoren niet gebruiken. In plaats daarvan gebruik je de `LIKE`-operator.
SQL kent twee wildcards voor het vergelijken van strings. De % vervangt een willekeurig aantal karakters, de _ precies één.

IN

De where clausule kent ook de mogelijkheid om als voorwaarde te stellen dat een waarde zich in een lijst moet bevinden. Dat is de `IN`-operator.
Om bijvoorbeeld te kijken wat de gegevens zijn van meer dan één student kun je de volgende opdracht gebruiken:

```
mysql> select * from studenten where naam in ('Jansen', 'Boer');
```

Group by

Met de deze clausule kun je rijen groeperen en eventueel kun je aan deze groep voorwaarden stellen door middel van de HAVING clausule. Wil je bijvoorbeeld weten hoeveel examens elke student gemaakt heeft, dan kun je de COUNT(*) functie gebruiken op de per stud_id gegroepeerde records. Deze clausule gebruik je alleen als je iets *gegroepeerd per …* weer wil geven.

```
MariaDB [Arne]> select stud_id, count(*) from examens group by stud_id;
+---------+----------+
| stud_id | count(*) |
+---------+----------+
| hofm    |        3 |
| jans    |        2 |
| mole    |        1 |
| nieu    |        3 |
| spaa    |        2 |
+---------+----------+
```

```
5 rows in set (0.001 sec)
```

ORDER BY

Omdat de leden van een verzameling per definitie niet geordend zijn, kun je er in principe niet van uit gaan dat de items die een query terug geeft dat wel zijn, of zelfs dat ze altijd op dezelfde volgorde in het resultaat verschijnen!
Om wel een bepaalde volgorde af te dwingen kun je aan het eind van de query de 'order by' clausule toevoegen. Zie de documentatie voor meer details.

Volgorde van bewerkingen

De database manager voert in principe (de werkelijkheid kan hier sterk van afwijken) de instructies die in een SQL-query staan stap voor stap uit en elke stap levert een nieuwe, tijdelijke tabel (relatie) als resultaat op, die dan weer als input voor de volgende stap gebruikt wordt.
De volgorde waarin de stappen uitgevoerd worden is niet gelijk aan die waarin ze in de query staan!

Een volledige select-query ziet er als volgt uit:

```
select ...from ....where ...group by ...having ...order by ....
```

1. Als eerste wordt de FROM-clausule uitgevoerd. Hierin worden alle gegevens uit de aangegeven tabel (c.q. tabellen) gehaald
2. Op de resultaattabel van de FROM-clausule worden vervolgens door de WHERE-clausule alle ongewenste records verwijderd. Dit levert een nieuwe resultaattabel op.
3. Vervolgens worden de records volgens de voorwaarden in de GROUP BY-clausule gegroepeerd en de daaruit ontstane resultaattabel gefilterd door de HAVING-clausule.
4. De nieuwe resultaatset wordt gesorteerd door de ORDER BY clausule en, dus pas als laatste, worden de ongewenste attribuuttypen verwijderd door de SELECT-clausule.

Opdrachten:

Maak de queries die antwoord geven op de volgende vragen:

1. selecteer de achternamen van alle studenten in aflopende volgorde (z->a)
2. selecteer de cursussen, maar vervang in het kolomkopje van het resultaat 'cur_id' door code
3. selecteer de studenten die een a in hun achternaam hebben
4. hoeveel studenten wonen er in groningen?
5. laat zien welke studenten in groningen of assen wonen. Geef de naam en de woonplaats
6. Laat per woonplaats zien hoeveel studenten er wonen. Geef alleen de woonplaats en het aantal studenten en sorteer op alfabetische volgorde van woonplaats.
7. welke studenten zijn tussen 1990 en 1993 geboren?
8. welke niet?
9. Bestudeer de syntax die je nodig hebt om een student toe te voegen en die je nodig hebt om een student te verwijderen?
10. Voeg een student toe en verwijder hem weer.
11. Bestudeer de syntax die je nodig hebt om de naam van een student te wijzigen?
12. Voeg een student toe, wijzig zijn voornaam en verwijder hem weer.
13. Bestudeer de syntax die je nodig hebt om een kolom aan een tabel toe te voegen?
14. Voeg aan de tabel studenten een kolom toe met de naam ‘burgerlijke_staat’ en kies daarvoor een geschikt datatype.
15. Verwijder de kolom die je toegevoegd hebt weer.

7. Meerdere tabellen

Het idee van een relationele database is dat je gegevens uit verschillende tabellen kunt combineren tot nieuwe informatie. Het combineren van tabellen kan op twee manieren.
In dit hoofdstuk gaan we uit van de volgende twee tabellen:

```
mysql> select * from soorten;
+-----------+---------------+
| genusnaam | speciesnaam   |
+-----------+---------------+
| Alces     | alces         |
| Anas      | clypeata      |
| Anas      | platyrhynchos |
+-----------+---------------+
3 rows in set (0.00 sec)

mysql> select * from genera;
+-----------+-------------------+
| genusnaam | genusomschrijving |
+-----------+-------------------+
| Anas      | zoetwater eenden  |
| Alces     | NULL              |
+-----------+-------------------+
2 rows in set (0.00 sec)
```

De eerste, niet zo nette, manier om tabellen te combineren is in de FROM-clausule de tabellen, gescheiden door komma's, op te nemen.
De manager maakt dan een **carthesisch product** van alle tabellen. De resultaattabel ziet er dan als volgt uit:

```
mysql> select * from genera, soorten;
+-----------+-------------------+-----------+---------------+
| genusnaam | genusomschrijving | genusnaam | speciesnaam   |
+-----------+-------------------+-----------+---------------+
| Anas      | zoetwater eenden  | Anas      | platyrhynchos |
| Alces     | NULL              | Anas      | platyrhynchos |
| Anas      | zoetwater eenden  | Anas      | clypeata      |
| Alces     | NULL              | Anas      | clypeata      |
| Anas      | zoetwater eenden  | Alces     | alces         |
| Alces     | NULL              | Alces     | alces         |
+-----------+-------------------+-----------+---------------+
6 rows in set (0.00 sec)
```

De overtollige records moet je wegfilteren met behulp van de WHERE-clausule. Een eland is tenslotte geen zoetwatereend.

```
mysql> select * from genera g, soorten s
    -> where g.genusnaam = s.genusnaam;
+-----------+-------------------+-----------+---------------+
| genusnaam | genusomschrijving | genusnaam | speciesnaam   |
+-----------+-------------------+-----------+---------------+
| Anas      | zoetwater eenden  | Anas      | clypeata      |
| Anas      | zoetwater eenden  | Anas      | platyrhynchos |
| Alces     | NULL              | Alces     | alces         |
+-----------+-------------------+-----------+---------------+
```

```
3 rows in set (0.00 sec)
```

Omdat de kolomnaam 'genusnaam' in beide tabellen voorkomt en de database manager zeker wil weten welke kolom uit welke tabel precies bedoeld wordt, moet je in dit soort situaties de tabelnaam met een punt voor de kolomnaam plakken. Je kunt hier, net zoals in het voorbeeld, een **alias** voor gebruiken. Een alias is een afkorting voor een tabelnaam. In de FROM-clausule geef je aan welke alias je voor welke tabel wil gebruiken. Altijd als de database manager niet zeker kan weten welke kolom uit welke tabel hij moet gebruiken, is het gebruik van een alias verplicht. Ook als het niet verplicht is, kan het gebruik van een alias handig zijn. 'S' is tenslotte minder tikwerk dan 'soorten';

Omdat er bij de FROM-clausule een carthesisch product van de deelnemende tabellen als resultaatset geeft, kun je je voorstellen dat dit voor de database manager hard werken is. Er is ook een andere manier om tabellen met elkaar te combineren die dit bezwaar niet heeft. Je geeft dan per tabelcombinatie aan op welke velden de tabellen samengevoegd ('gejoined') moeten worden. Vergelijk de onderstaande resultaattabel maar eens met de resultaattabel van de FROM-clausule zonder JOIN. Je hebt de hele WHERE-clausule nu niet eens nodig. Met name bij ingewikkelde queries met meerdere grote tabellen kan het gebruik van de JOIN een forse performance winst opleveren. Daarom verwacht ik ook dat je deze vorm gebruikt.

```
 mysql> select * from genera g join soorten s on g.genusnaam = s.genusnaam;
+-----------+-------------------+-----------+---------------+
| genusnaam | genusomschrijving | genusnaam | speciesnaam   |
+-----------+-------------------+-----------+---------------+
| Anas      | zoetwater eenden  | Anas      | clypeata      |
| Anas      | zoetwater eenden  | Anas      | platyrhynchos |
| Alces     | NULL              | Alces     | alces         |
+-----------+-------------------+-----------+---------------+
3 rows in set (0.00 sec)
```

Het resultaat van een join is dat een tabel in het geheugen staat met daar in gegevens uit de twee betrokken tabellen. Vaak komt het voor dat je uit meer dan twee tabellen gegevens wil combineren. Hoe pak je dat aan? Eigenlijk heel simpel (maar dat is alles als je het weet). Je voert een join uit tussen de tabel in het geheugen en de volgende tabel die je nodig hebt. En eventueel nog eens, en nog eens, en …

Bijvoorbeeld je wil informatie uit drie tabellen, A, B en C.

Stap 1: join de tabellen A en B.

```
select * from A join B on A.a = B.a;
```

Het resultaat is een tabel in het geheugen.

Stap 2: join tabel C op de de resulaattabel in het geheugen (de nieuwe code staat vet cursief geprint).

```
select * from  from A join B on A.a = B.a
        join C on A.c = C.a;
```

Opdrachten:

1. Laat de resultaten van het vak ‘databases 1’ zien.
2. Idem, maar gebruik de ‘where’-syntax.
3. Laat de resultaten voor de beide database vakken zien, gesorteerd per vak.
4. geef voor de studenten die meer dan 1 examen afgelegd hebben hun code en hun gemiddelde?
5. geef van de studenten die meer dan een examen hebben gedaan hun naam en woonplaats
6. geef code, achternaam en laatste examendatum van de studenten die gemiddeld meer dan een 5 gehaald hebben over al hun examens, biologie 1 daarbij NIET meegerekend
7. welke studenten hebben zich al wel voor een examen ingeschreven maar nog geen punt gehaald? geef code en examendatum en cursuscode?
8. Schrijf de nieuwe student 'Dirty Harry' die in 'El Pueblo de Nuestra Señora la Reina de los Ángeles del Río de Porciúncula' woont in. (let op de waarschuwingen!)
9. laat zien welke studenten elke mentor heeft. gebruik de join syntax (moeilijk!). N.B. Een mentor is een student. Geef alleen de namen.

8. Subqueries

In vergelijkingen kun je ook het resultaat van andere queries gebruiken. Als je bijvoorbeeld uit een lijst met personen de oudste op wil halen zul je eerst de 'laagste' geboortedatum uit tabel willen halen om daarna daarmee de gewenste records uit de tabel te filteren.
Het ophalen van de 'laagste'geboortedatum ziet er bijvoorbeeld zo uit:

```
select min(geb_datum) from studenten;
```

Dit resultaat wil je dus gebruiken om de geboortedata van alle studenten mee te vergelijken. De query komt dus in de where-clausule van de hoofdquery te staan:

```
select * from studenten where geb_datum = (select min(geb_datum) from
studenten);
```

Net als in andere progammeertalen wordt eerst het deel wat tussen haakjes staat uitgevoerd en vervangen door het resutaat.

Let er op dat je de goede operator gebruikt. Als een subquery, zoals in bovenstaand voorbeeld, één waarde teruggeeft, dan kun je ‘=’ gebruiken. Geeft de subquery daarentegen een lijst terug dan moet je het resultaat gebruiken in combinatie met de `IN`-operator.

Je kunt ook controleren of een query wel of geen resultaat terug geeft met behulp van de EXISTS-operator.

Opdrachten:

1. wie is de oudste student en wanneer is die geboren?
2. wie hebben de hoogste cijfers behaald (voor alle cursussen samen, dus niet per cursus). Geef de namen van de studenten.
3. De student met id parn heeft - natuurlijk ;o) - voor het vak databases2 een 10 gehaald. Voeg dit resultaat toe.
4. wie hebben de hoogste cijfers behaald (voor alle cursussen samen, dus niet per cursus). Geef voorletters, naam, naam van het vak en het cijfer.
5. de student met de achtenaam bakker heeft gisteren voor het vak databases2 een 7 gehaald. Voeg dit toe aan de database.

9. Relationele Databases

In de eerste hoofdstukken is uitgelegd dat een databeheersysteem flexibel genoeg moet zijn om steeds weer andere informatie uit dezelfde set aan basisgegevens te halen. Met andere woorden een dergelijk systeem moet in staat zijn de basisgegevens ad hoc op een willekeurige manier kunnen groeperen. Ook zijn een aantal basisbegrippen uit de relationele algebra de revue gepasseerd. Het moet je opgevallen zijn dat het uitgangspunt hiervan (een relatie is een verzameling die ontstaat door het combineren van andere verzamelingen) heel aardig aansluit bij de eisen die aan een databeheersysteem gesteld worden. In ieder geval viel dat in de jaren '70 van de vorige eeuw wel ene meneer Codd op. Samen met anderen ontwikkelde hij een databeheersysteem dat gebaseerd was op de ideeën uit de relationele algebra, dat zich kenmerkt door de volgende punten

- gegevens worden gegroepeerd in relaties
- relaties die je wil bewaren (zeg maar je 'basisrelaties') worden opgeslagen in tabellen
- doordat de opgeslagen relaties willekeurig gecombineerd kunnen worden kun je nieuwe relaties vormen (en dus informatie uit de gegevens tevoorschijn halen)
- de opgeslagen gegevens voldoen aan de regels zoals die gelden in de relationele algebra

Relaties

In de relationele algebra is een '**relatie**' een verzameling waarin ieder element een geordend n-tal is van elementen uit andere verzamelingen. Iets minder abstract geformuleerd: een relatie bevat elementen waarbij

elk element bestaat uit elementen uit andere relaties
alle elementen bestaan uit dezelfde deel-elementen.

Duidelijk toch? Maar ik zal voor de zekerheid toch maar een voorbeeld geven ;0) .

Sinds Linnaeus heeft in de biologie elk organisme een wetenschappelijke naam die bestaat uit een 'geslachtsnaam' (genusnaam) en een 'soortnaam' (speciesnaam). In meer wiskundige termen bestaat de relatie 'wetenschappelijke namen' dus altijd uit een element uit de verzameling 'geslachtsnamen' en een element uit de verzameling 'soortnamen'.
De wetenschappelijke naam van de wilde eend is bijvoorbeeld *Anas platyrhynchos*. De geslachtsnaam is dan '*Anas' en* de soortnaam is *'platyrhynchos'*. Andere elementen uit dezelfde relatie zouden bijvoorbeeld *Anas clypeata* (de slobeend) of *Homo habilis* kunnen zijn.


Figuur 1: Slobeend (foto: wikipedia)

Relaties hebben nog een aantal andere belangrijke eigenschappen:

- een relatie bevat *nooit* duplicaattupels, met andere woorden, copieën van elementen komen niet voor
- tupels binnen een relatie zijn *ongeordend*.

Een manier om een relatie weer te geven is de tabel. De naam van de tabel is de naam van de

relatie. De kop van de tabel bevat de namen van de onderliggende verzamelingen, terwijl de rijen van de tabel de individuele elementen, de tupels, bevatten.

Ons voorbeeld van de soorten zou er als tabel 2 uit kunnen zien:

genusnaam	soortnaam
Anas	platyrhynchos
Anas	clypeata
Homo	habilis
...	...

Tabel 2: Soortentabel

Domeinen

Zoals al is aangegeven bestaan de tupels in een relatie uit elementen uit andere relaties. De basisverzamelingen waaruit een relatie bestaat worden '**domeinen**' genoemd. In ons voorbeeld zijn zowel `genusnaam` als `soortnaam` dus domeinen. Tupels die onderdeel uitmaken van een domein moeten aan de regels die gelden voor dat domein voldoen. Alle tupels in het domein Postcodes moeten bijvoorbeeld bestaan uit vier cijfers gevolgd door twee letters.

Het maximale aantal tupels in een relatie krijg je door alle elementen uit alle onderliggende domeinen van de relatie aan elkaar te koppelen. Dit wordt het '**carthesisch product**' genoemd. Als bijvoorbeeld het domein genusnamen alleen zou bestaan uit '*Anas*' en '*Homo*' en het domein speciesnamen alleen zou bestaan uit '*habilis*', '*platyrhynchos*' en *'clypeata*' dan zou het carthesisch product van deze twee domeinen er uit zien zoals tabel 3.

genusnaam	speciesnaam
Anas	habilis
Anas	platyrhynchos
Anas	clypeata
Homo	habilis
Homo	platyrhynchos
Homo	clypeata

Tabel 3: Carthesisch product van genusnamen en speciesnamen

De elementen waaruit een domein bestaat zijn per definitie atomair (ondeelbaar). De relatie 'Soortsnamen' waarbij elk tupel bestaat uit een genus- en een speciesnaam is *niet* atomair en kan dus nooit of te nimmer een domein zijn!

Attributen

Binnen een relatie kan een domein vaker dan eens aanwezig zijn. In dat soort gevallen moet er een onderscheid gemaakt worden tussen het domein zelf en de *rollen* van dat domein in die relatie. Een voorbeeld van een domein dat vaak meerdere rollen heeft is 'data' (in de betekenis van tijdstippen). Binnen de relatie 'Personen' zal het domein data bijvoorbeeld de rol van geboortedatum en sterfdatum kunnen hebben. De rol die een domein speelt wordt **attribuuttype** genoemd. Elke rol mag maar één keer in een relatie voorkomen; je kunt bijvoorbeeld maar één keer geboren worden (per leven althans). Elk tupel uit een relatie heeft een waarde, een attribuut, voor alle attribuuttypen waaruit de relatie bestaat.
Let wel, omdat een attribuuttype gekoppeld is aan een domein en een waarde uit een domein per definitie atomair is, is een attribuuttype per definitie ook ondeelbaar. Genus- en speciesnaam zijn dus geldige attribuuttypen maar een samenvoeging van beiden niet!

Als je een relatie als een tabel weergeeft, dan is de naam van een kolom het attribuuttype. Eventueel kun je bij een attribuuttype de naam van het onderliggende domein aangeven door dat tussen haakjes bij het attribuuttype te zetten (zie tabel 4).

Samenvatting:

- Een relatie bestaat uit elementen die tupels genoemd worden.
- Elk tupel bestaat uit attributen die behoren tot een basisverzameling, een domein.
- Attribuutwaarden zijn per definitie ondeelbaar.
- Elk tupel mag maar één keer in een relatie voorkomen.
- De volgordes van attributen binnen een tupel en van tupels binnen een relatie zijn ongedefiniëerd.

In tabel 4 staat een en ander nogmaals grafisch weerggeven.

domein

naam (namen)	**genusnaam (genusnamen)**	**soortsnaam (speciesnamen)**	**vertaling (vertalingen)**
wilde eend	Anas	platyrhynchos	platneuseend
slobeend	Anas	clypeata	schildeend
manager	Homo	habilis	...
...	...	...	...

attribuuttype

tupel

Tabel 4: Grafische weergave van een aantal begrippen uit de relationele algebra.

DatabaseTerminologie

Mensen die met databases werken (nerds) hebben hun eigen jargon. De bewaarde relaties worden **tabellen** genoemd. Elk tupel in zo'n tabel heet een **record** en een attribuut van een record heet **veld**. De opdracht waarmee relaties gecombineerd worden tot een nieuwe relatie is een **query** en het resultaat van zo'n query is een **resultaatset** of een **resultaat**. In de onderstaande tabel staat een en ander nogmaals samengevat, samen met de engelse termen.

omschrijving	relationele algebra	database nederlands	database engels
verzameling opgeslagen gegevens	relatie	tabel	table
verzameling gegevens ontstaan uit het hercombineren van opgeslagen gegevens	relatie	resultaatset (resultaat)	result set
element uit een relatie	element	record	record
onderdeel van een record	attribuut	veld	field
opdracht tot het hercombineren van gegevens	-	query (vraag)	query

Tabel 5: Database – en relationeel jargon vergeleken.

Relationele data

De eis dat gegevens die in een relationele database opgeslagen worden moeten voldoen aan de regels van de relationele algebra, heeft een aantal gevolgen die hieronder duidelijk gemaakt zullen worden.

Veldwaarden

Attributen moet atomair zijn. Oftewel: iedere cel in een tabel mag maar één waarde bevatten en geen verzameling van waarden. Hierbij moet je wel bedenken dat een waarde bijvoorbeeld uit meerdere woorden of letters of cijfers kan bestaan. De naam 'wilde eend' is een voorbeeld hiervan. Als je die waarde op gaat splitsen dan hou je niets over wat, binnen de context van de database althans, een eigen betekenis heeft. De wetenschappelijke naam van hetzelfde beestje, *Anas platyrhynchos*, kan (en moet dus!) wel opgesplitst worden omdat beide woorden wèl een eigen functie hebben.
Het kan voorkomen dat een record voor een attribuut geen waarde heeft, of dat deze waarde onbekend is. De waarde van dat veld is dan **null** oftwel 'niet gedefinieerd'.

Sleutels

Een relatie mag geen duplicaattupels bevatten. Dus mogen er geen identieke records in een tabel

staan. Maar hoe bepaal je dat? Het antwoord is dat je op zoek gaat naar attribuuttypen die op zichzelf, of in combinatie, een tupel uniek maken. Dit worden **kandidaatsleutels** genoemd. Bestaat een kandidaatsleutel uit een groep van attribuuttypen dan mag daar niet een een andere kandidaatsleutel tussen zitten. Met andere woorden, alle attribuuttypen moeten noodzakelijk zijn om de kandidaatsleutel uniek te maken.
In tabel 6 zou bijvoorbeeld de combinatie van geslachtsnaam en speciesnaam een kandidaatsleutel zijn, maar geslachtsnaam in combinatie met speciesnaam en naam niet omdat de eerste twee velden samen al uniek zijn. Het laatste attributtype voegt dus niets zinnigs toe. Wel is naam in tabel 6 een kandidaatsleutel.

Figuur 2: een eland (Foto: wikipedia)

Uit de kandidaatsleutels wordt er één aangewezen als de **primaire sleutel** van de relatie. Die primaire sleutel *garandeert* dus dat elk tupel uniek is. En omdat ook elk record uniek moet zijn **moet** ook elke tabel een primaire sleutel hebben.
De keuze van de primaire sleutel kan grote gevolgen hebben voor de database. We nemen weer de soorten als voorbeeld. Deze tabel is uitgebreid met de eland (zie tabel 6).

naam (namen)	**genusnaam (genusnamen)**	**soortsnaam (speciesnamen)**
wilde eend	Anas	platyrhynchos
slobeend	Anas	clypeata
manager	Homo	habilis
eland	Alces	alces

Tabel 6: De soortentabel met eland

Stel nu dat we als primaire sleutel het attribuuttype 'naam' hadden aangewezen en iemand wil het beest dat in figuur 3 staat toevoegen. Juist. *Taurotragus oryx*, de eland. Dat kan dan niet want de uniciteit van een record wordt bepaald door de primaire sleutel. Achteraf kun je zeggen dat naam niet eens een kandidaatsleutel is, maar je zit dan wel met de brokken. Omdat dikwijls niet met zekerheid gezegd kan worden dat een set aan attribuuttypen *gegarandeerd* uniek zijn, wordt er vaak een extra veld toegevoegd waarin een nummer komt te staan waarvan de beheerder van de database garandeert dat het uniek was, is en zal blijven. Denk bijvoorbeeld aan je studentnummer of je persoonsnummer.

Figuur 3: andere elanden (Foto: wikipedia)

Omdat de primaire sleutel gebruikt wordt om records uniek te maken, mag geen enkel attribuut null zijn. De waarde van een veld dat null is, is niet gedefinieerd, je weet niet wat het is maar je weet ook niet wat het niet is en dus kun je het nergens mee vergelijken en kan het geen onderdeel zijn van een primaire sleutel.

Relaties

Een relatie mag geen herhalende attribuuttypen bevatten. In een tabel mogen dus niet meerdere kolommen met dezelfde naam staan. (Wel mogen meerdere kolommen op hetzelfde domein gedefinieerd zijn.)

Het idee achter een relationele database is dat gegevens die in tabellen opgeslagen worden gecombineerd kunnen worden tot nieuwe informatie. De gegevens die in een tabel opgeslagen worden moeten daarom bij elkaar horen. De datum dat Donald Duck uit het ei gekropen is, heeft misschien wel iets te maken met eenden, maar nog veel meer met één specifieke eend dan met de soort. Daarom hoort die informatie niet thuis in de soortentabel.
Alle ganzen en eenden hebben zwemvliezen tussen de tenen. Daarom hoort die informatie niet thuis in de soortentabel, maar in een tabel die de families beschrijft.
Je kunt in feite zeggen dat elke tabel slechts over één **entiteit** (ding) informatie bevat. Realiseer je wel dat een entiteit niet perse iets tastbaars hoeft te zijn. Ook bijvoorbeeld over een gebeurtenis kun je informatie op slaan. Denk maar aan een huwelijk. Dat is een gebeurtenis die z'n eigen gegevens heeft. Denk maar aan huwelijksdatum.

Uit een tabelnaam moet duidelijk blijken wat voor gegevens er in staan en omdat in een tabel gegevens over meerdere dingen van dezelfde soort bevat, is een tabelnaam in principe meervoud.

Zoals jullie natuurlijk nog weten uit het eerste jaar, zijn de wetenschappelijke namen gebaseerd op onderlinge verwantschappen. Organismen die dezelfde geslachtsnaam hebben, hebben ook een aantal kenmerken gemeenschappelijk die ze onderscheiden van alle andere wezens. Een vereenvoudigde set van kenmerken voor het geslacht *Anas* is: 'eenden die in het zoete water leven'. Als die informatie ook in de database opgenomen moet worden, dan gaat dat vaak op de volgende manier:

naam (namen)	**genusnaam (genusnamen)**	**genus omschrijving**	**soortsnaam (speciesnamen)**
wilde eend	Anas	eenden die in het zoete water leven	platyrhynchos
slobeend	Anas	eenden die in het zoete water leven	clypeata
manager	Homo	null	habilis
eland	Alces	null	alces

Tabel 7: Een foute tabel

Dit klopt niet met de regels. Ook logisch klopt dit niet. In deze tabel worden soorten opgeslagen en de 'genus omschrijving' is indirect misschien wel van toepassing op een soort, maar veel meer op het genus waar de soort bij hoort. In de bovenstaande tabel wordt dus eigenlijk informatie over een genus aan een soort toegevoegd. Het was nou net het idee van de relationele algebra dat je door het combineren van relaties nieuwe relaties kon krijgen. En bij relationele databases moet je door het combineren van de gegevens die je in tabellen op slaat, nieuwe informatie kunnen genereren.

Bijvoorbeeld dat een wilde eend en een slobeend allebei zoetwatereenden zijn! Genus zou dus een eigen tabel, een eigen relatie, moeten zijn met als primaire sleutel de genusnaam!

genusnaam (genusnamen)	**genus omschrijving (omschrijvingen)**
Anas	eenden die in het zoete water leven
Homo	null
Alces	null

Tabel 8: Genera

Als je de verschillende genera in een eigen tabel zet, dan kunnen de gegevens over de genera dus uit de tabel met wetenschappelijke namen. Let wel op dat je genusnaam niet uit deze tabel verwijderd! De volledige wetenschappelijke naam van de wilde eend is nog steeds *Anas platyrhynchos* en dat zou niet meer duidelijk worden uit de database als je ook de genusnaam zou verwijderen. De tabel met soorten komt er dan als volgt uit te zien:

naam (namen)	**genusnaam (genusnamen)**	**soortsnaam (speciesnamen)**
wilde eend	Anas	platyrhynchos
slobeend	Anas	clypeata
manager	Homo	habilis
eland	Alces	alces

Tabel 9: Soorten

Wil je nu weten wat voor eigenschappen de slobeend en de wilde eend gemeenschappelijk hebben, dan zoek je in de soorten-tabel hun genusnaam op en vervolgens in de genera-tabel de omschrijving die bij dat genus hoort! Of je maakt een nieuwe (tijdelijke) relatie aan die er uit ziet als tabel 9. waarbij je de informatie over het genus van een soort combineert met de informatie over de soort op basis van de genusnaam. Het zou bijvoorbeeld geen zin hebben om de beschrijving van het genus *Alces* toe te voegen aan een slobeend.

Een primary key identificeert een record en dus de gegevens binnen dat record. In meer formele bewoordingen zeg je dat de gegevens binnen een entiteit **volledig functioneel afhankelijk moeten zijn van de volledige sleutel**. Gegevens die niet door de sleutel van een record geïdentificeerd worden hebben dus niets in dat record te maken. maar ook gegevens die niets met de hele sleutel te maken hebben horen niet in het bijbehorende record thuis. In de praktijk zijn het vaak aparte entiteiten.
Om bij ons voorbeeld in tabel 7 te blijven: de sleutel van de soorten-tabel is genusnaam in combinatie met soortnaam. Een genus omschrijving heeft niets te maken met een soortnaam; het is dus niet volledig functioneel afhankelijk van **de hele sleutel**.

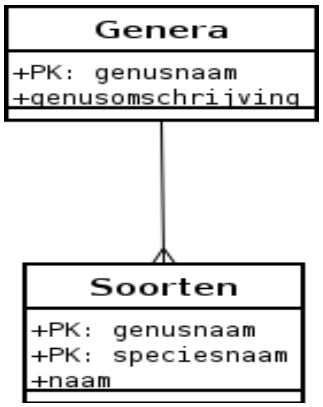
foreign key en referentiële integriteit

Het veld genusnaam uit de soorten-tabel verwijst dus naar de primaire sleutel van de genera-tabel. Daarom wordt het veld een '**verwijzende sleutel**' ('**foreign key**') genoemd.
Een **foreign key** is een veld, of een groep van velden, die verwijzen naar de primary key in een andere tabel. Alle waardes van de primary key waar naar verwezen wordt moeten aanwezig zijn! Dit heet **referentiële integriteit**.
Let op, het omgekeerde is niet waar. Een record mag best bestaan zonder dat daar naar verwezen wordt. Je mag dus zonder meer het genus *Bonellia* aan de tabel genera toevoegen zonder dat je een Bonellia-soort aan de soorten tabel toevoegt.
Het bestaan van de referentiele integriteit heeft nog een belangrijk gevolg. Als je in tabel A een foreign key hebt die verwijst naar tabel B, dan kan tabel B niet een foreign key hebben die naar tabel A verwijst! Want als je in een zo'n situatie eerst een record in A zou zetten, dan zou het gerefereerde record in B nog niet bestaan en vice versa. Méér op méér relaties kunnen dus niet voorkomen!
De oplossing hiervoor is het aanmaken van een **koppeltabel**.
Een koppeltabel tussen twee tabellen bestaat uit een foreign key naar de ene tabel en een foreign key naar de tweede tabel. Eventueel kunnen er nog extra attributen aan toegevoegd worden.

Een foreign key kan altijd maar naar één record verwijzen; de primary key waarnaar verwezen wordt is immers uniek. Er kan wel vanuit verschillende records naar dezelfde primary key verwezen worden. Denk maar aan *Anas*. Zowel de slob- als de wilde eend horen bij dit geslacht. Daarom wordt een relatie met een foreign key ook wel een **één-op-n relatie** genoemd (n kan hierbij ook 0 zijn!).

Entiteiten diagram

In de praktijk is het vaak handig om een overzicht te hebben uit welke tabellen een database bestaat en welke tabellen naar elkaar verwijzen en omdat een plaatje meer zegt dan duizend woorden zijn er verschillende schematechnieken hiervoor ontwikkeld. Een daarvan, **Entity Relation Diagrams** (ERD), zullen wij ook gebruiken.



Figuur 4: Een entiteiten diagram

Een ERD lijkt wel wat op een klasse-diagram. In een ERD wordt een tabel namelijk weergegeven als een vierkantje met daarin de naam van de tabel. Eventueel kun je ook de namen van de attributen in het vierkantje zetten. Tussen de naam van de tabel en de attributen moet dan wel een horizontale streep staan. Als je de attributen weer laat geven, kun links naast de velden die deel uitmaken van een primary key de letters PK zetten. Maken ze deel uit van een foreign key, dan kun je de letters FK gebruiken. De verwijzing van een tabel naar een andere wordt weergegeven door de beide tabellen met een lijn met elkaar te verbinden. Aan de kant van de verwijzende tabel (de tabel dus waarin de foreign key staat die verwijst naar de primary key van de andere tabel) eindigt de lijn in een 'kraaienpootje'. Een open source programma waar in je een entiteitendiagram kunt maken is bijvoorbeel dia.

Een voorbeeld

We gaan nu het voorbeeld van de soorten en de genera verder uitwerken.
Wetenschappelijke namen worden niet zomaar gegeven maar hebben ook een betekenis. Vaak zijn

ze beschrijvend. Om het makkelijker te maken om soorten te herkennen willen we daarom de vertalingen van de wetenschappelijke namen in de soortentabel opnemen. Ook willen we aangeven van welke griekse of latijnse woorden een wetenschappelijke naam is afgeleid.

De wetenschappelijke naam van de slobeend is bijvoorbeeld *Anas clypeata*. De geslachtsnaam is dus *Anas* (latijn voor 'eend'), de soortnaam is *clypeata* en dat slaat op de snavel van de slobeend (een clupeus was een bepaald soort schild dat de romeinen gebruikten). De nederlandse vertaling van de wetenschappelijke naam van een slobeend is dus 'schildeend' ('geschilde eend' zou misschien een meer letterlijke vertaling zijn, maar niet iedereen heeft daar als associatie 'een eend met een schild' mee).
Hoe slaan we deze gegevens op in de database?

De vertaling van de wetenschappelijke naam is makkelijk. Die is specifiek voor een soort en wordt dus in de soortentabel opgeslagen. Meer formeel: 'de vertaling van de wetenschappelijke naam is volledig functioneel afhankelijk van de volledige sleutel'. Als je in een tabel gaat zoeken en je vindt *Anas*, dan betekent het nog niet dat dat het record is waarin je ook schildeend vindt. Hetzelfde geldt voor *clypeata* (er is bijvoorbeeld een paddestoel zonder nederlandse naam die *Pachyella clypeata* heet). Alleen bij de combinatie van *Anas* als genusnaam en *clypetata* als speciesnaam weet je zeker dat je ook het record te pakken hebt waar ook 'schildeend' in staat.
Maar de soortsnaam? Clypeata is zoals gezegd afgeleid van clupeus. Met andere woorden, clypeata verwijst naar clupeus.
Verwijzen? Klinkt als een foreign key en die verwijst van de ene naar een andere tabel! Zou clupeus in een eigen tabel moeten komen? Als je wil weten wat clupeus betekent, dan zou je zoek je clupeus op in een woordenboek en dan staat de betekenis er achter. In feite is een woordenboek een tabel met als primary key het op te zoeken woord en een enkel attribuut, de vertaling. Precies hetzelfde willen we ook in onze database kunnen doen. We maken dus een tabel 'vreemde woorden' aan met als PK het woord waarvan een genusnaam of een soortsnaam afgeleid is en de vertaling als attribuut.

vreemd_woord	**vertaling**
clupeus	romeins schild
anas	latijn voor eend

Tabel 10: vreemde woorden

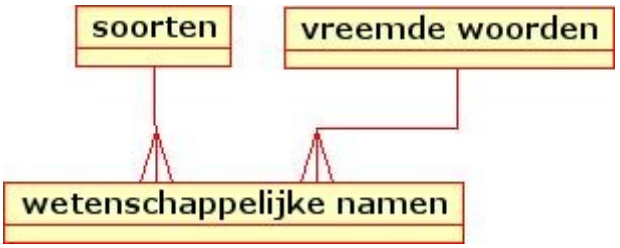
Deze oplossing ziet er prima uit, de vertaling is volledig functioneel afhankelijk van de volledig PK. Geen van de velden valt verder op te splitsen.
Maar een FK verwijst naar een PK en de PK van 'romeins schild' is clupeus en geen clypeata. Op de een of andere manier moet er dus een verwijzing van clypeata naar clupeus opgenomen worden. Clupeus ergens als attribuut opnemen in de tabel met soorten mag niet. De verwijzing naar clupeus is namelijk niet volledig functioneel afhankelijk van de volledige sleutel, maar slechts van een deel van de sleutel (de speciesnaam).
Je maakt dus een nieuwe tabel 'wetenschappelijke_namen' aan, met daarin de wetenschappelijke naam als PK en het woord waar het van afgeleid is als attribuut (dit attribuut is dus de FK die verwijst naar 'vreemde woorden').
Het vullen van deze tabel gaat goed, totdat je bij '*platyrhynchos*' aan komt. Dit woord is afgeleid van twee griekse woorden, namelijk 'platus'(plat) en 'rhynchos' (neus). Je zou dat op kunnen lossen door een tweede attribuut op te kunnen nemen dat ook verwijst naar de vreemde woordentabel, maar wat doe je dan als een wetenschappelijke naam afgeleid is van drie, vier of meer woorden? Je

kunt als databaseontwerper nooit garanderen dat je genoeg velden toegevoegd hebt. Bovendien mogen herhalende groepen binnen een tabel gewoonweg niet zoals je weet.
De oplossing bestaat uit het uitbreiden van de pk.
De combinatie van wetenschappelijke naam en het woord waarnaar verwezen wordt is tenslotte uniek. Dit betekent dat het attribuuttype vreemd woord als pk en tegelijkertijd als fk fungeert!

In dit geval krijgen we dus een tabel zoals in tabel 11 en figuur 5.

naam	vreemd_woord
platyrhynchos	platus
platyrhynchos	rhynchos
clypeata	clupeus

Tabel 11: wetenschappelijke namen



Figuur 5: Diagram van de woordafleidingen

Normaalvormen

Termen die je in de context van relationele databases vaak tegen zult komen zijn 'normaalvorm' en het daarvan afgeleide 'normaliseren'. Een **normaalvorm** is niets anders dan een serie eisen (normen) waar een dataset aan moet voldoen. Er bestaan verschillende normaalvormen. Voor jullie is van belang dat een databaseontwerp moet voldoen aan de **Boyce-Codd normaalvorm (BCNF)**.

De BCNF stelt de volgende eisen:

- Een rij (record) bevat gegevens over één entiteit
- Een record bevat attributen
- Elk attribuut is ondeelbaar
- Alle items van hetzelfde attribuuttype zijn komen uit hetzelfde domein (m.a.w. alle gegevens uit een kolom zijn van dezelfde soort)
- Elk attribuuttype komt maar een keer voor per relatie (m.a.w. elke kolom heeft een unieke naam)
- Herhalende attribuuttypen zijn verboden (m.a.w. attributen mogen niet dezelfde rol binnen een relatie hebben, ook niet als ze een andere naam hebben).
- De volgorde van de attribuuttypen binnen de relatie is niet gedefinieerd
- De volgorde van de records binnen de relatie is niet gedefinieeerd
- Er mogen geen dubbele rijen voorkomen
- Elke entiteit mag maar één primary key hebben
- Elk attribuut moet volledig functioneel afhankelijk zijn van de volledige primary key

10. werkelijkheid en data

In de eerdere hoofdstukken heb je aan de hand van voorbeelden kennis gemaakt met een aantal begrippen uit de wereld van de relationele databases en de eisen die gesteld worden aan de data die daar in opgeslagen worden. In dit deel van het dictaat ga je die kennis toepassen en leer je hoe je een databaseontwerp maakt. Het maken van een databaseontwerp lijkt wel wat op het maken van een klassediagram.

Bij het ontwerpen van een datamodel doorloop je de volgende stappen:

- je verzamelt alle belangrijke attributen
- je stelt de entiteiten vast (let op, kunnen dus ook niet-tastbare dingen zijn!)
- je stelt de foreign keys op die je nodig hebt
- je normaliseert je ontwerp.

Informatiebehoeften

Het verzamelen van attributen gaat vaak het makkelijkst door gebruik te maken van bestaande informatiebehoeften. Een **informatiebehoefte** kan van alles zijn waar gebruik van gemaakt wordt door degene die een database ontworpen wil hebben. Denk aan (web)formulieren, rekeningen en dergelijke. Ook kun je door te praten veel op het spoor komen. Je moet er wel op letten dat verschillende mensen met dezelfde woorden verschillende zaken kunnen bedoelen of met verschillende namen identieke dingen. Ook kan het zijn dat **procesgegevens** (afgeleide gegevens) gebruikt worden in plaats van hun basis. Denk bijvoorbeeld aan leeftijd.

Entiteiten vaststellen

Het vaststellen van entiteiten (tabellen) kan op verschillende manieren. Je kunt uit een lijst met attributen de attributen die bij elkaar horen groeperen en die groep een betekenisvolle naam geven en een pk geven.
Je kunt ook, als je genoeg kennis van het probleemgebied hebt, direct een entiteit aanwijzen en daar attributen aan toe wijzen. Als je bijvoorbeeld een soortendatabase gaat ontwerpen, dan kun je direct al zeggen dat 'soorten' een tabel moet worden.
Een derde manier om entiteiten op te sporen is als je tussen twee tabellen een meer-op-meer relatie vindt.
Altijd als je een meer op meer relatie tussen twee tabellen ziet vervang je die door een koppeltabel. Zoals al eerder is verteld, krijg je dan een koppel-tabel en elke tabel is ook een entiteit. Door over de betekenis van een koppeltabel in de werkelijke wereld na te denken kun je vaak een ook weer attributen op het spoor komen. Dat vraagt wel achtergrondkennis van het doel waar de database voor gebruikt wordt. Een voorbeeld:
Je hebt een meer-op-meer relatie tussen studenten en vakken. De fantasie- en inzichtsloze IT-oplossing zou zijn de koppeltabel 'studenten2vakken' te noemen. Met een kleine inspanning kun je ook bedenken dat 'tentamens' een betere naam is en zomaar ineens komen attributen als 'tentamendatum' en 'cijfer' tevoorschijn.

Controle

De verschillende stappen zullen in de praktijk niet na elkaar plaatsvinden maar veel meer gelijktijdig. Wel is het nuttig als je denkt dat je je ontwerp klaar hebt elke entiteit en elke foreign

key nog eens goed te bekijken. Staat elke entiteit wel in de BCNF? Kloppen de koppelingen tussen de relaties (zijn ze allemaal 1-op-n)? Kloppen de namen? Vaak blijkt bij een dergelijke controle dat er toch nog wat vergeten is. Sla dit dus nooit over!

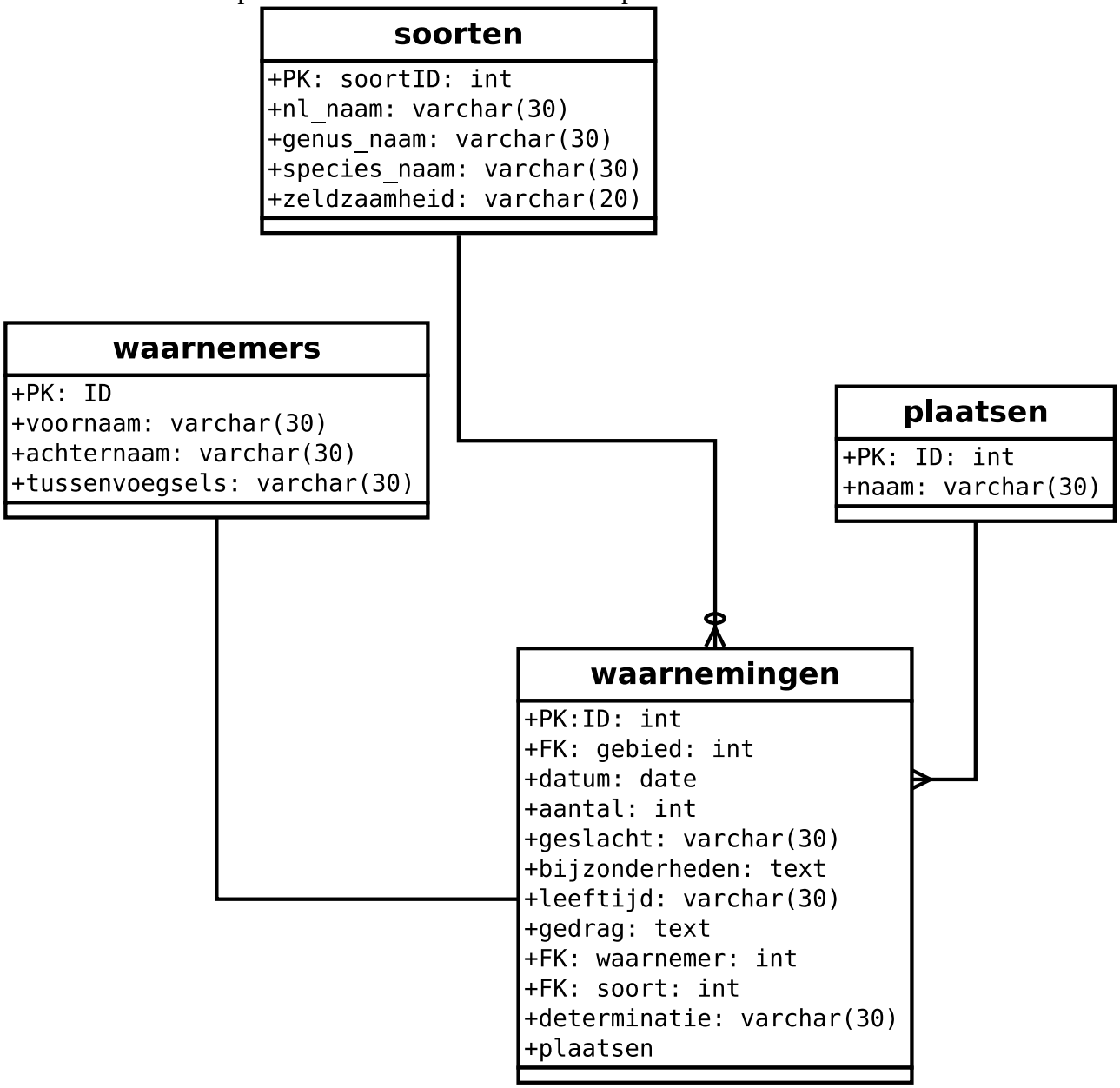
Indices

Iets wat buiten het relationele model valt, maar wel bij het ontwerpen van een database thuishoort zijn indices.
Een **index** heeft als doel zoekacties makkelijker en sneller te laten verlopen. Het zijn in feite zoektabellen die bestaan uit de zoekterm en de records die aan de zoekterm voldoen. Een nadeel van een index is dus dat hij (veel) ruimte inneemt en dat hij, telkens als er gegevens in de tabel die hij indiceert wijzigen, zelf ook bijgewerkt moet worden.
Je vraagt de database manager dus alleen maar een index voor een tabel aan te maken als bijvoorbeeld:

- de gegevens uit de tabel veel opgevraagd worden
- de gegevens uit de tabel relatief weinig gewijzigd worden
- de gegevens die geïndiceerd worden deel uitmaken van een fk
- er binnen de geïndiceerde kolommen veel versschillende waarden voorkomen (indiceren op een boolean heeft bijvoorbeeld geen zin)

Opdrachten:

1. maak in dia het schema van studenten.sql
2. maak een script dat het onderstaande schema implementeert.



Antwoorden

```
4.1 select naam from studenten order by naam;

4.2 select cur_id as 'code', naam from cursussen;

4.3 select * from studenten where naam like '%a%' or naam like '%A%';

4.4 select count(*) from studenten where woonplaats = 'groningen';

4.5 select naam, woonplaats from studenten where woonplaats = 'groningen' or
woonplaats = 'assen';

    naam, woonplaats from studenten where woonplaats in ('groningen', 'assen');

4.6 select woonplaats, count(*) from studenten group by woonplaats order by
woonplaats;

4.7 select * from studenten where geb_datum between '1990-01-01' and '1993-01-
01';

    select * from studenten where geb_datum > '1990-01-01' and geb_datum <
'1993-01-01';

4.8 select * from studenten where not(geb_datum > '1990-01-01' and geb_datum <
'1993-01-01');

4.10 voor toevoegen zie studenten.sql

     delete from studenten where stud_id = 'bakk';

4.12 update studenten set voorletters = 'j' where stud_id = 'jans';

4.14 alter table studenten add column burgerlijke_staat char(1);

     alter table studenten add column burgerlijke_staat char(1) default 'V';

4.15 alter table studenten drop column burgerlijke_staat;


5.1 select cijfer from examens e join cursussen c on e.cur_id = c.cur_id;

5.2 select cijfer from examens e, cursussen c where e.cur_id = c.cur_id;

5.3 select c.naam, cijfer from cursussen c join examens e on c.cur_id = e.cur_id
where naam like 'data%' order by naam, cijfer;

5.4 select s.stud_id, avg(cijfer) from studenten s join examens e on s.stud_id =
e.stud_id group by s.stud_id having count(cijfer) > 1;

5.5 select s.naam, woonplaats from studenten s join examens e on s.stud_id =
e.stud_id group by s.stud_id having count(cijfer) > 1;

5.6 select s.stud_id, s.naam, max(ex_datum) from studenten s join examens e on
s.stud_id = e.stud_id join cursussen c on e.cur_id = c.cur_id where not c.naam =
'biologie 1' group by s.stud_id having avg(cijfer) > 5;

5.7 select stud_id, ex_datum, cur_id from examens where cijfer is null;

5.8 twee queries nodig: eerst de lengte van het woonplaatsveld aanpassen
vervolgens kun je de student toevoegen.

5.9 select s.naam as 'student', m.naam as 'mentor' from studenten s join
studenten m on s.mentor = m.stud_id;


6.1 select * from studenten where geb_datum = (select min(geb_datum) from
```

```
studenten);
```

```
6.2 select s.naam from studenten s join examens e on s.stud_id = e.stud_id where
cijfer = (select max(cijfer) from examens);
```

```
6.3 insert into examens(stud_id, cur_id, ex_datum, cijfer) values('parn',
(select cur_id from cursussen where naam = 'databases2'), (select curdate()),
10);
```

```
6.4 select voorletters, s.naam, c.naam, cijfer from studenten s join examens e
on s.stud_id = e.stud_id join cursussen c on c.cur_id = e.cur_id where cijfer =
(select max(cijfer) from examens);
```

```
7.1 select cijfer from examens e join cursussen c on c.cur_id = e.cur_id where
naam = 'databases 1';
```

```
7.2 select cijfer from examens e, cursussen c where c.cur_id = e.cur_id and naam
= 'databases 1';
```

```
7.3 select naam, cijfer from examens e join cursussen c on c.cur_id = e.cur_id
where naam like 'database%' group by naam, cijfer;
```

```
7.4 select stud_id, avg(cijfer) from examens group by stud_id having
count(cijfer) > 1;
```

```
7.5 select s.naam, woonplaats from studenten s join examens e on s.stud_id =
e.stud_id group by s.stud_id having count(cijfer) > 1;
```

```
7.6 select s.stud_id, s.naam, max(ex_datum) from studenten s join examens e on
s.stud_id = e.stud_id join cursussen c on e.cur_id = c.cur_id where c.naam !=
'biologie 1' group by s.stud_id having avg(cijfer) > 5;
```

```
7.7 select stud_id, ex_datum, cur_id from examens where cijfer is null;
```

```
7.8 alter table studenten modify column woonplaats varchar(255) not null;
```

```
    insert into studenten(stud_id, voorletters, naam, tussenvoegsel, woonplaats,
geb_datum, mentor) values('clin', 'd', 'harry', null, 'El Pueblo de Nuestra
Señora la Reina de los Ángeles del Río de Porciúncula', '1967-02-05', null);
```

```
7.9 select s.naam as 'student', m.naam as 'mentor' from studenten s join
studenten m on s.mentor = m.stud_id;
```

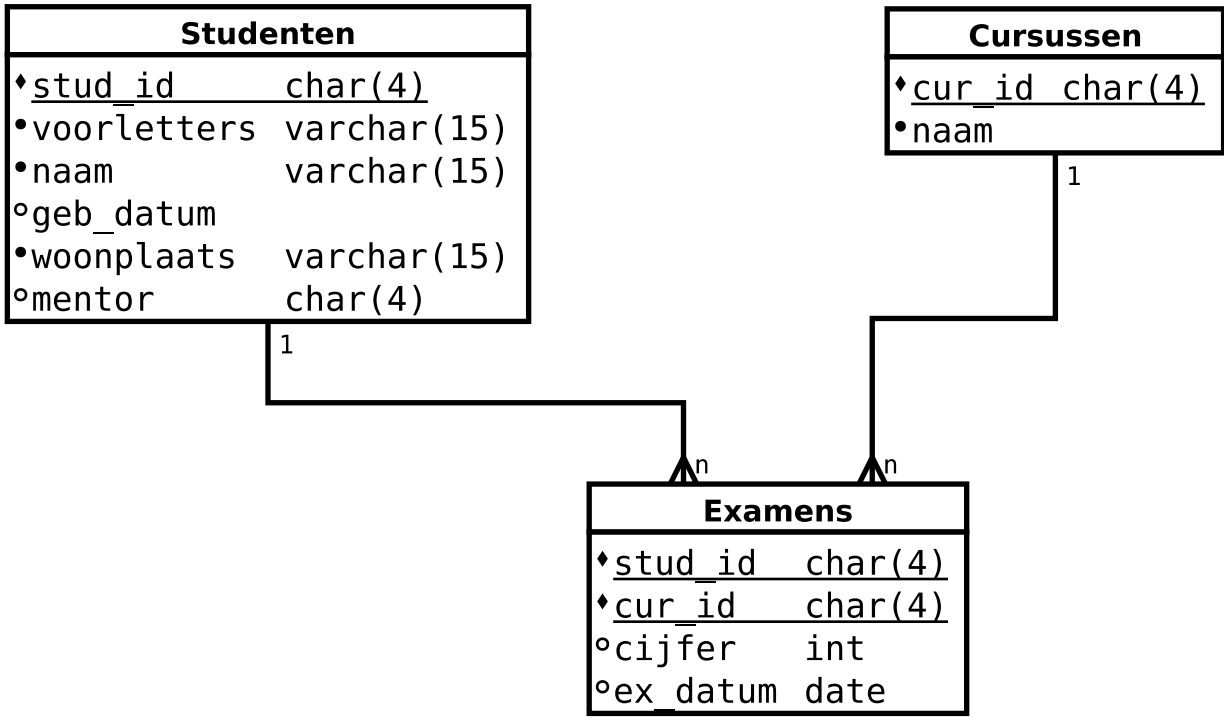
```
8.1 select * from studenten where geb_datum = (select min(geb_datum) from
studenten);
```

```
8.2 select naam from studenten s join examens e on s.stud_id = e.stud_id where
cijfer = (select max(cijfer) from examens);
```

```
8.3 insert into examens(stud_id, cur_id, ex_datum, cijfer) values('parn',
(select cur_id from cursussen where naam = 'databases2'), '05-11-25', 10);
```

```
8.4  select s.voorletters, s.naam, c.naam, e.cijfer from studenten s join
examens e on s.stud_id = e.stud_id join cursussen c on e.cur_id = c.cur_id where
cijfer = (select max(cijfer) from cursussen);
```

```
8.5 insert into examens(stud_id, cur_id, ex_datum, cijfer) values((select
stud_id from studenten where naam = 'bakker'), (select cur_id from cursussen
where naam = 'databases2'), '19-10-09', 7);
```


Studenten
♦stud_id char(4)
•voorletters varchar(15)
•naam varchar(15)
○geb_datum
•woonplaats varchar(15)
○mentor char(4)
Cursussen
♦cur_id char(4)
•naam
1
1
n
n
Examens
♦stud_id char(4)
♦cur_id char(4)
○cijfer int
○ex_datum date

9.1