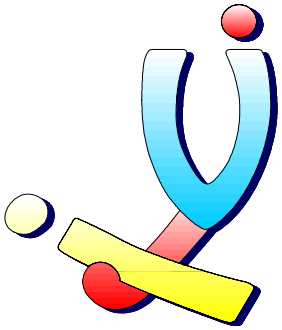
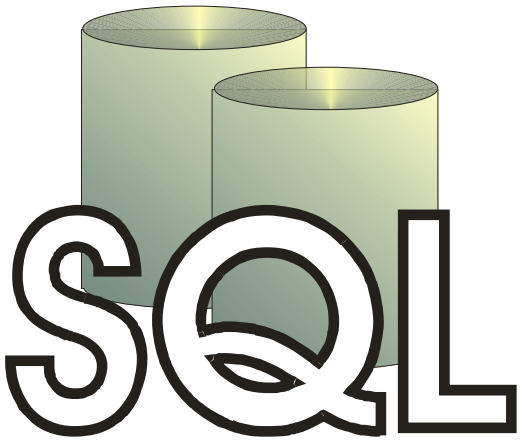


Computerclub Volwassenen,
Jeugd en Informatica vzw
www.vji.be

Cursus bij demo databases en SQL



- Introductie in terminologie databases
- Werken met universele query taal SQL
- Vergelijking verschillende databases, tools en Delphi componenten

Stefan Cruysberghs
www.scip.be
Februari 2003

Inhoudsopgave

Inleiding

Een database is tegenwoordig de belangrijkste toepassing van een computer. Alle informatie gaande van persoonsinformatie tot bestellingen, facturen, boekhouding, archief, medische dossiers, voorraden, … worden in een database bewaard. Zowel thuisgebruikers als bedrijven, de overheid, bibliotheken, … hebben we een database in gebruik.

Tijdens deze demonstraties zullen we allereerst de theoretische aspecten van een database bekijken; hoe worden gegevens gestructureerd, welke relaties bestaan, welke verschillende soorten database systemen bestaan er, … Termen zoals velden, records, primaire en vreemde sleutels, referentiële integriteit, indexen, triggers, transacties, client-server, multi-tier, ... zullen allen besproken worden.

Hierna verdiepen we ons in SQL, een universele database taal waarmee we een database structuur kunnen aanmaken of wijzigen en data kunnen bekijken of manipuleren. We testen onze select, update, delete, insert, alter table, … queries op een echte database.

Uiteindelijk gaan we aan de slag met Delphi en zullen we een Interbase/Firebird of MySQL database op een server proberen aan te spreken.

Sommige delen uit deze cursus zijn gekopieerd van artikels en handleidingen op het internet. Daarom dank aan Marc Andries en Henk Jan Nootenboom.

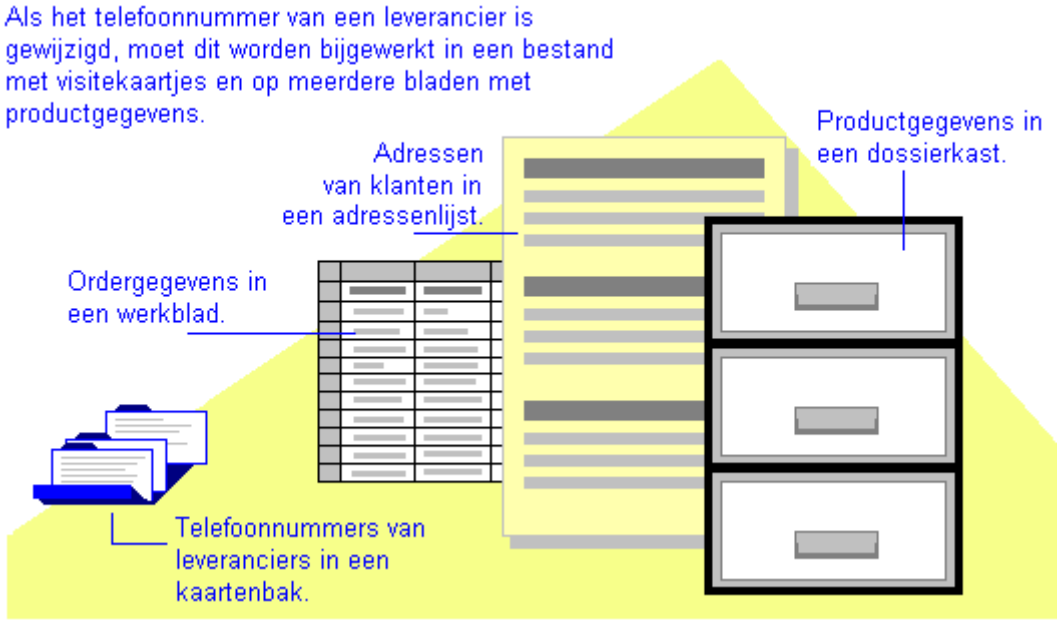
Stefan Cruysberghs
www.scip.be

Opbouw van een database

Wat is een database ?

Een database is een verzameling van gegevens over een bepaald onderwerp of met een bepaald doel, zoals het bijhouden van klantorders. Als uw database niet in een computer is opgeslagen of als de database slechts gedeeltelijk is opgeslagen, moet u de gegevens in verschillende bronnen bijhouden en alles zelf coördineren en organiseren.

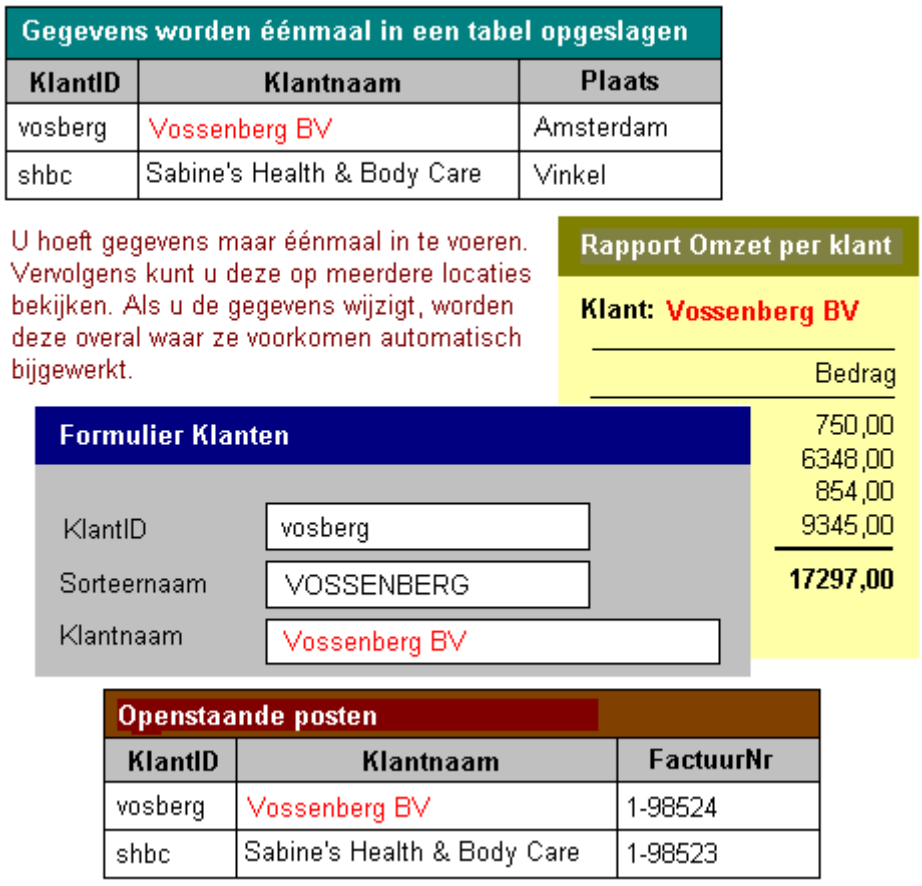
Wanneer het telefoonnummer van een leverancier verandert, moet dit worden bijgewerkt in een kaartenbak en op een aantal productlijsten. Adressen van klanten moeten worden gewijzigd in een verzendlijst, productinformatie in een dossierkast, etc. etc.



Gerelateerde gegevens van klanten, producten of orders kunnen zich op vele plaatsen binnen een bedrijf bevinden en de administratie kan al snel onoverzichtelijk, zo niet chaotisch worden, als er niet continu aan gewerkt wordt om het geheel onder controle te houden. Hoge kosten en een stroef lopend geheel zijn dan het resultaat.

Zoals nu...

In een goed 'genormaliseerde' database worden gegevens slechts één keer opgeslagen. De gegevens zijn onderling gerelateerd (gekoppeld): aan een klant zijn bijvoorbeeld meerdere facturen gekoppeld. In elke factuur zijn producten/diensten opgenomen die weer specifieke gegevens (voor die klant) bevatten zoals de BTW, het aantal, de prijs etc. Aan de facturen zijn gegevens gekoppeld die betrekking hebben op de betaaldatum, het betaald bedrag, hoe de betaling heeft plaatsgevonden of wanneer er voor die betreffende factuur een herinnering of een aanmaning is verzonden. Een klantnaam kan bijvoorbeeld worden weergegeven in een overzicht van openstaande posten of in omzet rapporten.



Databases bestaan er tegenwoordig in alle maten en soorten, gaande van kleine lokale databases voor persoonlijk gebruik tot supergrote server databases waarop wel duizenden mensen tegelijkertijd data kunnen raadplegen. Een database is eigenlijk de software die de data bewaard en mechanismen aanbiedt om deze te raadplegen, te wijzigen, … Een grafische user interface om de database aan te spreken, hoort eigenlijk niet echt bij de database zelf. De meeste database fabrikanten leveren echter wel extra hulpmiddelen mee. Met bijna alle huidige object georiënteerde programmeertalen zoals Delphi, VB, Java, C++, … kan je een applicatie voor een database ontwerpen.

Hoe is een database gestructureerd ?

Veld (field) / Kolom (column)

In één **veld/kolom** (data-element) staan gegevens van dezelfde soort, bijvoorbeeld de velden naam, adres, leeftijd.

Datatype

Elke veld is van een bepaald **datatype**. Bijvoorbeeld **integer, string, date, float, …**
Bijvoorbeeld een naam is een string, de leeftijd is een integer, de geboortedatum is een date.

Record / Rij (row)

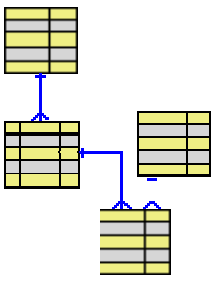
Een **record** of **rij** zijn gegevens die bij elkaar horen, bijvoorbeeld de gegevens van één persoon

Tabel (table)

Een **tabel** is een matrix met gegevens. Een tabel in een database bestaat uit meerdere records.

Database

Een **database** is een verzameling tabellen, met onderling gerelateerde gegevens.



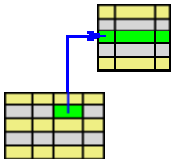
Andere onderdelen van een database

Primaire sleutel (PK, primary key)

Een **sleutel** of **key** bestaat uit één of meerdere velden/kolommen. Met de sleutel kan je aangeven op welke manier gegevens uniek zijn. Een key waarde mag niet dubbel voorkomen in een tabel. Met een key vind je dus hooguit één rij. De key is als een sleutel waarmee je toegang krijgt tot de juiste rij.

Vreemde sleutel (FK, foreign key)

Een **vreemde sleutel** of **foreign key** is de verbindende schakel tussen twee tabellen.
Met een waarde uit een rij van de ene tabel kun je in een andere tabel het juiste record met gerelateerde gegevens opzoeken. De ene tabel geeft als het ware de sleutel voor de andere, 'vreemde' tabel.



Referentiële integriteit (RI, referential integrity)

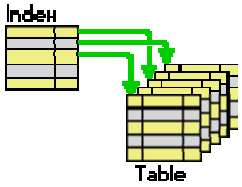
Referentiële Integriteit zorgt ervoor dat een foreign key waarde altijd naar een bestaande row verwijst. Een "dode" verwijzing zoals in het plaatje kan dan niet meer voorkomen.

Er zijn een paar mogelijke maatregelen die **referentiële integriteit** afdwingen.

- Bij het maken of wijzigen van een rij controleert het systeem of de foreign keys wel geldige waarden hebben.
- Daarnaast moet de database-ontwerper een keuze maken voor een delete:
    - Je mag een rij in de one tabel pas weggooien als er geen gerelateerde many rows meer zijn. Je zal als gebruiker dan ook een foutmelding krijgen dat de data nog in gebruik is. Bijvoorbeeld, je wil een klant verwijderen maar er bestaan nog orders voor deze klant.
    - Bij het verwijderen van een row in de one tabel gooit de database automatisch alle gerelateerde gegevens in de many tabel weg. Dit heet een **cascaded delete**. Bijvoorbeeld, je wil een order verwijderen en automatisch worden ook alle orderlijnen verwijderd.

Index

Een index in een database lijkt op een index achter in een boek. Met een zoekwoord kun je de juiste bladzijde nummers vinden van rijen, binnen een tabel. Een **index** bestaat uit één of meerdere velden/kolommen. Een index hoeft, in tegenstelling met een sleutel, niet uniek te zijn. Indexen worden vooral gebruikt om het opzoeken te versnellen.



Relaties

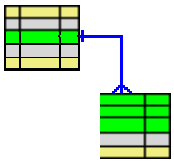
Tussen verschillende tabellen bestaan er relaties. Deze relaties ontstaan meestal door het aanmaken van vreemde sleutels. In het soort relaties kunnen we 4 types onderscheiden.

Een op veel (one to many)

De meeste relaties tussen tabellen zijn one-to-many.

Voorbeeld:

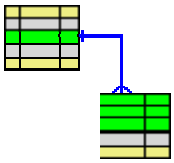
- In één gebied kunnen vele lezers wonen.
- Één lezer kan vele abonnementen hebben.
- Één krant kan vele abonnementen hebben.



Meer op een (many to one)

Een Many to One relatie is hetzelfde als one-to-many, maar dan van een ander gezichtspunt.

- Vele lezers wonen in één gebied.
- Vele abonnementen kunnen van één en dezelfde lezer zijn.
- Vele abonnementen zijn op één en dezelfde krant.

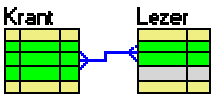


Een op een (one to one)

Een één op één relatie is uitzonderlijk in databases. Het kàn voorkomen, maar vaak is het een teken dat het database-ontwerp nog voor verbetering vatbaar is

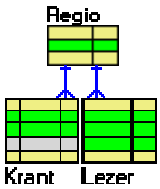
Veel op veel (many to many)

Een many to many relatie komt veel voor. Een krant heeft bijvoorbeeld vele lezers en een lezer leest vele kranten. Een many-to-many relatie is echter onduidelijk. Many-to-many relaties zijn vaak een teken dat nog nadere analyse vereist is.

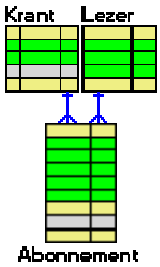


Meestal kan de relatie een stuk duidelijker, door er een koppel tabel tussen te zetten, aan de 'boven' of aan de 'onder' kant.

Voor een huis aan huis blad zal het verspreidingsgebied, de regio, de link zijn naar de lezer, aan de bovenkant.



Voor een betaalde krant is de link het abonnement, aan de onderkant.



Normalisatie

Iedereen die al wat langer met een database werkt, kent het probleem. Vroeg of laat kom je er achter dat je tijdens het ontwerpen iets niet helemaal goed hebt gedaan, waardoor de boel groter is geworden dan nodig was. Of nog erger: er zitten nu grote fouten in je database die je er niet snel meer uit kunt halen. Het is dan ook nodig om goed na te denken over het database ontwerp wat je maakt. Zorg voor een goede structuur zodat je het later gemakkelijk kan uitbreiden.

Een manier om de je database beter te maken is: **database normalization**, hierbij gaat het vooral om het vermijden van dubbele velden in je database ontwerp. Er bestaan een reeks regels, de **regels van Codd** genaamd, die je dient toe te passen. De regels zijn echter redelijk theoretisch en vaak kan je met een beetje gezond verstand deze al automatisch toepassen.

Regels van CODD

a. Inventariseer alle **elementaire gegevens**. b. Verwijder alle **procesgegevens**. c. Kies een **primaire sleutel.** d. Zonder **herhalende deelverzamelingen** af en voeg er de primaire sleutel van de originele gegeensreeks bij. Deze items worden in de lijst met elementaire gegevens reeds met een sterretje aangeduid. Deze moeten nu een afzonderlijke reeks vormen omdat uiteraard niet vast te leggen is hoe dikwijls deze gegevens op hetzelfde document herhaald zullen worden. In termen van tabelstructuur uitgedrukt: je weet niet hoeveel velden je moet aanmaken. De punten c en d moeten herhaald worden tot er geen herhalende groepen meer zijn. De gegevens staan dan in de **eerste normaalvorm**.
• Verwijder de items die slechts afhankelijk zijn van een deel van de sleutel en plaats ze samen met het deel van de sleutel waarvan ze afhankelijk zijn in een nieuwe gegevensreeks. Pas daarop de regels toe vanaf de eerste normaalvorm. De gegevens staan dan in de **tweede normaalvorm**.
• Verwijder de attributen die afhankelijk zijn van andere niet-sleutel attributen en plaats ze samen met het deel van het attribuut waarvan ze afhankelijk zijn in een nieuwe gegevenstreeks. Pas daarop de regels toe vanaf de eerste normaalvorm. De gegevens staan dan in de **derde normaalvorm**.
De gegevensreeksen krijgen nu een naam en overal waar een herhalende groep of een functioneel afhankelijk gegeven is afgeplitst worden relaties gelegd.

Een voorbeeld

Om het allemaal wat duidelijk en tastbaarder te maken gaan we hier uit van de volgende situatie, je hebt een software bedrijf, hierin werken verschillende mensen, met verschillende lonen aan verschillende projecten. Om dit allemaal bij te kunnen houden, heb je een leuke database in elkaar gezet, deze is alleen nog niet helemaal perfect.

Project nr.	**Project naam**	**Werknemer nr.**	**Werknemer naam**	**Betalings catergorie**	**Uurtarief**
1023	Facturatie module	11	Tom Thomas	A	60
1023	Facturatie module	12	Jan Jansen	B	50
1023	Facturatie module	16	Piet Pieters	C	40
1056	Website ontwikkeling	11	Tom Thomas	A	60
1056	Website ontwikkeling	17	Dirk Diercksens	B	50

Op zich een goede tabel, zeker als je niet meer dan drie werknemers, twee opdrachten hebt, en na die opdrachten ophoudt te bestaan.

Als je goed naar de tabel kijkt zie je dat er niet een veld, de zogenaamde primary key vormt van deze tabel. Met andere woorden, geen enkel veld is uniek voor zijn/haar rij. Wat nog meer opvalt is dat ‘Facturatie module’, drie keer voorkomt in deze tabel, mocht je ooit besluiten de naam van het project te veranderen, dan moet je dat in elke record waarin deze naam voorkomt gaan doen, niet echt handig dus. Laten we die kolom dus uit de tabel halen. We krijgen dan het volgende.

werknemer_project tabel:

Project nr.	**Werknemer nr.**
1023	11
1023	12
1023	16
1056	11
1056	17

werknemer tabel:

Werknemer nr.	**Werknemer naam**	**Betalings catergorie**	**Uurtarief**
11	Tom Thomas	A	60
12	Jan Jansen	B	50
16	Piet Pieters	C	40
17	Dirk Diercksens	B	50

project tabel:

Project nr.	**Project naam**
1023	Facturatie module
1056	Website ontwikkeling

OK, een groot deel van de dubbele velden, zogenaamd *anomalies* zijn nu verwijderd, maar nog niet allemaal als we goed kijken naar de werknemer tabel, zien we er nog een zitten.

werknemer tabel:

Werknemer nr.	**Werknemer naam**	**Betalings catergorie**	**Uurtarief**
11	Tom Thomas	A	60
12	Jan Jansen	B	50
16	Piet Pieters	C	40
17	Dirk Diercksens	B	50

Zoals je ziet zijn de laatste twee kolommen, duidelijk aan elkaar gerelateerd. Bij betalings categorie A hoor 60 en bij B hoort 50, enz. Laten we deze twee dingen dan ook nog maar afsplitsen.

werknemer tabel:

Werknemer nr.	**Werknemer naam**	**Betalings categorie**
11	Tom Thomas	A
12	Jan Jansen	B
16	Piet Pieters	C
17	Dirk Diercksens	B

salaris tabel:

Betalings catergorie	**Salaris**
A	60
B	50
C	40

Zoals je ziet zijn nu alle verschillende soorten gegevens, netjes in een aparte tabel op geslagen, hierdoor is het makkelijk deze gegevens correct te houden. Ons oorspronkelijke tabel is nu dus opgesplits in 4 tabellen : projecten, salaris, werknemer en werknemer-projecten. En kunnen er ook niet zo makkelijk fouten worden gemaakt, zolang alles in de salaris tabel goed staat, kan je nooit de fout maken om betalings categorie met iets anders te verbinden dan 50.

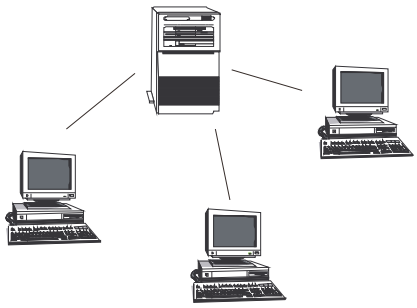
Het proces wat we net hebben doorgemaakt, is dus normalisatie, we zijn nu aanbeland in de de zogenaamde **eerste normaalvorm** (**first normal form)**. Er bestaan ook nog de zogenaamd **tweede normaal vorm (second normal form)** en de **derde normaalvorm (third normal form**), deze zijn zeker voor de beginnende database ontwerper nog niet echt van belang. Die laten we dan ook maar even voor wat het is.

Welke soorten databases zijn er ?

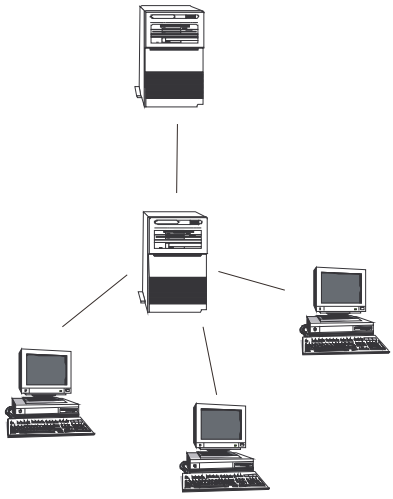
Standalone (file based), client-server, multi-tier

Standalone : deze kleinere databases staan lokaal op een PC. De database zal rechtstreeks als bestand aangesproken worden.

Client-server : dit principe, wat in de praktijk het meeste voorkomt, bestaat er uit dat de database op een server staat en het programma dat de database gebruikt op een client, meestal een gewone desktop PC. Op de client dient men een klein stukje software te installeren zodat er communicatie met de database mogelijk is. Op de server draait er ook een stukje software, meestal in de vorm van een service.



Multi-tier : multi-tier of ook wel 3-tier genaamd is een uitbreiding van het client-server principe. De database staat op een server, er is nog steeds een client, maar het grootste deel van de verwerking gebeurt niet meer op de client maar wel op een applicatie server.



Single user, multi-user

Het aantal gebruikers van een database hangt nauw samen met het type van database. Alle client-server en multi-tier databases kunnen werken met meerdere gebruikers die vanaf meerdere clients kunnen inloggen op de database. Wanneer men met meerdere gebruikers op eenzelfde database kan werken, zijn er ook altijd beveiligingen ingebouwd die per gebruiker kunnen gewijzigd worden.

Een standalone database is meestal enkel geschikt voor 1 gebruiker.

Transacties

Het principe van transacties wordt gebruikt wanneer meerdere database acties (update, insert, delete) moeten doorgevoerd worden. Met een transactie kan je controleren of alle acties gelukt zijn ofwel kan je alles ongedaan maken.

Een transactie start je meestal met een bepaald commando zoals **StartTransaction**. Na het uitvoeren van update, delete en insert statements kan je alle wijzigen doorvoeren met het **Commit** commando. Alles ongedaan maken kan met de **Rollback**. Deze rollback zal meestal ook gebruikt worden als er ergens een fout is opgetreden.

Een Delphi voorbeeldje met de BDE :

```
Database1.StartTransaction;
try
  QueryInsertMaster.ExecSQL;
  QueryInsertDetail.ExecSQL;
  Database1.Commit;
except
  Database1.Rollback;
end;
```

Een Delphi voorbeeldje met de dbExpress :

```
SQLConnection1.StartTransaction(TD1);
try
  SQLQueryInsertMaster.ExecSQL;
  SQLQueryInsertDetail.ExecSQL;
  SQLConnection1.Commit(TD1);
except
  SQLConnection1.Rollback(TD1);
end;
```

Programmatie op een database

De meeste grote databases hebben een ingebouwde programmeertaal die de combinatie is tussen SQL en een gewone procedurele taal met elementen zoals if, for, repeat, … Het voordeel van routines op de database is dat deze door de server verwerkt worden en hierdoor is er ook geen netwerkverkeer. Sommige databases zoals Oracle, MS SQL Server, DB2, … hebben een zeer krachtige programmeertaal die je kan gebruiken in stored procedures, triggers, packages, … Ook kan je eigen functies schrijven die je in gewone SQL opdrachten kan gebruiken.

Stored procedure

Een **stored procedure** is een procedure die geïmplementeerd is op de database. Deze kan vanuit een trigger ofwel vanuit een programma gestart worden. Een stored procedure kan ook een functie zijn en deze kan dan eventueel in een SQL opdracht gebruikt worden.

Een PLSQL voorbeeld uit Oracle

```
procedure ChangeAddresses(StrCountryPost in string)
is
  StrPrintCity varchar2(50);
  StrPrintAddress varchar2(70);
  StrPrintCountry varchar2(50);
  IntPrintcountrypostyn pls_integer;
  IntPrintpostalcodeyn pls_integer;
  IntPrintcityyn pls_integer;
  IntPrintareamapyn pls_integer;
  IntPrintcountryyn pls_integer;
  IntPrintstreetyn pls_integer;
  IntPrinthousenumberyn pls_integer;
  IntPrintpostboxyn pls_integer;

  cursor Cur_Addrss is
    SELECT a.AddressNo, a.Countryname
    FROM GMADDRSS a
    WHERE a.Countrypost = StrCountryPost;

begin
  SELECT
    z.Printcountrypostyn, z.Printpostalcodeyn, z.Printcityyn, z.Printareamapyn,
    z.Printcountryyn, z.Printstreetyn, z.Printhousenumberyn, z.Printpostboxyn
    INTO
    IntPrintcountrypostyn, IntPrintpostalcodeyn, IntPrintcityyn, IntPrintareamapyn,
    IntPrintcountryyn, IntPrintstreetyn, IntPrinthousenumberyn, IntPrintpostboxyn
  FROM FMCNTRY z
  WHERE z.Countrypost = StrCountryPost;

  UPDATE GMADDRSS a
  SET
    a.Printcountrypostyn = IntPrintcountrypostyn,
    a.Printpostalcodeyn = IntPrintpostalcodeyn,
    a.Printcityyn = IntPrintcityyn,
    a.Printareamapyn = IntPrintareamapyn,
    a.Printcountryyn = IntPrintcountryyn,
    a.Printstreetyn = IntPrintstreetyn,
    a.Printhousenumberyn = IntPrinthousenumberyn,
    a.Printpostboxyn = IntPrintpostboxyn
  WHERE a.Countrypost = StrCountryPost;
```

```
  for Rec_Addrss in Cur_Addrss loop
    StrPrintCity := GCGETPACK.GCGetPrintCity(Rec_Addrss.AddressNo);
    StrPrintAddress := GCGETPACK.GCGetPrintAddress(Rec_Addrss.AddressNo);
    if IntPrintcountryyn = 1 then
      StrPrintCountry := Upper(Rec_Addrss.Countryname);
    else
      StrPrintCountry := '';
    end if;

    UPDATE GMADDRSS a
    SET a.Printcity = StrPrintCity,
      a.Printaddress = StrPrintAddress,
      a.Printcountry = StrPrintCountry
    WHERE a.Addressno = Rec_Addrss.AddressNo
      AND a.Countrypost = StrCountryPost;
  end loop;
end;
```

Trigger

Een **trigger** is ook een stukje programmatie wat vergelijkbaar is met een gebeurtenis (event) in een object georiënteerde taal. Bij een bepaalde actie (before insert, after update, before delete, …) op een tabel zal deze routine gestart worden.

In een trigger zijn er meestal 2 speciale pointers naar het record met de oude waardes en een record met de nieuwe waardes.

In Oracle kan je deze aanspreken met :OLD en :NEW. In Interbase/Firebird is het ongeveer hetzelfde; nl OLD en NEW

Een PLSQL voorbeeld uit Oracle

```
create or replace trigger "ADMIN".B_I_AMDELSUP
  before insert on AMDELSUP
  for each row
begin
  if (not DBMS_REPUTIL.FROM_REMOTE)
    and (not DBMS_SNAPSHOT.I_AM_A_REFRESH) then
    :NEW.DATEINPUT := TO_DATE(SYSDATE);
  end if;
end;
```

Een voorbeeld uit Interbase/Firebird

```
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE AFTER UPDATE
AS
BEGIN
  IF (OLD.SALARY <> NEW.SALARY) THEN
    INSERT INTO SALARY_HISTORY
      (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY,
       PERCENT_CHANGE)
      VALUES (OLD.EMP_NO, 'NOW', USER, OLD.SALARY,
       (NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
END
```

Functies

SQL kent standaard een reeks functies voor wiskundige berekeningen of datum of string bewerkingen. In de meeste grote database kan je echter ook eigen functies maken. In Interbase/FireBird noemt men dit UDF (User Defined Function). In Oracle kan je functies en procedures ook verzamelen in een Package. Deze zelf gemaakt functies kan je dan gebruiken in SQL statements.

Een PLSQL voorbeeld uit Oracle

```
function GetWagegroupEmployee(IntCompany in pls_integer,
  IntEmployee in pls_integer) return pls_integer
is
  IntWagegroup pls_integer;
begin
  SELECT a.Wagegroup
    INTO IntWagegroup
  FROM PMEMPLOY a
  WHERE a.Company = IntCompany
    AND a.Employee = IntEmployee;

  return(IntWagegroup);

  exception
    when no_data_found then
      return(-1);
end;
```

Views

Vaak ondersteunen database ook het principe van views. Dit zijn eigenlijk SELECT queries op tabellen. Het verschil met een gewone query is dat deze bewaard wordt op de database en bovendien ook al gecompileerd is. Een view zal dan ook sneller werken dan een query die je vanuit een programma uitvoert.

Een PLSQL voorbeeld uit Oracle

```
CREATE OR REPLACE VIEW AVORDSUP AS
  SELECT
    S.COMPANY,
    S.ORDERSUPPLIERTYPE,
    S.ORDERSUPPLIER,
    S.CODEARTICLERAWMATERIAL,
    SUM(S.TOTALQUANTITYORDERED)   AS TOTALQUANTITYORDERED,
    SUM(S.TOTALQUANTITYDELIVERED) AS TOTALQUANTITYDELIVERED,
    SUM(S.TOTALAMOUNTORDEREDCC) AS TOTALAMOUNTORDEREDCC,
    SUM(S.TOTALAMOUNTDELIVEREDCC) AS TOTALAMOUNTDELIVEREDCC,
    SUM(S.TOTALAMOUNTORDEREDFC) AS TOTALAMOUNTORDEREDFC,
    SUM(S.TOTALAMOUNTDELIVEREDFC) AS TOTALAMOUNTDELIVEREDFC,
    SUM(DECODE(S.ORDERSTATUS,3,0,4,0,6,0,(S.TOTALQUANTITYORDERED-
      S.TOTALQUANTITYDELIVERED))) AS TOTALQUANTITYOPEN,
  FROM AMORDSPD S
  GROUP BY S.COMPANY,S.ORDERSUPPLIERTYPE,S.ORDERSUPPLIER,S.CODEARTICLERAWMATERIAL;
```

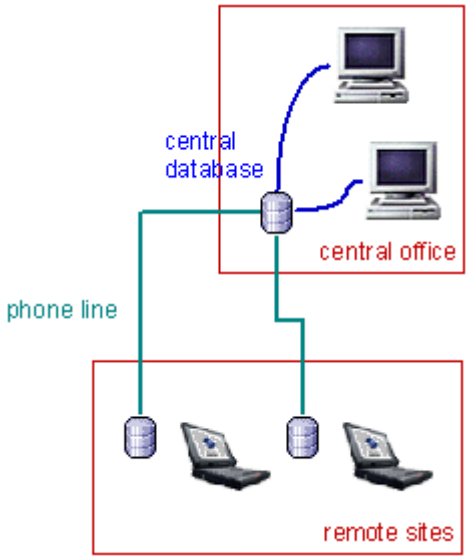
Nog enkele termen

RDBMS

Deze term **RDBMS** betekent **Relational Database Managment System** en eigenlijk is het datgene wat de meeste mensen simpelweg een database noemen.

Replicatie

Replicatie is de techniek om 2 of meerdere databases met dezelfde structuur te synchroniseren op bepaalde momenten. Databasereplicatie wordt gebruikt in situaties waarin niet iedereen direct toegang heeft tot de centrale plek waar de programmadata is opgeslagen Bijvoorbeeld, een bedrijf heeft een centrale database en enkele vertegenwoordigers hebben een portable die niet altijd aangesloten kan worden op het bedrijfsnetwerk. Op deze portables zal dan ook een eigen database geïnstalleerd worden en op bepaalde momenten kunnen de gegevens tussen de database op de portable en deze op de central server gesynchroniseerd worden.



ERD

Een **ERD (Entity Relationship Diagram)** beschrijft de datastructuur van het relevante deel van de werkelijkheid. Het is een standaard schematechniek om een database structuur visueel voor te stellen. Het is een beetje vergelijkbaar met ULM (Unified Modelling Language) dat gebruikt wordt om objecten en klassen grafisch weer te geven.

DBA

Een **DBA** is een **Database Administrator**. Dit is een persoon die de database beheert. Dit wil zeggen dat deze persoon de gebruikers aanmaakt, beveiligingen instelt, backup neemts, controles uitvoert en er voor zorgt dat de database optimaal blijft werken.

Veel gebruikte databases en hun fabrikanten

Kleine databases

Kleine database zijn meestal file-based en worden lokaal geïnstalleerd. Deze databases zijn klein, gemakkelijk en bieden de nodige mogelijkheden voor huis-tuin-en-keuken gebruik.

-  **Access (Microsoft)** : Single file database waarin ook formulieren, rapporten, queries en VBA programmacode kan bewaard worden. Access zelf heeft een ontwikkelingsomgeving om een applicatie voor de database te maken.

-  **Paradox (Borland)** : Single file database waarbij elke tabel ook een ander bestand vormt. Werd in het verleden veel gebruik door Delphi en Clipper ontwikkelaars.

- **DBase :** Onder DOS was dit de meestgebruikte database, maar onder Windows wordt het nog weinig gebruikt.

Middelgrote (tot grote) databases

Deze databases werken meestal via het client-server principe en laten toe om met meerdere gebruikers op eenzelfde database te werken. Ze ondersteunen transacties en meestal ook programmatie-mogelijkheden zoals functies, stored procedures en triggers.

- 

MySQL : Platform onafhankelijke open-source database. Zeer snel en populair op webservers. T.o.v sommige andere databases zijn de mogelijkheden nog beperkt, maar aan volgende versies wordt zeer hard gewerkt. http://www.mysql.com

- 

Interbase (Borland) : Platform onafhankelijke database. Gemakkelijk in onderhoud en vrij uitgebreid qua functionaliteit zoals transacties, triggers, stored procures, … http://www.borland.com/interbase.

-  Open source versie van Interbase. Iets minder mogelijkheden dan de commerciële Borland versie van Interbase, maar wel gratis en met veel bugfixes. http://firebird.sourceforge.net, http://www.firebirdsql.org

-  **PostgreSQL :** Open-source database die voornamelijk onder Linux gebruikt wordt. http://www.pgsql.com

- **FoxPro (Microsoft)** http://msdn.microsoft.com/vfoxpro/
- **Ingress (Computer Associates)** http://www3.ca.com/Solutions/Product.asp?ID=1013

Grote databases

Op de markt van de zeer grote databases zijn er eigenlijk maar 3 spelers; Microsoft, Oracle en IBM. Deze databases met ontzettend veel mogelijkheden kunnen gigabytes tot terrabytes aan gegevens bevatten en laten het toe om met veel gebruikers tegelijkertijd te werken. Het opzetten en onderhoud van deze databases vraagt veel kennis.

-  **SQL Server (Microsoft)** : Krachtige database die enkel beschikbaar is voor Windows servers. http://www.microsoft.com/sql/default.asp

-  **Oracle** : Platform onafhankelijke database. De database met de langste geschiedenis. Mogelijkheden qua triggers, stored procedures, replicatie, beveiligingen, … zijn bijna onbeperkt. Mogelijk tot dedicated server door eigen besturingssyteem. http://www.oracle.com

-  **DB2 (IBM) :** Krachtige platform onafhankelijk database die voornamelijk op mainframes gebruikt wordt. http://www-3.ibm.com/software/data/db2

- **Informix (IBM) :** Database die ondertussen is opgekocht door IBM. http://www-3.ibm.com/software/data/informix

-  **Sybase** http://www.sybase.com/home

MySQL

http://www.mysql.com

About MySQL AB
MySQL AB is the company that develops, supports and markets the MySQL database server globally. Our mission is to make superior data management available and affordable for all, and to contribute to building the mission-critical high-volume systems and products of tomorrow. We have made our product available at zero price under the GNU General Public License (GPL), and we also sell it under a commercial license to those who do not wish to be bound by the terms of the GPL.

The MySQL database server embodies an ingenious software architecture that maximises speed and customisability. Extensive reuse of pieces of code within the software and an ambition to produce minimalistic but functionally rich features have resulted in a database management system unmatched in speed, compactness, stability and ease of deployment. The unique separation of the core server from the table handler makes it possible to run MySQL under strict transaction control or with ultrafast transactionless disk access, whichever is most appropriate for the situation.

Today MySQL is the most popular open source database server in the world with more than 4 million installations powering websites, datawarehouses, business applications, logging systems and more. Customers such as Yahoo! Finance, MP3.com, Motorola, NASA, Silicon Graphics, and Texas Instruments use the MySQL server in mission-critical applications.

The company was set up in Sweden by two Swedes and a Finn: David Axmark, Allan Larsson and Michael "Monty" Widenius who have worked together since the 80's. MySQL AB is the sole owner of the MySQL server source code, the MySQL trademark and the mysql.com domain worldwide. The company is privately held and without debt, and it is financed by venture capital since July 2001.

MySQL AB employs some 55 staff around the globe, and thousands more contribute to the success of MySQL by testing the software, integrating it into other software products, and writing about it.

MySQL AB has three main sources of revenue

- Online support and subscription services sold globally over the MySQL.com website to all users of the MySQL server.
- Sales of commercial MySQL licenses to users and developers of software products and of products that contain software.
- Franchise of MySQL products and services under the MySQL brand to value-added partners.
- We provide our customers and partners with support services they can depend on, consulting services, training programs, and more.

We have described our core values as follows. We want the MySQL server to be

- The best and the most used database in the world
- Available and affordable for all
- Easy to use
- Continuously improved while remaining fast and safe
- Fun to use and improve
- Free from bugs

Firebird (Interbase)

http://firebird.sourceforge.net, http://www.firebirdsql.org

Introduction to Firebird
In August 2000, Borland Software Corp. (formerly known as Inprise) released the beta version of InterBase 6.0 as open source. The community of waiting developers and users preferred to establish itself as an independent, self-regulating team rather than submit to the risks, conditions and restrictions that the company proposed for community participation in open source development. A core of developers quickly formed a project and installed its own source tree on SourceForge. They liked the Phoenix logo which was to have been ISC's brandmark and adopted the name "Firebird" for the project.

Because Borland's open source efforts regarding InterBase never really took off beyond prime release of the source code and the company returned its focus to closed commercial development, Firebird became THE Open Source version of InterBase.

Although we made many attempts to reunite our efforts with Borland's to strengthen the position of InterBase and its further development, it's obvious (especially in the light of events around recent release of Borland InterBase 6.5) that our definitive divorce is inevitable. So you can see Firebird as an independent product that's almost 100% compatible with Borland InterBase. Our intention is to keep backward compatibility with Firebird/InterBase, but we can't guarantee that Firebird will support all InterBase features beyond its version 6.0. Anyway, because Firebird and InterBase user's community overlap in large, we'll always look for ways to facilitate the co-existence of both products, or at least secure the migration path.

About Firebird, InterBase's successor
In contrast to the open source InterBase 6 and new versions of InterBase provided by Borland that is available only on Windows, Linux and Solaris, Firebird has been successfully built and run on additional platforms like MacOS X (Darwin), FreeBSD and OpenBSD, HP-UX, AIX and others as well.

Firebird 1.0 improves on InterBase in more ways than just the number of available platforms. Firebird excels in how bugs (and some bugs always appear in any piece of software) are handled. As in any other Open Source project, the whole development process is transparent to everyone, so you know the exact state of a bug that you have reported (or anyone else's for that matter), when it's fixed and how it was fixed, and the download the fixed version of the software as soon as it is available. Although bugs are found and fixed by development team on a regular basis, you too can have an influence on the development process, by providing your feedback, your assistance or helping to finance the work.

The Firebird development team doesn't just concentrate solely on bug fixes, but it also adds new features and improvements as well. Because Firebird developers are also Firebird everyday users, they are focused on features and improvements that really "scratch an itch", rather than on features for features sake like you find in many commercial products.

What is the project's mission ?
The project's primary goal is to enhance and develop the Firebird Relational Database Engine, but our efforts have also expanded into closely related areas like documentation for end users and developers, testing, connectivity options such as JDBC, ODBC and .NET drivers, plus related management tools.

Who's behind Firebird ?
The Firebird project has currently around 60 plus active members. Of course, that doesn't mean that there are 60+ developers working directly on Firebird code. Some people volunteer for various tasks that surround the core development work, such as documentation, building, testing and packaging on various platforms, mentoring and providing technical advice, web site maintenance etc. Many members are dedicated to a particular sub-project, or to particular problem area. The Firebird Team consists of many skilled and enthusiastic members including primary Interbase developers, former Interbase engineers, experienced Interbase users, and complete newcomers keen to lend a hand in any way they can. This diverse, multi-talented, and ever-growing community is our greatest asset -- one that guarantees a very healthy future for the Firebird Project.

SQL : Structured Query Language

Inleiding SQL

Een Database Management System (DBMS) moet een tekstuele taal bevatten waarmee je tabellen kan creëren en gegevens opvragen. De meest gebruikte taal bij relationele DBMS (RDBMS) is **SQL** . In deze tekst volgt een beschrijving van een aantal commando's uit de Access variant van SQL.

De database management talen bevatten twee soorten commando's, die meestal statements worden genoemd:

- De **Data Definition** statements worden o.m. gebruikt om tabelstructuren te beheren. Ze vormen de **Data Definition Language (DDL)**.
- De **Data Manipulation** statements worden gebruikt om de eigenlijke gegevens op te vragen en te wijzigen. Ze laten de tabelstructuren daarentegen ongewijzigd. Ze vormen de **Data Manipulation Language (DML**).

SQL is een wereldwijde standaard maar toch bestaan er vele SQL dialecten. Elke database voegt zo zijn eigen functionaliteit toe en het principe van joins is in vele databases op een andere manier geïmplementeerd. In deze cursus bekijken we SQL taal van Microsoft Access. Achteraan worden de verschillen met Interbase/Firebird, MySQL en Oracle weergegeven.

SQL in MS Access

Een **SQL** statement bevat een aantal sleutelwoorden, en begint steeds met een sleutelwoord. In het algemeen zal men de sleutelwoorden in hoofdletters vermelden en de niet-sleutelwoorden zoals tabel- en veldnamen niet. Hiertoe bestaat echter geen verplichting. Je kan verschillende regels gebruiken om een commando te formuleren. In Access SQL (en vele andere varianten) moet je SQL statements van elkaar scheiden met een kommapunt (**;**). Standaard wordt elke Access SQL statement met een **;** beëindigd, ook wanneer het statement op zichzelf staat.

Een SQL statement wordt opgesplitst in een aantal clauses, die elk met een sleutelwoord beginnen.

In de hiernavolgende syntaxis van de statements worden optionele elementen tussen [ ] vermeld. Alternatieve elementen worden tussen { } vermeld en van elkaar gescheiden door een | . Een herhaling wordt aangeduid door drie ...

Het veruit belangrijkste statement is het **SELECT** statement, dat behoort tot de **DML** en waarmee je gegevens kan opvragen. Wanneer je in Access een **Query** opstelt, kan je dit in een **Query By Example** venster doen in het **Query Design View**. Je kan echter ook de query intypen in het **SQL View**. Welke werkwijze je ook gebuikt, steeds wordt door Access automatisch de andere versie geproduceerd. Wanneer je een SQL query bewaart, gebeurt dit steeds in zijn tekstuele **SQL** gedaante.

1. De SQL Data Definition Language (DDL)

In de **Data Definition Language (DDL)** kan je onder meer nieuwe tabellen creëren, de structuur van bestaande tabellen wijzigen, en tabellen verwijderen.

In Access wordt de **SQL DDL** weinig gebruikt omdat de **QBE** (**Query By Example**) methode gebruiksvriendelijker en uitgebreider is. Daarom worden hierna slechts beknopt een aantal commando's met hun betekenis en een voorbeeld vermeld.

1.1 CREATE TABLE

Je creëert een nieuwe tabel in Access met de **New** knop in het **Tables** tabblad van het **Database** venster.

In SQL gebruik je het statement

```
CREATE TABLE tablename
   (fieldname datatype [(fieldsize)]
    [CONSTRAINT name {PRIMARY KEY | UNIQUE | REFERENCES tablename}]
    [, ...]
    [, CONSTRAINT name
       {PRIMARY KEY (fieldname [, ...]) | UNIQUE (fieldname [, ...])
        | FOREIGN KEY (fieldname [, ...]) REFERENCES tablename} [, ...]]);
```

Met de `CREATE TABLE tablename` clause geef je de nieuwe tabel een naam.

Met `(fieldname datatype [(fieldsize)] [, ...])` geef je de lijst van velden aan met hun data type en eventuele grootte. Dit komt overeen met de *field names, data types* en *size* en eventueel *format properties* in het Table Design venster.

De SQL notaties die overeenkomen met de Access datatypes, verschillen nogal van de SQL/92 standaard:

Data type (field size) (QBE)	datatype (SQL - Access)	datatype (SQL/92 standaard)
Text	TEXT (of CHAR, CHARACTER)	CHARACTER [VARYING]
Memo	LONGTEXT (of MEMO)	-
Number (Byte) Number (Integer) Number (Long Integer) Number (Single) Number (Double)	BYTE SHORT (of SMALLINT) LONG (of INT, INTEGER) SINGLE (of REAL) DOUBLE (of FLOAT)	- SMALLINT INTEGER REAL DOUBLE PRECISION, FLOAT
Date/Time	DATETIME (of DATE, TIME, TIMESTAMP)	DATE, TIME, TIMESTAMP
Currency	CURRENCY	-
AutoNumber (Long Integer)	COUNTER	-
Yes/No	BOOLEAN (of LOGICAL, YESNO)	-

Na elke *fieldname* kan je met de `CONSTRAINT` *`name`* `{PRIMARY KEY | UNIQUE | REFERENCES` *`tablename`*`}` clause aanduiden dat het veld een primaire sleutel is (`PRIMARY KEY`), unieke waarden moet bevatten (`UNIQUE`), of een vreemde sleutel is die verwijst naar de primaire sleutel van een andere tabel (`REFERENCES` *`tablename`*). *Name* is de naam van de index die je terugvindt in het Index venster.

Na het laatste veld kan je met de `,CONSTRAINT` *`name`* `{PRIMARY KEY (`*`fieldname`*` [, ...]) | UNIQUE (`*`fieldname`*` [, ...]) | FOREIGN KEY (`*`fieldname`*` [, ///]) REFERENCES` *`tablename`*`} [, ...]` clause nog indexen definiëren die samengesteld zijn uit meerdere velden. Hierbij moeten telkens de velden worden vermeld waaruit de samengestelde primaire sleutel, sleutel met unieke waarden of vreemde sleutel is samengesteld.

Je kan één enkele primaire sleutel definiëren met de CONSTRAINT *name* PRIMARY KEY clause. Wanneer je één enkel veld als primaire sleutel wilt definiëren, plaats je CONSTRAINT *name* PRIMARY KEY onmiddellijk achter de veldnaam. Wanneer je meerdere velden als primaire sleutel wilt definiëren, plaats je CONSTRAINT *name* PRIMARY KEY (*fieldname* [, ...]) na de laatste veldnaam en gescheiden van deze veldnaam door een komma.

Het sleutelwoord UNIQUE komt overeen met de Indexed (Yes, No Duplicates) field property en kan meermaals voorkomen.

Je kan vreemde sleutels definiëren met de REFERENCES clause, die meermaals kan voorkomen. Na REFERENCES wordt de naam van de tabel vermeld waarmee een relatie wordt gelegd. Wanneer je één enkel veld als vreemde sleutel wilt definiëren, plaats je CONSTRAINT *name* REFERENCES *tablename* onmiddellijk achter de veldnaam. Wanneer je meerdere velden als vreemde sleutel wilt definiëren, plaats je CONSTRAINT *name* FOREIGN KEY (*fieldname* [, ...]) REFERENCES *tablename* na de laatste veldnaam en gescheiden van deze veldnaam door een komma.

In de SQL/86 standaard kan je in het CREATE TABLE statement geen primaire of vreemde sleutel definiëren , maar je kan wel aanduiden dat een veld uniek moet zijn (met het sleutelwoord UNIQUE). In de SQL/92 standaard kan je een primaire of vreemde sleutel definiëren op een analoge wijze als in Access 97, maar het woord CONSTRAINT en de *name* vallen weg en de definitie gebeurt steeds op het einde van het CREATE TABLE statement

Voorbeeld:

De creatie van de tabel **Studenten** gebeurt in SQL als volgt:

Open het **SQL View**. Hiervoor moet je in het **Database** venster het **Queries** tabblad selecteren en de **New** (of **Design**) knop klikken. Vervolgens schakel je met de **View** knop over naar het **SQL View** en typ je in het **SQL** venster het volgende statement in:

```
CREATE TABLE Studenten
    (studentnr LONG CONSTRAINT PrimaryKey PRIMARY KEY,
     achternaam TEXT(25),
     voornaam TEXT(15),
     adres TEXT(30),
     postcode TEXT(4),
     woonplaats TEXT(30),
     telefoon TEXT(13),
     geb_datum DATE,
     geslacht TEXT(5),
     vooropleiding LONGTEXT,
     jaar SMALLINT,
     klas TEXT(4));
```

1.2 ALTER TABLE

Je kan de structuur van een bestaande tabel in Access wijzigen met de **Design** knop in het **Tables** tabblad van het **Database** venster.

In SQL gebruik je het statement

```
ALTER TABLE tablename
 {ADD COLUMN fieldname datatype [(fieldsize)]
   [CONSTRAINT name {PRIMARY KEY | UNIQUE | REFERENCES tablename}]
 | ADD CONSTRAINT name
   {PRIMARY KEY (fieldname [, ...]) | UNIQUE (fieldname [, ...])
    | FOREIGN KEY (fieldname [, ...]) REFERENCES tablename}
 | DROP COLUMN fieldname
 | DROP CONSTRAINT name};
```

Met de `ALTER TABLE tablename` clause duidt je de tabel aan waarvan je het ontwerp wil wijzigen. Daarna kan je één uit vier mogelijke wijzigingen aanbrengen:

Met de `ADD COLUMN fieldname datatype [(fieldsize)][CONSTRAINT name {PRIMARY KEY | UNIQUE | REFERENCES tablename}]` clause kan je een nieuw veld met bijbehorende indexen achteraan toevoegen.

Met de `ADD CONSTRAINT name {PRIMARY KEY (fieldname [, ...]) | UNIQUE (fieldname [, ...]) | FOREIGN KEY (fieldname [, ...]) REFERENCES tablename}` clause kan je een nieuwe samengestelde index toevoegen.

Met de `DROP COLUMN fieldname` clause kan je een veld verwijderen.

Met de `DROP CONSTRAINT name` kan je een samengestelde index verwijderen.

Voorbeeld:

Het leggen van een relatie tussen twee tabellen gebeurt in SQL door een veld (of meerdere velden) als een vreemde sleutel te definiëren en aan te geven naar welke (primaire sleutel van een) tabel wordt verwezen. Je legt als volgt een relatie tussen de tabellen **Stagiaires** en **Studenten**:

Open het **SQL View**. Hiervoor moet je in het **Database** venster het **Queries** tabblad selecteren en de **New** (of **Design**) knop klikken. Vervolgens schakel je met de **View** knop over naar het **SQL View** en typ je in het **SQL** venster het volgende statement in:

```
ALTER TABLE Stagiaires
    ADD CONSTRAINT studentnr
        FOREIGN KEY (studentnr) REFERENCES Studenten;
```

Hierbij wordt aangenomen dat `studentnr` reeds vooraf als primaire sleutel voor de tabel Studenten werd gedefinieerd.

1.3 DROP TABLE

Je kan een bestaande tabel in Access verwijderen door hem in het **Tables** tabblad van het **Database** venster te selecteren en de Delete knop in te drukken.

In SQL gebruik je hiervoor het statement

```
DROP TABLE tablename;
```

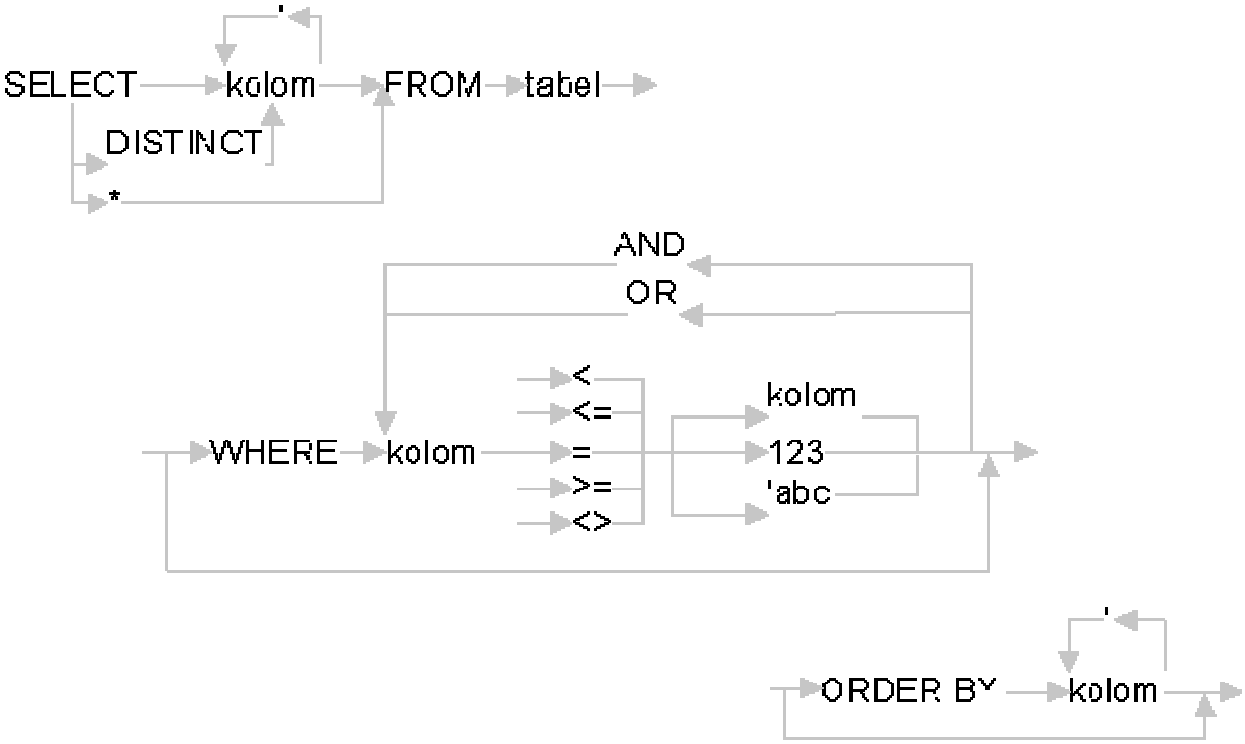
2. De SQL Data Manipulation Language (DML)

In Access is SQL vooral interessant als alternatieve taal voor QBE om queries op te stellen. Deze queries vallen uiteen in twee categorieën:
Met een Select Query kan je gegevens uit één of meerdere tabellen opvragen, zonder de inhoud te wijzigen. Het equivalente SQL statement heet SELECT.
Met een Action Query kan je gegevens in een tabel wijzigen. Access kent vier soorten Action Queries:

1. Met een Make Table Query maak je een nieuwe tabel door de resultaten van een Select Query als een nieuwe tabel te bewaren. Het equivalente SQL statement heet SELECT INTO
2. Met een Update Query wijzig je de waarden van één of meer velden in een tabel voor één of meer records. Het equivalente SQL statement heet UPDATE.
3. Met een Append Query voeg je nieuwe records toe aan een tabel. Het equivalente SQL statement heet INSERT INTO.
4. Met een Delete Query verwijder je één of meer records uit een tabel. Het equivalente SQL statement heet DELETE.

3. Het SELECT statement

Met het SELECT statement kan je gegevens uit één of meerdere tabellen opvragen.



Hierbij moet je een onderscheid maken tussen het SELECT statement zelf, dat eigenlijk een programma is, en de tabel die als resultaat van het SELECT statement wordt getoond. Deze tabel is virtueel. Dit betekent dat hij niet echt bestaat, maar bij het uitvoeren van het SELECT statement wordt opgesteld en slechts bestaat zolang het bijhorende venster geopend blijft. Wanneer je gegevens in een virtuele tabel tracht te wijzigen, zullen de wijzigingen in werkelijkheid in de onderliggende echte tabellen (tables, ook base tables of reële tabellen genoemd) gebeuren. In vele gevallen is er geen eenduidig verband tussen een rij of kolom uit een virtuele tabel en een rij of kolom uit een reële tabel. In die gevallen zal de virtuele tabel of een deel ervan niet kunnen worden gewijzigd.

In Access wordt een virtuele tabel die het resultaat is van een query een **recordset** genoemd. Heel vaak kan je hierin de inhoud van sommige of alle rijen of kolommen wijzigen. Dan spreekt Access van een **dynaset** (dynamische verzameling). Soms kan je geen enkele waarde in de resulterende tabel wijzigen. Dan spreekt Access van een **snapshot** (momentopname).

Hierbij zijn enkele belangrijke opmerkingen nodig:

In standaard SQL bestaan zowel het SELECT statement als de virtuele tabel tijdelijk. Je kan daar niet (zoals in Access) het SELECT statement bewaren.

In standaard SQL komt een query die je (als programma) bewaart overeen met de definitie van een **view**. Het gebruikte statement heet dan CREATE VIEW AS (SELECT ...). De definitie van de view wordt bewaard. Dit komt (ongeveer) overeen met een bewaarde query in Access. Je kan dus (ongeveer) stellen dat de objecten in het **Query** tabblad van het **Database** venster met de definitie van **views** (virtuele tabellen met een naam) overeenkomen i.p.v. met queries. Queries komen eigenlijk overeen met de definities van virtuele tabellen die je geen naam geeft en niet bewaart, omdat je ze slechts eenmalig nodig hebt. In deze tekst wordt steeds de Access term **query** i.p.v. **view** gebruikt.

In Access kan je met SQL meer soorten **Queries** opstellen dan met **Query By Example**. Zo kan je een **Union Query**, een **Pass Through Query** en een **Data Definition Query** enkel in SQL opstellen. Hierop wordt niet verder ingegaan.

Algemeen bevat een SELECT query zes soorten clauses, waarvan de eerste twee steeds voorkomen, en de vier andere kunnen voorkomen:

```
SELECT fieldlist
FROM tablelist
WHERE condition
GROUP BY group_fieldlist
HAVING group_condition
ORDER BY fieldlist;
```

In de SELECT clause definieer je de kolommen van de virtuele tabel die je wil opstellen.

In de FROM clause geef je aan uit welke tabellen je gegevens komen.

In de WHERE clause geef je voorwaarden op waarin je bepaalt welke rijen wel of niet zullen worden getoond.

In de GROUP BY clause geef je aan of je rijen wilt groeperen.

In de HAVING clause geef je eveneens voorwaarden op, waarin je bepaalt welke rijen wel of niet worden getoond. Deze voorwaarden gelden echter voor gegroepeerde rijen.

In de ORDER BY clause geef je de volgorde van de getoonde rijen aan.

3.1 SELECT in één tabel

De eenvoudigste vorm van het SELECT statement haalt de gegevens uit één enkele tabel, die in de FROM clause wordt vermeld.

3.1.1 De SELECT en FROM clauses

De SELECT en FROM clauses zijn steeds de eerste clauses van een SELECT statement.

De syntaxis van de SELECT en FROM clauses is:

```
SELECT [{ALL | DISTINCT | DISTINCTROW | TOP n [PERCENT]}]
    fieldname [AS alias] [, ...]
    FROM tablename
```

In zijn eenvoudigste vorm `SELECT fieldname [, ...] FROM tablename;` worden de aangegeven kolommen uit een tabel getoond in de opgegeven volgorde.

Als *fieldname* mag * worden vermeld. Dit is een korte notatie voor alle velden uit de tabel, in dezelfde volgorde als in de tabel.

Fieldname hoeft niet noodzakelijk een veld uit een tabel te zijn. Het kan ook een berekend veld zijn zoals **prijs*hoeveelheid**. In dit laatste geval spreek je beter van een berekende kolom, aangezien deze kolom niet met een veld overeenkomt. De inhoud van een berekende kolom is read-only.

Je kan elke kolom een andere naam geven door na *fieldname* `AS` *alias* te vermelden. De *alias* geldt dan als alternatieve kolomnaam. Hij wordt als kolomhoofd getoond. Bij dubbele voorkomende kolommen of kolomnamen moet je een *alias* opgeven. Bij berekende kolommen hoef je geen *alias* op te geven, maar je doet dit wel best.

Als *fieldname* kan een zogenaamde *aggregate function* worden gebruikt, waarbij totalen voor alle records worden berekend, indien er geen `GROUP BY` clause is, en totalen per groep, indien er een `GROUP BY` clause is (zie infra). In standaard SQL zijn er vijf *aggregate functions* (`COUNT` - aantal, `SUM` - totaal, `AVG` - gemiddelde, `MAX` - grootste waarde, `MIN` - kleinste waarde). Access SQL heeft daarenboven extra *aggregate functions* (`STDEV` - standaardafwijking van de steekproef, `VAR` - variantie van de steekproef, `STDEVP` - standaardafwijking van de populatie, `VARP` - variantie van de populatie). Daarnaast kan je ook berekende totalen opstellen. Per groep wordt één rij getoond. Het resultaat is steeds een **snapshot** en kan niet worden gewijzigd.

Wanneer een *aggregate function* wordt gebruikt, moeten alle andere te tonen kolommen

ofwel ook een *aggregate function* zijn,

ofwel een groeperingsveld zijn (vermeld in de GROUP BY clause),

Na het sleutelwoord SELECT kan één van vier sleutelwoorden volgen. Deze hebben de betekenis:

Met `SELECT ALL` (standaard, dus gelijkwaardig met `SELECT`) toon je alle records, waarbij dubbele rijen worden herhaald.

Met `SELECT DISTINCT` elimineer je dubbele rijen. Elke getoonde rij verschilt van de andere. Hierdoor kan een getoonde rij met meerdere rijen uit de onderliggende tabel overeenkomen, waardoor ze read-only wordt. Het gebruik van het sleutelwoord `DISTINCT` heeft dus steeds als gevolg dat het resultaat een **snapshot** is.

Met `SELECT DISTINCTROW` toon je alle rijen, waarvan de onderliggende records van elkaar verschillen. Dubbele rijen worden enkel geëlimineerd wanneer alle veldwaarden uit de onderliggende tabel gelijk zijn. Bij `SELECT DISTINCT` worden daarentegen rijen geëlimineerd wanneer alle getoonde veldwaarden gelijk zijn. Niet getoonde veldwaarden mogen bij `SELECT DISTINCT` wel verschillen, bij `SELECT DISTINCTROW` niet. `SELECT DISTINCTROW` en `SELECT ALL` zijn gelijkwaardig wanneer de tabel een primaire sleutel heeft, of wanneer alle rijen in de tabel van elkaar verschillen. `SELECT DISTINCTROW` komt enkel voor bij Access SQL.

Met `SELECT TOP` *n* `[PERCENT]` toon je enkel de eerste *n* of de eerste *n* % rijen. `SELECT TOP` is enkel zinvol wanneer de records geordend zijn; er is dus een ORDER BY clause nodig.

Voorbeelden:

Neem aan dat de tabel **Klanten** de volgende records en velden bevat:

klantnr (primaire sleutel)	**naam**	**gemeente**
1	Janssens	Brussel
2	Janssens	Brussel
3	Peeters	Brussel

De hiernavolgende SELECT statements leveren de erbij vermelde **recordsets** op:

```
SELECT ALL naam, gemeente FROM Klanten;
```
(of `SELECT naam, gemeente FROM klanten;`):

naam	**gemeente**
Janssens	Brussel
Janssens	Brussel
Peeters	Brussel

`SELECT DISTINCT naam, gemeente FROM Klanten;`:

naam	**gemeente**
Janssens	Brussel
Peeters	Brussel

`SELECT DISTINCTROW naam, gemeente FROM Klanten;`:

naam	**gemeente**
Janssens	Brussel
Janssens	Brussel
Peeters	Brussel

`SELECT TOP 2 naam, gemeente FROM Klanten ORDER BY naam;`:

naam	**gemeente**
Janssens	Brussel
Janssens	Brussel

`SELECT TOP 50 PERCENT naam, gemeente FROM Klanten ORDER BY naam;`:

naam	**gemeente**
Janssens	Brussel
Janssens	Brussel

In dit laatste geval wordt het getoonde percentage (50% van 3) naar boven afgerond. Bij gelijke waarden voor het `ORDER BY` veld worden alle gelijk geordende records getoond.

Wanneer de eerste kolom in de tabel ontbreekt, en er dus geen primaire sleutel is, leveren de hiernavolgende SELECT statements de volgende **recordsets** op:

`SELECT ALL naam, gemeente FROM Klanten;` (of `SELECT naam, gemeente FROM klanten;`):

naam	**gemeente**
Janssens	Brussel
Janssens	Brussel
Peeters	Brussel

`SELECT DISTINCT naam, gemeente FROM Klanten;`:

naam	**gemeente**
Janssens	Brussel
Peeters	Brussel

`SELECT DISTINCTROW naam, gemeente FROM Klanten;`:

naam	**gemeente**
Janssens	Brussel
Peeters	Brussel

In dit laatste geval worden er slechts twee rijen getoond, omdat alle kolommen in de *fieldlist* voorkomen, en er twee identieke rijen zijn.

3.1.2 De WHERE clause

De WHERE clause is niet verplicht in een SELECT statement. Ze wordt gebruikt om via voorwaarden een restrictie op te leggen aan de te tonen records of rijen.

De syntaxis van de WHERE clause is:

`WHERE` *`condition`*

Hierbij is *condition* een uitdrukking waarvan de waarde TRUE of FALSE is. Ze wordt opgebouwd met de logische operatoren AND, OR en NOT en de vergelijkingsoperatoren =, <, >, <=, >= en <>. Daarenboven kunnen ook wildcards (*, ?, #, [ ] en !) en andere operatoren zoals Like, In en Between voorkomen. Het gebruik van ( ) dient om de volgorde van de evaluatie van de operatoren te bepalen.

De Like operator vergelijkt een tekstuitdrukking in het linkerlid, die een wildcard bevat, met een patroon tussen " "in het rechterlid. Een patroon is een veralgemeende tekstuitdrukking, d.w.z. een tekst waarin nog enige vrijheid bestaat. Zo staat "A*" voor alle teksten die met een A beginnen, en staat "?end" voor alle teksten van vier tekens waarbij het eerste teken volledig willekeurig is en de laatste drie letterlijk "end" zijn.

De betekenis van de wildcards is als volgt:

* Nul, één of meer willekeurige tekens

? Exact één willekeurig teken

Exact één willekeurig cijfer

[ ] Een reeks tekens tussen [ ] betekent exact één teken uit de opgegeven reeks. Binnen de [ ] betekent een streepje (-) een bereik van tekens. [A-Za-z] staat dus voor een willekeurige letter. Een uitroepteken betekent een uitsluiting. [!0-9] staat dus voor een willekeurig teken dat geen cijfer is. Wanneer de wildcards als teken worden bedoeld, staan ze tussen [ ].

Voorbeelden (tracht zelf de betekenis uit te zoeken):

```
SELECT * FROM Studenten
WHERE woonplaats = "Antwerpen";
```

```
SELECT * FROM Studenten
WHERE achternaam >="N";
```

```
SELECT * FROM Studenten
WHERE achternaam Like [!V]*
```

```
SELECT * FROM Studenten
WHERE NOT achternaam Like "V*"
```

```
SELECT * FROM Studenten
WHERE postcode Like "[1-3]###"
```

```
SELECT * FROM Begeleiders
WHERE vak Like "*[?]*"
```

```
SELECT * FROM Studenten
WHERE woonplaats In ("Brussel", "Gent", "Leuven");
```

```
SELECT * FROM Studenten
WHERE woonplaats = "Brussel" OR woonplaats = "Gent"
OR woonplaats = "Leuven";
```

```
SELECT * FROM Studenten
WHERE woonplaats = "Brussel" OR achternaam Like "V*";
```

```
SELECT * FROM Studenten
WHERE woonplaats = "Brussel" AND achternaam Like "V*";
```

```
SELECT * FROM Studenten
WHERE geb_datum Between #09/15/72# And #12/31/72#;
```

```
SELECT * FROM Studenten
WHERE geb_datum >= #09/15/72# AND geb_datum <= #12/31/72#;
```

```
SELECT stagenr, [einde stage] - [begin stage] AS [duur stage]
FROM Studenten
WHERE [begin stage] >="01/01/72#;
```

3.1.3 De ORDER BY clause (*)

De ORDER BY clause is niet verplicht in een SELECT statement. Ze wordt gebruikt om een ordening op te leggen aan de te tonen records of rijen.

De syntaxis van de ORDER BY clause is:

`ORDER BY` *`fieldname`* `[{ASC | DESC}][, ...]`

De getoonde rijen worden geordend op de waarden in de kolommen die worden opgegeven in de `ORDER BY` clause. Indien er meerdere kolommen zijn, wordt eerst geordend op de eerste kolom, daarna op de volgende enz. Standaard wordt stijdend geordend (0-9 en A-Z). Je kan de volgorde van ordenen wijzigen door na het *fieldname* `ASC` (stijgend) of `DESC` (dalend) te vermelden. Als *fieldname* moeten veldnamen of uitdrukkingen worden gebruikt, maar geen aliassen! Wanneer de ORDER BY clause ontbreekt, staat de volgorde van de rijen niet vast.

3.1.4 De GROUP BY clause (*)

De GROUP BY clause is niet verplicht in een SELECT statement. Ze wordt gebruikt wanneer via *aggregate functions* totalen berekend worden voor groepen records i.p.v. voor alle records samen.

De syntaxis van de GROUP BY clause is:

```
GROUP BY field [, ...]
```

Een groep bestaat uit alle records met gelijke waarden voor alle velden (of uitdrukkingen) in de GROUP BY clause. Niet in de GROUP BY clause vermelde kolommen die wel in de SELECT clause voorkomen, moeten een *aggregate function* bevatten.

Voorbeelden:

```
SELECT Count(*) AS aantal, Avg([einde stage] - [begin stage]) AS
[gemiddelde duur] FROM Stages;
```

```
SELECT Count(*) AS aantal, Avg([einde stage] - [begin stage]) AS
[gemiddelde duur] FROM Stages
GROUP BY instellingsnr;
```

In de eerste query wordt het aantal stages en de gemiddelde duur van een stage berekend. De laatste formule is een berekend totaal. Er is één enkele rij.

In de tweede query worden dezelfde berekeningen gemaakt, maar dan opgesplitst per instelling. Er is één rij per instelling.

3.1.5 De HAVING clause (*)

De HAVING clause is niet verplicht in een SELECT statement. Ze wordt gebruikt om via voorwaarden een restrictie op te leggen aan groepen. Dit betekent dat ze in principe enkel samen met de GROUP BY clause voorkomt .

De syntaxis van de HAVING clause is:

```
HAVING condition
```

Hierbij is *condition* een uitdrukking waarvan de waarde TRUE of FALSE is. Ze wordt opgebouwd met de logische operatoren AND, OR en NOT en de vergelijkingsoperatoren =, <, >, <=, >= en <>. Daarenboven kunnen wildcards zoals * en ? en andere operatoren zoals Like, In en Between voorkomen. Het gebruik van haakjes ( ) dient om de volgorde van de evaluatie van de operatoren te bepalen.

Het verschil tussen de WHERE en HAVING clauses bestaat er in dat de velden in de *condition* van de HAVING clause ofwel velden zijn die binnen een groep één waarde aannemen (maar dan kan een WHERE clause worden gebruikt, wat efficiënter is), ofwel *aggregate functions* bevatten.

Een tweede verschil tussen de WHERE en de HAVING clause betreft het ogenblik waarop de voorwaarde wordt getest. Bij de WHERE clause gebeurt dit voor individuele rijen vóór het groeperen; bij de HAVING clause gebeurt dit voor de groepstotalen na het groeperen.

Aan de hand van enkele voorbeelden wordt hierna het verschil in het gebruik van WHERE en HAVING geïllustreerd. De queries zijn gebaseerd op de volgende tabel Klanten:

naam (primaire sleutel)	geslacht	woonplaats	omzet
Adriaens	vrouw	Brugge	50 000
Bellens	man	Brussel	20 000
De Bock	vrouw	Gent	44 000
De Groot	vrouw	Brussel	36 000
Mortier	man	Kortrijk	15 000
Peeters	man	Brussel	29 000
Rogiers	man	Gent	97 000
Van den Broecke	man	Antwerpen	35 000
Willems	vrouw	Antwerpen	27 000
Wouters	man	Lier	16 000

```
SELECT woonplaats, Count(*) AS aantal, Sum(omzet) AS [totale omzet] FROM
Klanten
WHERE geslacht = "man"
GROUP BY woonplaats
ORDER BY woonplaats;
```

woonplaats	aantal	totale omzet
Antwerpen	1	35 000
Brussel	2	49 000
Gent	1	97 000
Kortrijk	1	15 000
Lier	1	16 000

```
SELECT woonplaats, Count(*) AS aantal, Sum(omzet) AS [totale omzet] FROM
Klanten
GROUP BY woonplaats
HAVING Count(*)>1
ORDER BY woonplaats;
```

woonplaats	aantal	totale omzet
Antwerpen	2	62 000
Brussel	3	85 000
Gent	2	141 000

```
SELECT woonplaats, Count(*) AS aantal, Sum(omzet) AS [totale omzet] FROM
Klanten
WHERE woonplaats = "Brussel" OR woonplaats = "Antwerpen"
GROUP BY woonplaats
ORDER BY woonplaats;
```

```
SELECT woonplaats, Count(*) AS aantal, Sum(omzet) AS [totale omzet] FROM
Klanten
GROUP BY woonplaats
HAVING woonplaats = "Brussel" OR woonplaats = "Antwerpen"
ORDER BY woonplaats;
```

woonplaats	aantal	totale omzet
Antwerpen	2	62 000
Brussel	3	85 000

```
SELECT woonplaats, Count(*) AS aantal, Sum(omzet) AS [totale omzet] FROM
Klanten
WHERE geslacht = "man"
GROUP BY woonplaats
HAVING Count(*)>1
ORDER BY woonplaats;
```

woonplaats	aantal	totale omzet
Brussel	2	49 000

3.2 SELECT in meerdere tabellen

Naast queries gebaseerd op één tabel zijn er ook queries gebaseerd op meerdere tabellen. Hierbij worden tabellen samengevoegd. Men spreekt dan van een **join** tussen tabellen. Er zijn drie grote soorten joins:

Bij een **equi join** worden tabellen records uit twee (of meer) tabellen samengevoegd op basis van gelijke waarden tussen één (of meer) velden in beide tabellen. Deze velden zullen dan meestal het zelfde data type, field size en format hebben. Belangrijker is dat zij een zelfde betekenis hebben (een zelfde soort attribuut aanduiden). Het meest voorkomende geval treedt op bij een **one to many** (of **many to one**) relatie tussen twee tabellen. De primaire sleutel van de tabel aan de **one** zijde van de relatie en de vreemde sleutel van de tabel aan de **many** tabel van de relatie moeten gelijke waarden hebben. Men zal dan meestal beide sleutelvelden dezelfde naam geven (al hoeft dit niet).

Bij een **theta join** wordt een relatie gelegd op basis van een ongelijkheid tussen de waarden van één of meer velden in beide tabellen.

Bij een **product** wordt elke rij van de éne tabel gekoppeld aan elke rij van de andere tabel. Dit komt overeen met het begrip productverzameling uit de relatieleer in de moderne wiskunde. gewoonlijk resulteert dit in een query waarvan de dynaset een zeer groot aantal rijen bevat (het product van het aantal rijen van beide tabellen).

Je kan elk van deze joins aanduiden m.b.v. de FROM en de WHERE clause:

In de FROM clause vermeld je de gejoinde tabellen, gescheiden door komma's.

In de WHERE clause neem je de gelijkheid (bij een equi join), de ongelijkheid (bij een theta join) tussen de velden die de relatie bepalen, of helemaal niets (bij een product) op als extra voorwaarde.

Algemene vereenvoudigde syntaxis (de niet relevante clauses worden weggelaten):

Equi join:

```
SELECT fieldlist
FROM table1, table2
WHERE table1.field1 = table2.field2;
```

Theta join:

```
SELECT fieldlist
FROM table1, table2
WHERE table1.field1 T table2.field2;
```

Product:

```
SELECT fieldlist
FROM table1, table2;
```

Hierbij zijn *table1* en *table2* de gejoinde tabellen. *Field1* en *field2* zijn de velden uit resp. *table1* en *table2* die de relatie bepalen. *T* is een symbool dat een vergelijkingsoperator verschillend van = aanduidt, bijvoorbeeld kleiner dan (<) of groter dan (>) (Gewoonlijk wordt hier de Griekse hoofdletter Theta voor gebruikt, maar op het world wide web kiezen we de gewone letter T).

Algemeen zullen veldnamen in een query gebaseerd op meer dan één tabel worden voorafgegaan door de naam van de tabel waaruit zij afkomstig zijn en een punt. Wanneer er geen dubbelzinnigheden zijn mag je deze tabelnaam (en het punt) echter weglaten.

In Access SQL en SQL/92 kan bij een **equi join** de join voorwaarde in de WHERE clause worden vermeld, maar zal ze standaard in de FROM clause worden vermeld. Deze heeft dan de gedaante

```
SELECT fieldlist
FROM table1 [jointype] JOIN table2
ON table1.field1 = table2.field2;
```

Deze formulering is enkel mogelijk bij een **equi join** bij een **one to many** of een **one to one** relatie. De linkse tabel *table1* is dan steeds de tabel aan de **one** zijde van de **one to many** relatie en de rechtse tabel *table2* de tabel aan de **many** zijde van de **one to many** relatie.

Het *jointype* kan in Access drie waarden aannemen (zie infra).

Wanneer eenmaal tabellen zijn samengevoegd kunnen de WHERE, GROUP BY, HAVING en ORDER BY clauses worden gebruikt zoals bij één tabel queries. In de resulterende dynaset zullen wel sommige gegevens (uit tabellen aan de **one** zijde van de relatie) herhaald worden getoond, met allerlei gevolgen voor de mogelijkheid tot wijzigen (niet mogelijk of een wijziging in meerdere rijen tegelijk!). Hierop wordt in deze tekst echter niet verder ingegaan.

3.2.1 INNER JOIN

Een **inner join** bevat enkel records uit beide tabellen waartussen een equi join relatie bestaat. Dit is de standaard join, die wordt aangeduid met de het *jointype* INNER. Records uit één van beide tabellen die geen corresponderend record hebben in de andere tabel worden niet getoond in de dynaset.

```
SELECT fieldlist
FROM table1 INNER JOIN table2
ON table1.field1 = table2.field2;
```

Voorbeeld (**one to many** relatie Studenten - Stagiaires):

```
SELECT Stagiaires.studentnr, achternaam, voornaam, stagenr, begeleidersnr
FROM Studenten INNER JOIN Stagiares ON Studenten.studentnr =
Stagiaires.studentnr;
```

Enkel de namen van de studenten die als stagiares stage lopen worden vermeld, met het nummer van hun stage en van hun begeleider.

3.2.2 LEFT JOIN

Een **left outer join** bevat alle records uit de tabel *table1* aan de **one** zijde van de relatie en enkel die records uit de tabel *table2* aan de **many** zijde van de relatie waarvoor een corresponderend record in de andere (**one**) tabel *table1* bestaat. Deze join wordt aangeduid met het *jointype* LEFT. De **join** wordt een **left outer join** genoemd omdat de tabel aan de **one** zijde van de relatie steeds links van de sleutelwoorden LEFT JOIN staat.

```
SELECT fieldlist
FROM table1 LEFT JOIN table2
ON table1.field1 = table2.field2;
```

Voorbeeld (**one to many** relatie Studenten - Stagiaires):

```
SELECT Studenten.studentnr, achternaam, voornaam, stagenr, begeleidersnr
FROM Studenten INNER JOIN Stagiares ON Studenten.studentnr =
Stagiaires.studentnr;
```

Alle namen van de studenten worden vermeld, en voor de studenten die stage lopen ook het nummer van hun stage en van hun begeleider.

3.2.2b LEFT JOIN in Oracle

In Oracle gebruikt men niet de standard methode voor joins. Een join wordt gemaakt door in de WHERE de velden aan elkaar gelijk te stellen. Van zodra je een left join wil maken voeg je een (+) toe aan de linkerkant. Een right join kan ook, maar dan voeg je rechts achter de veldnaam van de 2 tabel de (+) toe.

```
SELECT fieldlist
FROM table1, table2
WHERE table1.field1 (+) = table2.field2;
```

3.2.3 RIGHT JOIN

Een **right outer join** bevat alle records uit de tabel *table2* aan de **many** zijde van de relatie en enkel die records uit de tabel *table1* aan de **one** zijde van de relatie waarvoor een corresponderend record in de andere (**many**) tabel *table2* bestaat. Deze join wordt aangeduid met het *jointype* RIGHT. De **join** wordt een **right outer join** genoemd omdat de tabel aan de **many** zijde van de relatie steeds rechts van de sleutelwoorden RIGHT JOIN staat.

```
SELECT fieldlist
FROM table1 RIGHT JOIN table2
ON table1.field1 = table2.field2;
```

Voorbeeld (**one to many** relatie Studenten - Stagiaires):

```
SELECT Stagiaires.studentnr, achternaam, voornaam, stagenr, begeleidersnr
FROM Studenten RIGHT JOIN Stagiares ON Studenten.studentnr =
Stagiaires.studentnr;
```

Alle nummers van stagiaires worden vermeld, met daarbij de nummers van de studenten, stages en begeleiders.

Omdat met elke stagiare (per definitie) een student correspondeert, is het resultaat hetzelfde als bij de **inner join**. Zoiets gebeurt zeer vaak bij **right outer joins**.

Right outer joins treden in de praktijk vooral op bij **many to many** relaties die opgesplitst zijn in een **one to many** en een **many to one** relatie.

Voorbeeld (**many to many** relatie Studenten - Stagiaires - Begeleiders):

```
SELECT Studenten.studentnr, Studenten.achternaam, Studenten.voornaam,
stagenr, Stagiares.begeleidersnr, Begeleiders.voornaam
FROM Begeleiders RIGHT JOIN (Studenten LEFT JOIN Stagiaires
ON Studenten.studentnr = Stagiaires.studentnr)
ON Begeleiders.begeleidersnr = Stagiaires.begeleidersnr;
```

De plaatsing van de ( ) en de volgorde van de tabellen is steeds zó dat een **one** tabel steeds links van de gejoinde **many** tabel (of combinatie) staat:

Er is een **one to many** relatie Studenten - Stagiaires, en alle studenten moeten worden getoond (linkse tabel, dus LEFT JOIN).

Er is een **one to many** relatie Begeleiders - (Studenten-Stagiaires), en alle studenten moeten worden getoond (rechtse tabel, dus RIGHT JOIN).

Het combineren van uitsluitend **inner joins** onderling is steeds toegelaten. Het combineren van uitsluitend **outer joins** is steeds toegelaten wanneer de tabel waarvan alle rijen getoond worden een **outer join** vertoont met alle andere tabellen (dit betekent in de grafische voorstelling van de joins in het Query Design View dat opeenvolgende pijlen nooit naar elkaar toe mogen wijzen). Het combineren van **inner** en **outer joins** is echter meestal niet toegelaten. Bij het bewaren of tonen van de dynaset treedt dan steeds een foutmelding op.

Wanneer bij dezelfde many to many relatie alle begeleiders en enkel de studenten met een begeleider moeten worden getoond, neemt het SQL statement de volgende gedaante aan:

```
SELECT Stagiaires.studentnr, Studenten.achternaam, Studenten.voornaam,
Stagiaires.stagenr, Begeleiders.begeleidersnr, Begeleiders.voornaam
FROM Studenten RIGHT JOIN (Begeleiders LEFT JOIN Stagiaires
ON Begeleiders.begeleidersnr = Stagiaires.begeleidersnr)
ON Studenten.studentnr = Stagiaires.studentnr;
```

3.3 Voorbeelden

Hierna volgen de SQL statements van de queries uit het handboek 'Basiscursus Access 97':

Oefening 4.3:

```
SELECT [type instelling], naam, plaats, telefoon, contactpersoon
FROM Instellingen;
```

Oefening 4.9:

```
SELECT Studenten.studentnr, voornaam, achternaam, stagenr, begeleidersnr
FROM Studenten LEFT JOIN Stagiaires ON Studenten.studentnr =
Stagiaires.studentnr;
```

Oefening 4.10:

```
SELECT Stages.stagenr, [begin stage], stagiairnr, studentnr, begeleidersnr
FROM Stages INNER JOIN Stagiaires ON Stages.stagenr = Stagiaires.stagenr;
```

Oefening 4.11:

```
SELECT Begeleiders.begeleidersnr, achternaam, stagiairnr
FROM Begeleiders INNER JOIN Stagiaires ON Begeleiders.begeleidersnr =
Stagiaires.begeleidersnr;
```

Oefening 4.12:

```
SELECT stagenr, Instellingen.instellingsnr, naam, plaats
FROM Instellingen INNER JOIN Stages ON Instellingen.instellingsnr =
Stages.instellingsnr;
```

Oefening 4.13:

```
SELECT achternaam, voornaam, adres, postcode, woonplaats, telefoon
FROM Studenten
WHERE woonplaats = "Heerenveen";
```

Oefening 4.13:

```
SELECT achternaam, voornaam, adres, postcode, woonplaats, telefoon
FROM Studenten
ORDER BY voornaam;
```

Oefening 4.19:

```
SELECT Studenten.studentnr, achternaam, Stages.instellingsnr,
Begeleiders.begeleidersnr, Stagiaires.stagiairnr
FROM Studenten INNER JOIN (Stages INNER JOIN (Begeleiders INNER JOIN
Stagiaires
ON Begeleiders.begeleidersnr = Stagiaires.begeleidersnr) ON Stages.stagenr
= Stagiaires.stagenr)
ON Studenten.studentnr = Stagiaires.studentnr
ORDER BY achternaam;
```

Oefening 5.1:

```
SELECT studentnr, achternaam, woonplaats, geb_datum
FROM Studenten
WHERE (woonplaats="Sneek" Or woonplaats="Heerenveen")
AND (geb_datum>=#1/1/74# And geb_datum<=#1/1/75#);
```

Oefening 5.2:

```
SELECT achternaam, woonplaats
FROM Studenten
WHERE achternaam Like "M*" AND woonplaats = "Heerenveen";
```

Oefening 5.3:

```
SELECT naam, plaats, [type instelling]
FROM Instellingen
WHERE (plaats="Sneek" Or plaats="Heerenveen")
AND ([type instelling]="School" Or [type instelling]="Zorgverlening");
```

Oefening 5.4:

```
SELECT stagenr, instellingsnr, stagedagen
FROM Stages
WHERE stagedagen Is Not Null;
```

Oefening 5.5:

```
SELECT voornaam, achternaam, geb_datum
FROM Studenten
WHERE geb_datum Like "23/02/*";
```

Oefening 5.6:

```
SELECT instellingsnr, naam, plaats, telefoon
FROM Instellingen
WHERE plaats In ("Sneek");
```

Oefening 5.7:

```
SELECT stagenr, [begin stage], [einde stage]
FROM Stages
WHERE [begin stage] > #2/15/97#
AND [einde stage] < #6/15/97#;
```

Oefening 5.8:

```
SELECT Stagiaires.stagiairnr, Begeleiders.begeleidersnr, achternaam,
beg_uren, reistijd
FROM Instellingen INNER JOIN (Stages INNER JOIN (Begeleiders INNER JOIN
Stagiaires
ON Begeleiders.begeleidersnr = Stagiaires.begeleidersnr) ON Stages.stagenr
= Stagiaires.stagenr)
ON Instellingen.instellingsnr = Stages.instellingsnr
WHERE (Begeleiders.begeleidersnr)=3;
```

Oefening 5.9:

SELECT achternaam, SUM(geb_uren), SUM(reistijd)
FROM [Overzicht begeleider Galama]
GROUP BY achternaam;

Oefening 5.15:

```
SELECT Stagiaires.stagiairnr, Studenten.studentnr, Stages.stagenr,
achternaam, woonplaats, Instellingen.instellingsnr, naam, plaats,
Instellingen.telefoon, [type instelling], contactpersoon, reistijd
FROM Studenten INNER JOIN ((Instellingen INNER JOIN Stages ON
Instellingen.instellingsnr = Stages.instellingsnr) INNER JOIN Stagiaires ON
Stages.stagenr = Stagiaires.stagenr)
ON Studenten.studentnr = Stagiaires.studentnr
ORDER BY Stagiaires.stagiairnr;
```

4. De INSERT, UPDATE en DELETE statements

Het aanbrengen van wijzigingen aan één of meer tabellen valt uiteen in drie categorieën:

Je kan één of meer records toevoegen aan een tabel.

Je kan de waarden van één of meer velden voor één of meer records in een tabel wijzigen.

Je kan één of meer records verwijderen uit een tabel.

SQL heeft voor elk van deze bewerkingen een statement.

4.1 INSERT

Je kan één enkel record aan een bestaande tabel toevoegen met het statement

```
INSERT INTO tablename [(fieldname [, ...])] VALUES (value [, ...]);
```

Aan de tabel *tablename* worden de waarden *value, ...* toegevoegd in de velden *fieldname, ...* Dit komt overeen met het rechtstreeks intypen van nieuwe waarden in een tabel. Wanneer je geen veldnamen opgeeft, bepaalt de volgorde van de velden in *tablename* in welk veld de waarden worden opgenomen. Er moeten dan evenveel *values* zijn als velden in de tabel *tablename*.

Daarnaast heeft Access twee queries om records uit één of meerdere tabellen te selecteren en toe te voegen aan een nieuwe of bestaande tabel:

Met de **Make Table Query** bewaar je de dynaset van een SELECT query in een nieuwe tabel. De SQL syntaxis is:

```
SELECT fieldlist INTO tablename
FROM tablelist
WHERE condition
GROUP BY group_fieldlist
HAVING group_condition
ORDER BY fieldlist;
```

Het enige verschil met een gewone SELECT query wordt gevormd door de INTO *tablename* clause, waarin de naam van de nieuwe tabel wordt vermeld. De velden in deze nieuwe tabel zijn de velden uit de *fieldlist*. Je kan de velden in de nieuwe tabel een andere naam geven door *aliassen* te gebruiken in de *fieldlist*. De tabel *tablename* zal niet automatisch een primaire sleutel krijgen!

Met de **Append Query** voeg je de dynaset van een SELECT query toe aan een reeds bestaande tabel. De SQL syntaxis is:

```
INSERT INTO tablename [(fieldlist1)]
SELECT fieldlist2
FROM tablelist
WHERE condition
GROUP BY group_fieldlist
HAVING group_condition
ORDER BY fieldlist;
```

Het SELECT statement wordt voorafgegaan door de INSERT INTO clause, waarin je de tabel en de veldnamen in de tabel kan vermelden waarin de geselecteerde velden moeten worden geplaatst.

Indien *fieldlist1* ontbreekt, wordt aangenomen dat de velden in *tablename* dezelfde naam hebben als die in *fieldlist2*. Indien dit niet het geval is zal de volgorde van de velden in *tablename* en *fieldlist2* de overeenkomst bepalen.

Indien *fieldlist1* niet ontbreekt, wordt de overeenkomst van de velden bepaald door gelijke namen in *fieldlist1* en *fieldlist2*. Indien er geen overeenkomst is, is opnieuw de volgorde van de velden in *fieldlist1* en *fieldlist2* bepalend.

Steeds moet het aantal namen in *fieldlist1* (of *tablename* wanneer *fieldlist1* ontbreekt) en *fieldlist2* overeenkomen.

In het algemeen doe je er goed aan de namen én de volgorde van de velden in de dynaset van het SELECT statement en de INSERT INTO clause te laten overeenkomen. Velden waarvoor geen overeenkomst bestaat worden immers niet aan de tabel *tablename* toegevoegd.

Nieuwe records worden achteraan de bestaande tabel toegevoegd, in de volgorde bepaald door het SELECT statement. Wanneer de waarden van de primaire sleutel van reeds bestaande en nieuw toegevoegde records gelijk zijn, zullen deze nieuwe records niet worden toegevoegd. De primaire sleutel mag immers geen dubbele waarden aannemen.

De **Make Table** en **Append Queries** dupliceren gegevens. Dit is meestal niet gewenst vanuit het database standpunt, en wordt daarom niet erg vaak gedaan. Situaties waarbij dit wel zinvol is, zijn momentopnames die bewaard moeten worden, backups en tabellen bestemd voor gebruik door andere toepassingen, die niet met dynasets kunnen omgaan. Ook het opsplitsen van een tabel in twee nieuwe tabellen met een **one to one** relatie ertussen kan je met INSERT INTO uitvoeren. Je moet dan achteraf nog wel de primaire en vreemde sleutels definiëren.

Voorbeelden:

```
INSERT INTO Studenten (studentnr, achternaam, voornaam)
VALUES (25, Vervliet, Willy);
```

```
SELECT studentnr, achternaam, voornaam INTO Namen
FROM Studenten WHERE geb_datum <#1/1/75#;
```

```
INSERT INTO Namen
SELECT studentnr, achternaam, voornaam
FROM Studenten
WHERE geb_datum>=#1/1/75#;
```

4.2 UPDATE

Met een **Update Query** kan je de waarden van één of meer velden in een tabel wijzigen. De syntaxis van het SQL statement is:

```
UPDATE tablename SET fieldname = expression [, ...]
WHERE condition;
```

In de UPDATE clause geef je met *tablename* de te wijzigen tabel aan. Vervolgens vermeld je in de SET clause de te wijzigen velden en hun nieuwe waarden. Deze nieuwe waarden kunnen constanten (tekst of getallen) zijn, maar ook berekende formules. De records waarbij dit moet gebeuren kan je bepalen met een WHERE clause. Indien de WHERE clause ontbreekt worden de wijzigingen in alle records van de tabel aangebracht.

Voorbeelden:

```
UPDATE Studenten SET jaar = 1997, klas = 2KA;
```

```
UPDATE Begeleiders SET vergoeding = vergoeding + reistijd * 10
WHERE beschikbaar = Yes;
```

```
UPDATE Begeleiders SET vak = Null;
```

4.3 DELETE

Met een **Delete Query** kan je één of meer records uit een tabel verwijderen. De syntaxis van het SQL statement is:

```
DELETE [*] FROM tablename
WHERE condition;
```

In de DELETE clause kan je (facultatief) met de * aangeven dat je volledige records wilt verwijderen. In de FROM clause geef je de naam *tablename* van de tabel op. Welke records moeten worden verwijderd kan je bepalen met een WHERE clause. Indien de WHERE clause ontbreekt worden alle records verwijderd, en de tabel dus leeggemaakt.

Voorbeelden:

```
DELETE * FROM Stages;
```

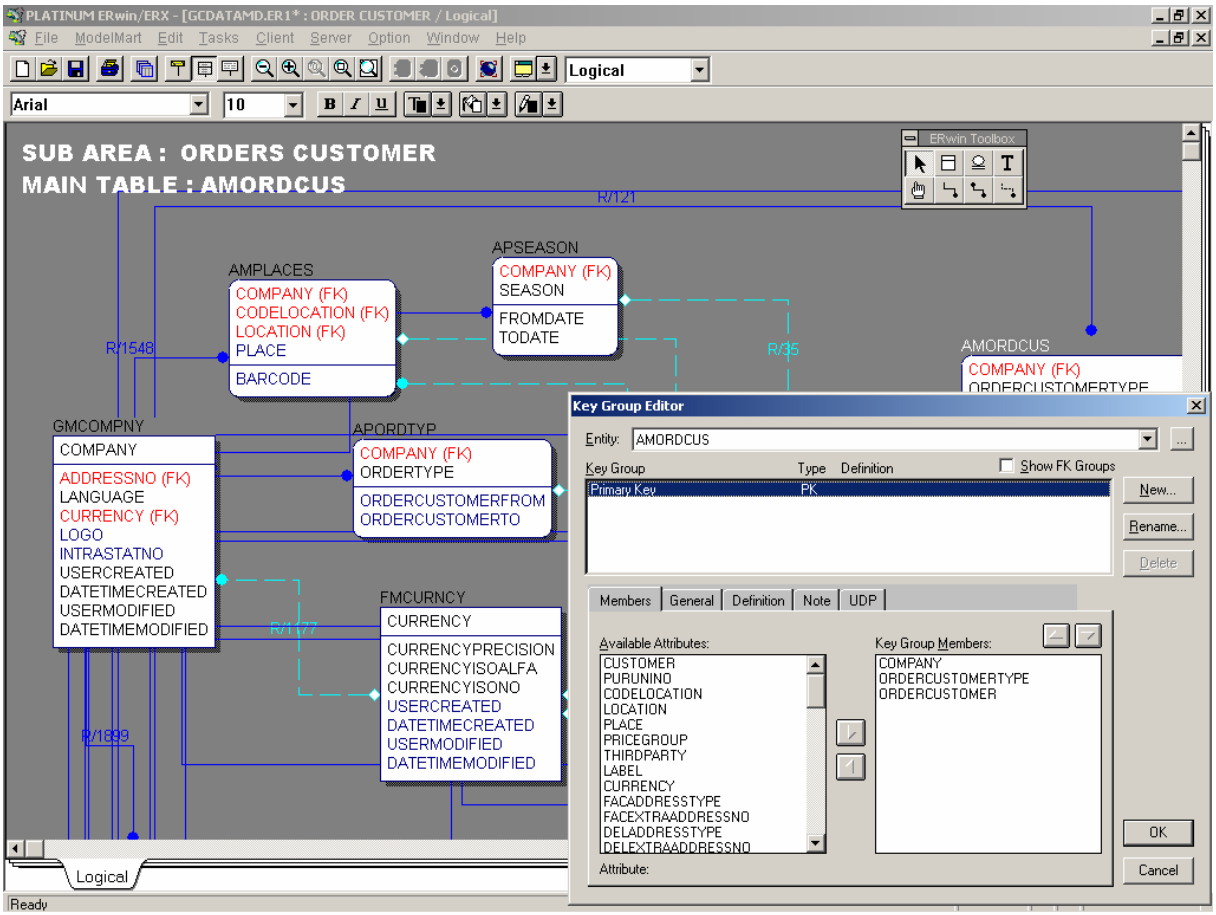
```
DELETE * FROM Studenten WHERE woonplaats = "Gent";
```

Enkele interessante database tools

Erwin

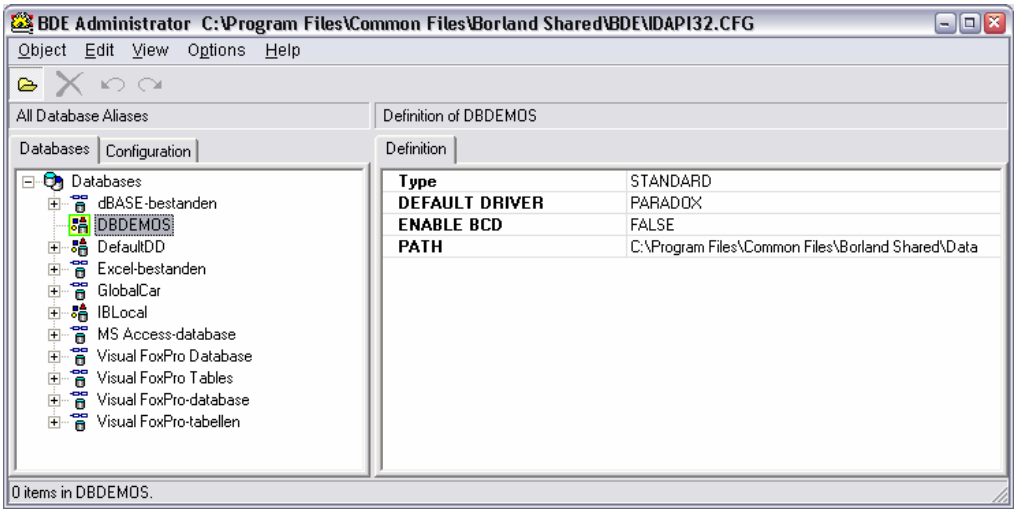
ERD tool voor het tekenen van een database model. Vanuit het model kunnen database scripts gegenereerd worden voor Oracle, Interbase, SQL Server, DB2, …

http://www3.ca.com/Solutions/Product.asp?ID=260



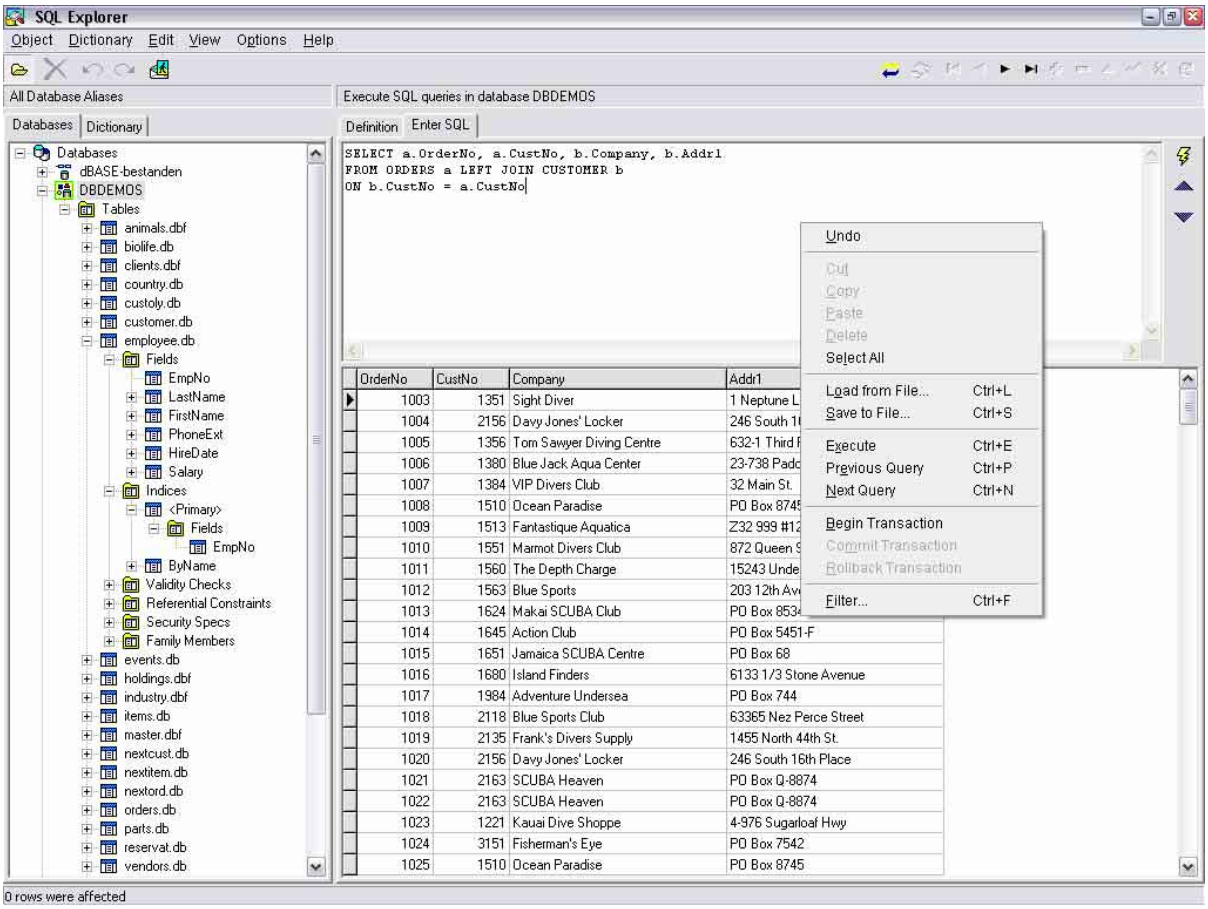
BDE Adminstrator

Met deze Borland tool kan je de aliases van de BDE beheren.



SQL Explorer

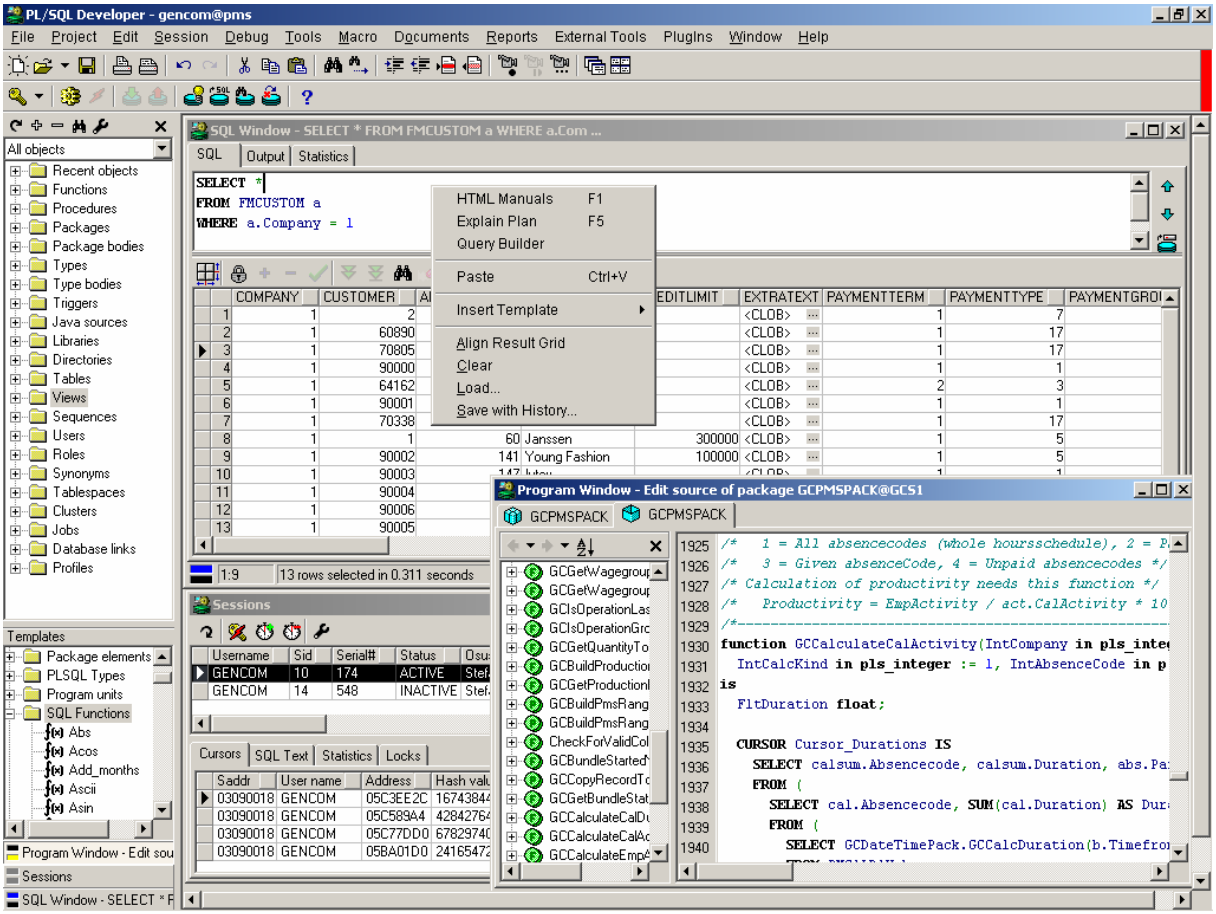
Deze eenvoudige Borland tool wordt geïnstalleerd wanneer je Delphi installeert. Met dit programma kan je alle databases aanspreken waarvoor een BDE alias is aangemaakt. Je kan de structuur van de tabellen bekijken en SQL's uitvoeren.



PL/SQL Developer

Zeer krachtige en gebruiksvriendelijke Windows tool voor het beheer en programmatie van een Oracle database.

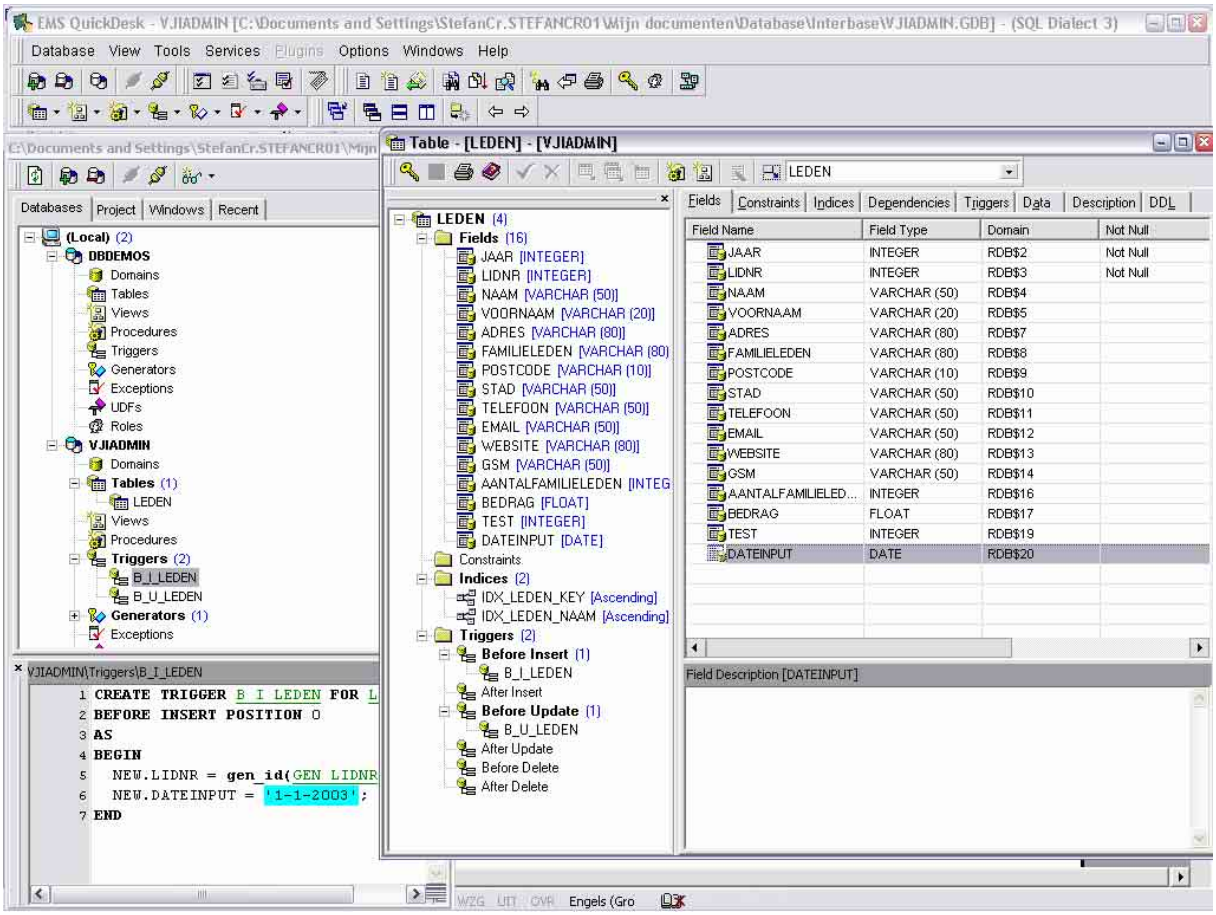
http://www.allroundautomations.com



EMS QuickDesk / IB Manager

Gebruiksvriendelijk en uitgebreide database tool voor Interbase en Firebird.

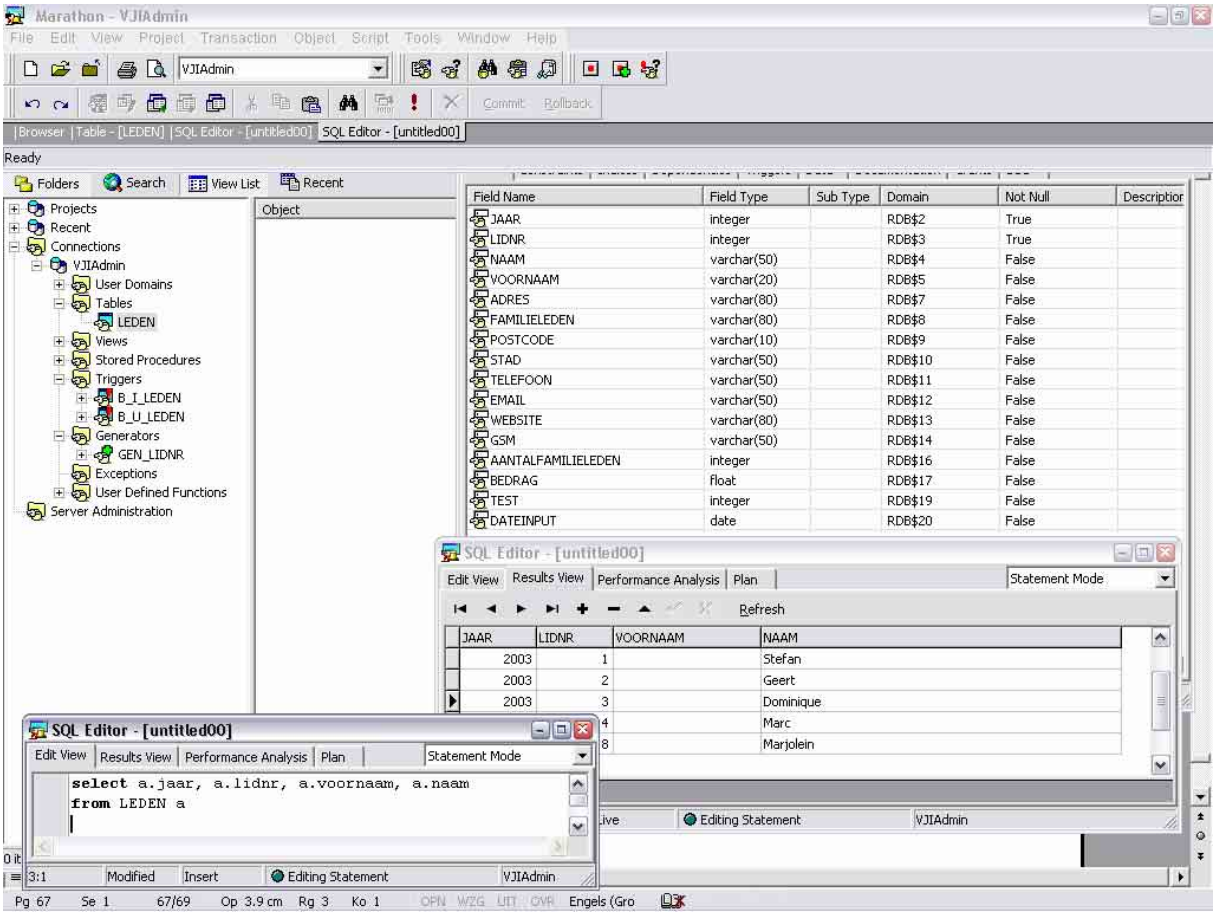
http://www.ems-hitech.com/ibmanager



Marathon

Interessante open source tool voor Interbase en Firebird. Dit programma kan gratis gedownload worden.

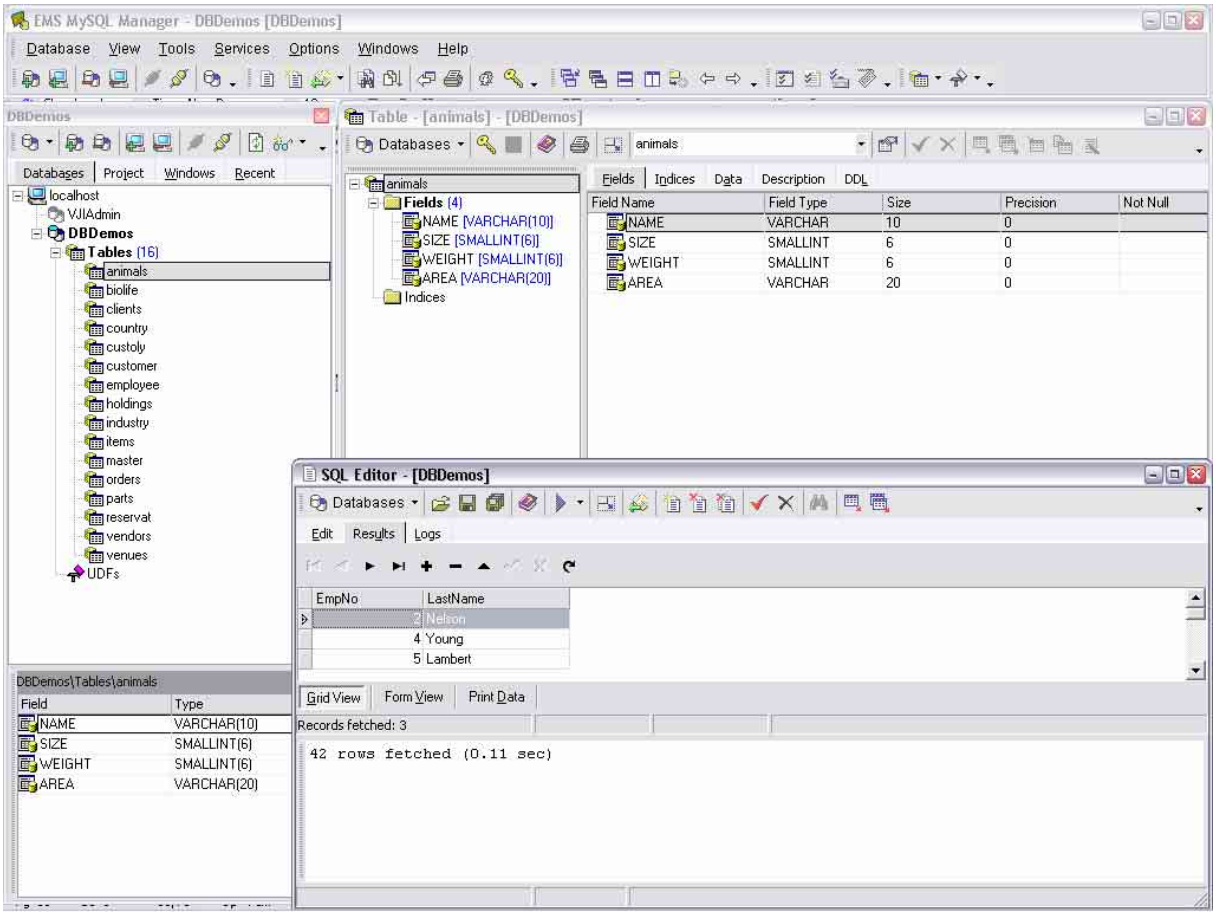
http://sourceforge.net/projects/gmarathon



EMS MySQL Manager

Knappe database tool voor MySQL. Beschikbaar voor Windows en Linux.

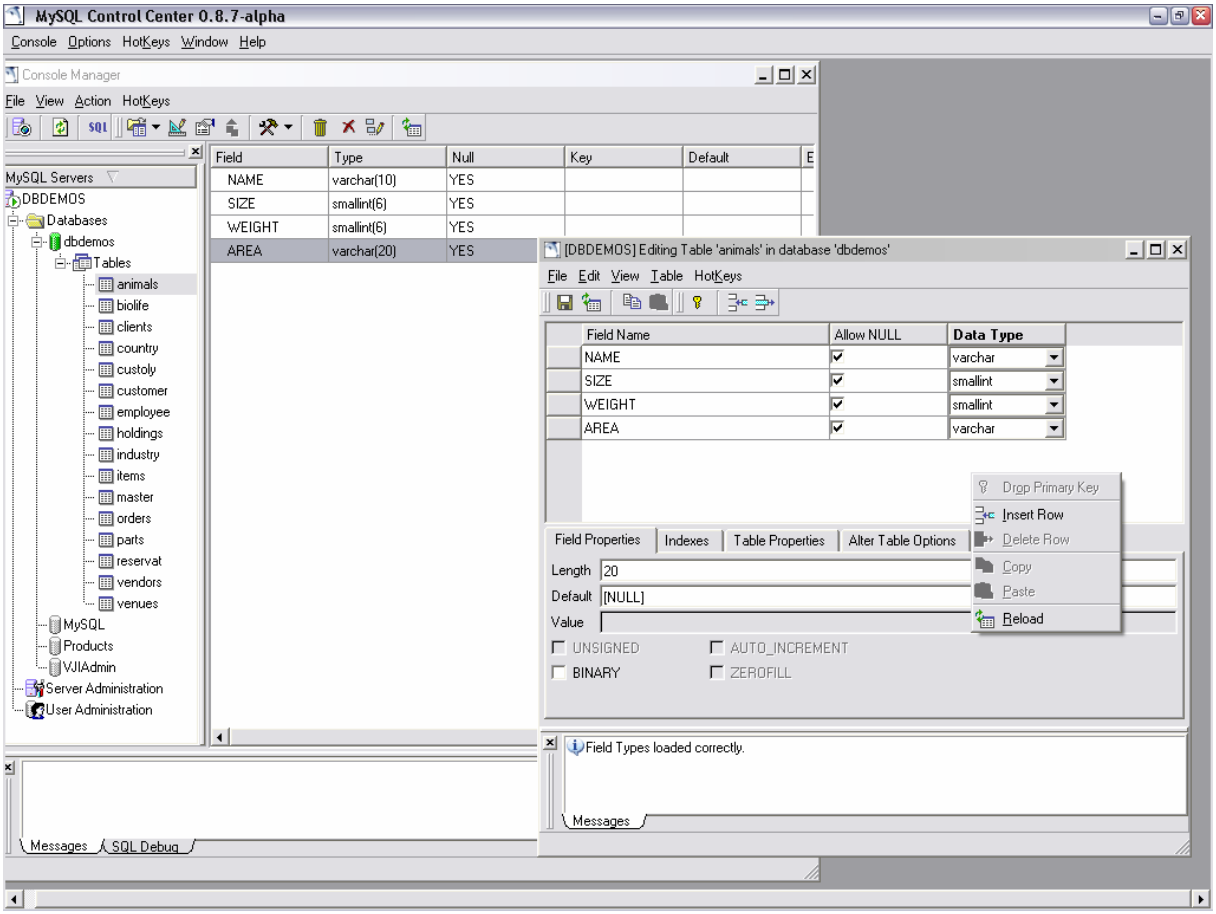
http://www.ems-hitech.com/mymanager



MySQL Control Center

Grafische open source tool van de ontwikkelaars van MySQL. Beschikbaar voor Windows en Linux.

http://www.mysql.com/downloads/mysqlcc.html



Delphi database componenten

Delphi (en Kylix en C++Builder) is een programmeertaal die zeer veel databases ondersteund. In de eerste versies van Delphi zat de BDE, achteraf zijn er allerlei nieuwe componenten voor nieuwe database engines aan toegevoegd.

BDE (Borland Database Engine)

Oudere Borland technologie die zeer veel databases zowel lokaal als op server ondersteunt en veel werd gebruikt in het verleden. Zal in de toekomst wel verdwijnen. Enkel beschikbaar onder Windows.

ADO (Microsoft ActiveX Data Objects)

Database engine van Microsoft die voornamelijk goed werkt bij Microsoft databases zoals Access, SQL Server en eventueel Excel. Enkel beschikbaar onder Windows.

IBX (InterBase Express)

Componenten speciaal voor een Interbase (en Firebird) databases. Standaard beschikbaar in Delphi en C++Builder, maar Kylix versies kunnen gedownload worden.

DBX (dbExpress)

Nieuwste platform onafhankelijke (voor Delphi en Kylix) database technologie van Borland die gebruikt kan worden voor de grotere server databases zoals Oracle, Interbase, DB2, MySQL en ook SQL Server.

Verder zijn er nog andere third party componenten beschikbaar zoals

DOA : Direct Oracle Access van AllRound Automations
http://www.allroundautomations.nl/doa.html

IBO : Interbase Objects
http://www.ibobjects.com

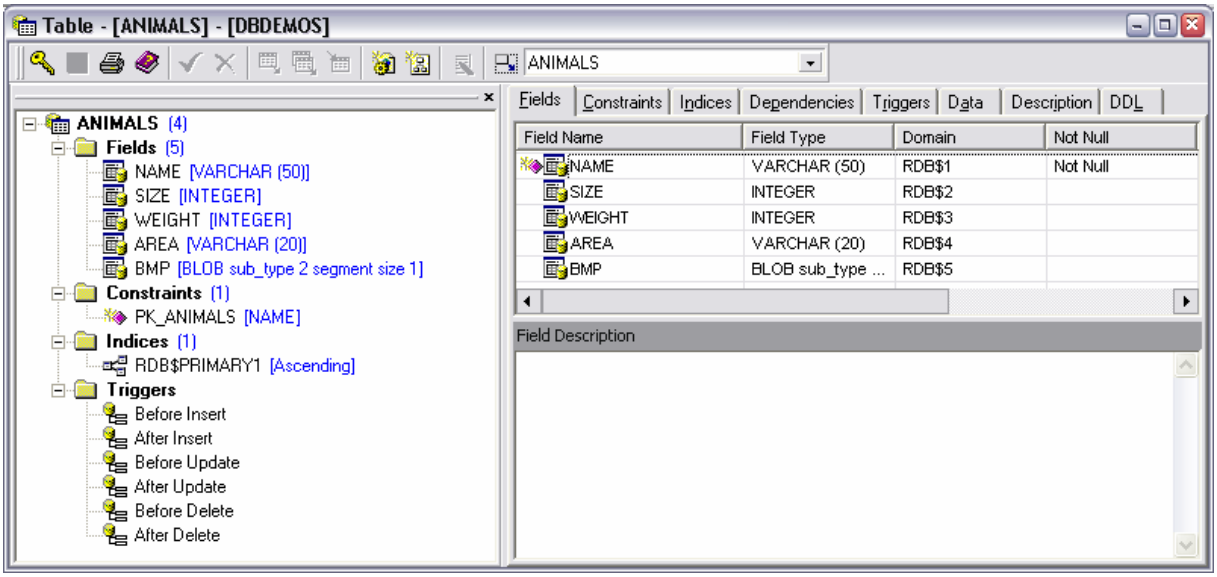
MySQLDAC
http://www.microolap.com/products/dac/mysqldac.htm

Structuur van DBDemos tabellen

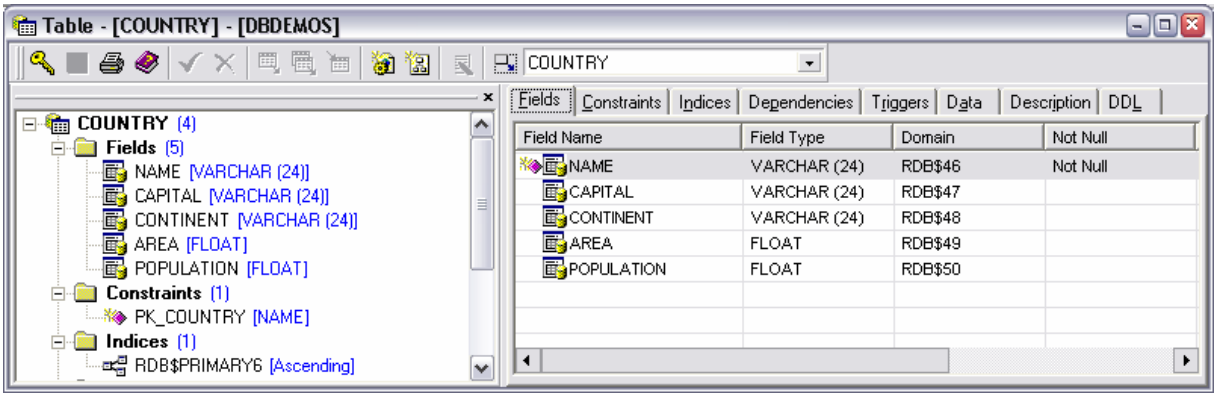
De DBDemos tabellen worden standaard meegeleverd met Borland producten. Het zijn allen eenvoudige Paradox en DBase tabellen. In de BDE wordt hiervoor een alias met de naam DBDemos aangemaakt.

Deze databases zijn echter ook geconverteerd naar MS Access, Firebird/Interbase en MySQL. Je kan ze downloaden op Stefan's Homepage. http://users.pandora.be/stefancr

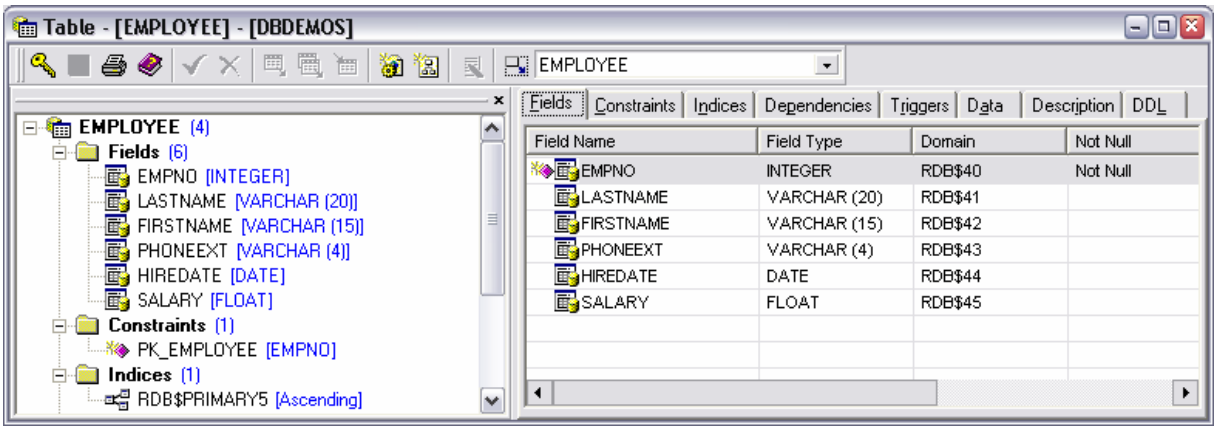
ANIMALS



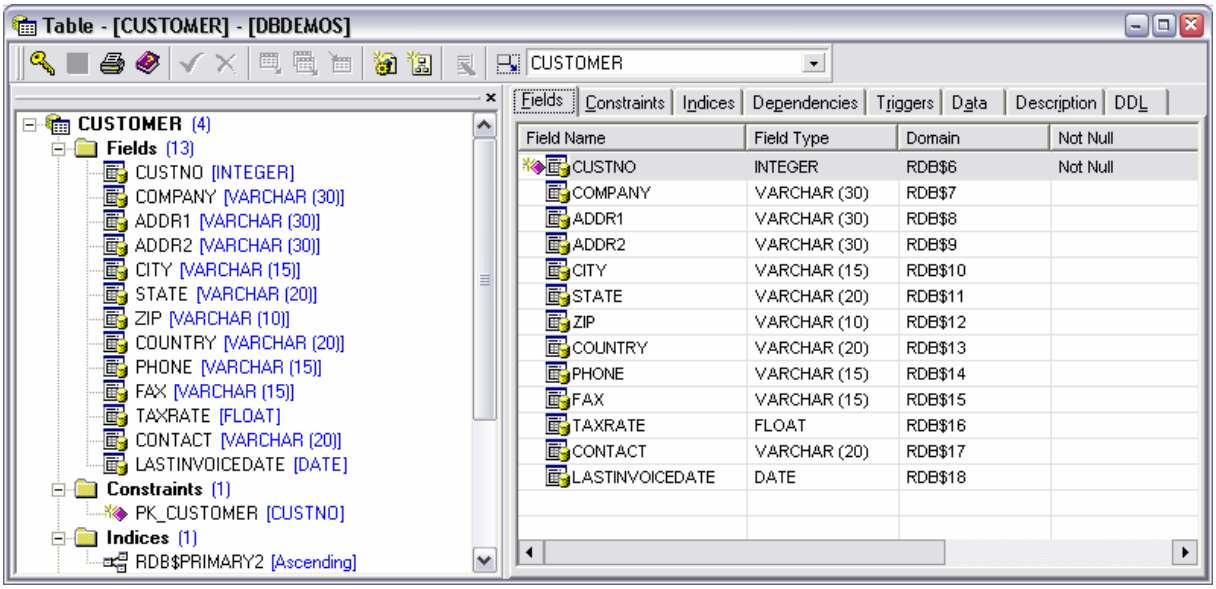
COUNTRY



EMPLOYEE



CUSTOMER



ORDERS

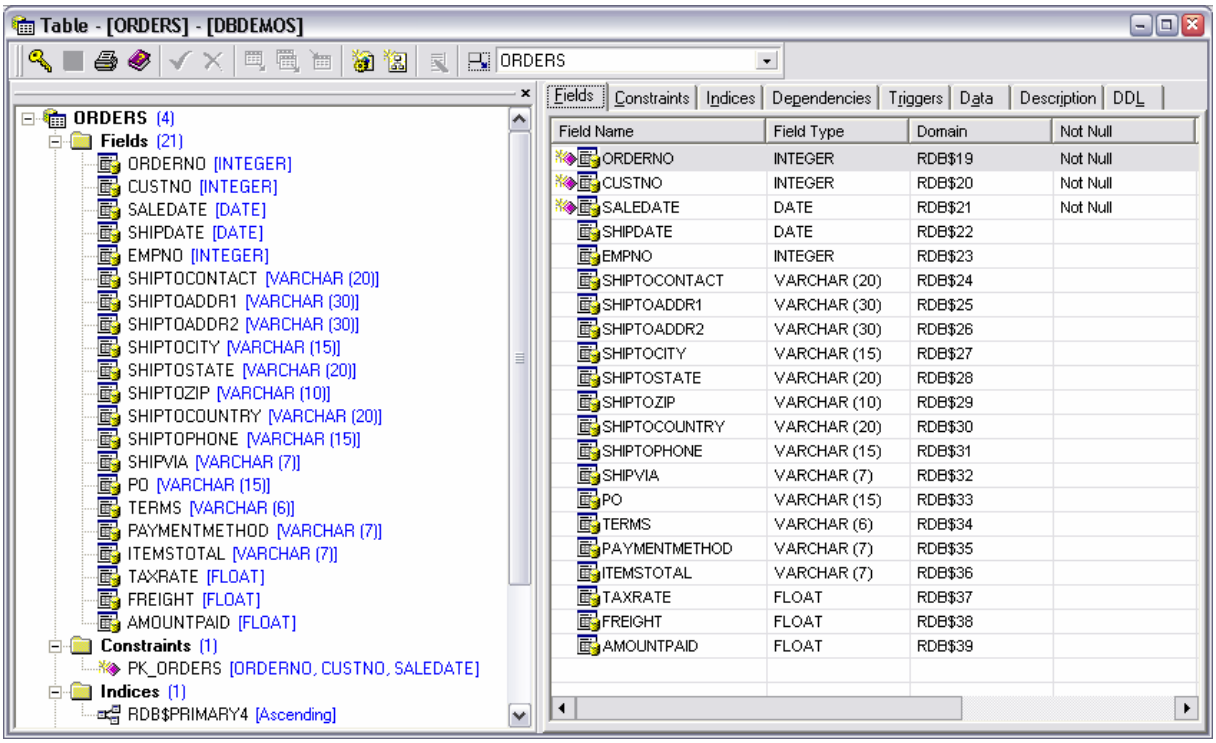


Table - [ORDERS] - [DBDEMOS]
ORDERS
ORDERS (4)
Fields (21)
ORDERNO [INTEGER]
CUSTNO [INTEGER]
SALEDATE [DATE]
SHIPDATE [DATE]
EMPNO [INTEGER]
SHIPTOCONTACT [VARCHAR (20)]
SHIPTOADDR1 [VARCHAR (30)]
SHIPTOADDR2 [VARCHAR (30)]
SHIPTOCITY [VARCHAR (15)]
SHIPTOSTATE [VARCHAR (20)]
SHIPTOZIP [VARCHAR (10)]
SHIPTOCOUNTRY [VARCHAR (20)]
SHIPTOPHONE [VARCHAR (15)]
SHIPVIA [VARCHAR (7)]
PO [VARCHAR (15)]
TERMS [VARCHAR (6)]
PAYMENTMETHOD [VARCHAR (7)]
ITEMSTOTAL [VARCHAR (7)]
TAXRATE [FLOAT]
FREIGHT [FLOAT]
AMOUNTPAID [FLOAT]
Constraints (1)
PK_ORDERS [ORDERNO, CUSTNO, SALEDATE]
Indices (1)
RDB$PRIMARY4 [Ascending]
Fields
Constraints
Indices
Dependencies
Triggers
Data
Description
DDL
Field Name
Field Type
Domain
Not Null
ORDERNO INTEGER RDB$19 Not Null
CUSTNO INTEGER RDB$20 Not Null
SALEDATE DATE RDB$21 Not Null
SHIPDATE DATE RDB$22
EMPNO INTEGER RDB$23
SHIPTOCONTACT VARCHAR (20) RDB$24
SHIPTOADDR1 VARCHAR (30) RDB$25
SHIPTOADDR2 VARCHAR (30) RDB$26
SHIPTOCITY VARCHAR (15) RDB$27
SHIPTOSTATE VARCHAR (20) RDB$28
SHIPTOZIP VARCHAR (10) RDB$29
SHIPTOCOUNTRY VARCHAR (20) RDB$30
SHIPTOPHONE VARCHAR (15) RDB$31
SHIPVIA VARCHAR (7) RDB$32
PO VARCHAR (15) RDB$33
TERMS VARCHAR (6) RDB$34
PAYMENTMETHOD VARCHAR (7) RDB$35
ITEMSTOTAL VARCHAR (7) RDB$36
TAXRATE FLOAT RDB$37
FREIGHT FLOAT RDB$38
AMOUNTPAID FLOAT RDB$39

Online cursussen

Cursus Microsoft Access (Nederlands)
http://www.sip.be/cursus/access/inhoud.htm

Cursus Installatie Microsoft SQL Server (Nederlands)
http://www.sip.be/cursus/sql7/sql.htm

Beknopte uitleg database ontwerp, optimalizaties, … (Nederlands)
http://www.sum-it.nl/cursus/index.php3

Uitgebreide cursussen SQL (Engels)
http://www.sqlcourse.com
http://www.w3schools.com/sql/default.asp

Firebird Quick Start Guide (Engels)
http://www.ibphoenix.com/downloads/qsg.pdf

MySQL Reference (Engels)
http://www.mysql.com/documentation/mysql/alternate.html

Oracle8 SQL Reference (Engels)
http://www-rohan.sdsu.edu/doc/oracle/server803/A54647_01/toc.htm