Databanken

Hogeschool Gent
Departement Industriële Wetenschappen
Vakgroep Informatica
Academiejaar 2004 - 2005

Agenda

Theorie

Les

duur : 13 weken, 2 uur
wanneer : vrijdag 8:15 - 10:15
waar : lokaal 408

geen theorie op 12/11

Quotatie

impact : 200 punten
vorm : mondeling examen
wanneer : 1e periode 1e zittijd

Labo

Les

duur : 13 weken, 2 x 1,5 uur
wanneer : maandag en vrijdag
waar : lokaal 225

geen labo op 1/11 12/11 en 15/11

Quotatie

impact : 225 punten
vorm : test/project persoonlijk werk
wanneer : test op 29/11 project laatste 4/5 weken

Richtlijnen

Er is geen oefeningenexamen, ook niet in 2e zit!

Er is één inhaaltest over ALLE stof tijdens de examenperiode ENKEL voor de gewettigde (uiterlijk op 1/12) afwezigen!

Iedereen heeft z'n resultaten van ALLE labo's ELK labo bij en kan die ter controle van het dagelijks werk voorleggen!

Vragen

Dirk.Vandycke@hogent.be

Ann.Vanoverberge@hogent.be

Bronnen

An introduction to Database Systems, Volume I (6th Ed.)

ISBN 0-201-82458-2

C.J. Date

Database Systems (3th Ed)

ISBN 0-201-70857-4

Oracle

- Installatie CD-rom's kunnen gekopieerd worden
- Het gebruik van oracle software door studenten valt onder onze OAI-licentie

WWW

- Oracle Technology Network

  http://www.oracle.com/pls/db92/db92.homepage
- Een login kun je zelf aanmaken of je gebruikt *student@hogent.be* met als paswoord *oracle*

Doelstellingen

De doelstellingen van de cursus databanken zijn:

- In staat zijn een goede databank te ontwerpen
- Deze te implementeren via SQL in Oracle
- Het verwerven van een gedegen kennis SQL (vooral, maar niet alleen, het select statement)
- In staat zijn een programmeertaal met embedded SQL te gebruiken (hiervoor gebruiken wij PL SQL aan boord van Oracle) en het maken van triggers en stored procedures als toepassing hierop
- Een perfect inzicht in het relationeel model en z'n denkpatronen, waarvan we ook de wiskundige grondslagen ons eigen maken.

Het is **niet** de bedoeling experts te worden in het beheer van een specifieke DBMS in de tijd waarover we beschikken en waar vooral ervaring z'n werk moet doen.

SQL Plus

- SQL Plus is een client die ons in staat stelt met Oracle te communiceren. Dit kan via:
  - SQL

    een sql-statement termineren met ';' voert het direct uit
  - opdrachten eigen aan deze client

    edit : toont de sql-buffer (het huidige/laatste sql-statement)
    dit laat ondermeer toe oefeningen te corrigeren/bewaren!

    run of /<enter> : voert de sql-buffer (herhaaldelijk) uit

    start : voert de sql-statements in een bestand uit
    hiermee kun je de gegeven scripts uitvoeren
- Er bestaan andere populaire clients, een voorbeeld van een zeer algemeen (niet enkel voor Oracle) bruikbare client is TOAD

Oefeningen

- Vergeet niet de Oracle-server voor je databank manueel op te starten!

  Als je databank *lll* noemt dan is de naam van de service *OracleServicelll*

- Inloggen op Oracle

  account : system

  paswoord : oracle

Terminologie

DB	Databank of gegevensbank	Database
DBMS	Databank beheersysteem (soms DB Beheerder)	Database Management System (soms DB Manager)
DBA	Databank beheerder	Database Administrator
DA	Databeheerder	Data Administrator

Informatie = gegevens + betekenis

Procesgegevens = afgeleide gegevens

DBMS : Bestanddelen

Architectuur (ANSI-SPARC 3 level arch.)

- Fysisch niveau (bestandstoegang en –beheer)
  - hashing, pointerlijsten, random access …
- Logisch niveau (hoe ziet data er conceptueel uit)

Beheer

- Implementatie schema's
- Backups
- Logging en auditing
- Beveiliging
- Performantie monitoring
- Business logic

Analyse en ontwerp databanken

DBMS : Taken

Query processor
Query optimizer
Database manager
File manager
Integriteitscontrole
Recovery manager (backup, herstel)
Transaction manager
Compiler (preprocessor, command processor)
Catalog manager (metadata, schemabeschrijving)
Scheduler (planner)
Toegangscontrole
Geheugenbeheer

DBMS : Voordelen

1) Redundantie daalt : '1 fact in 1 place'
2) Verhouding informatie/gegevens neemt toe
3) Consistentie stijgt
4) Integriteit makkelijker te waarborgen
5) Gedeelde en gedistribueerde systemen mogelijk (multi-user vs. single-user)
6) Beveiliging eenvoudiger te implementeren
7) Data-onafhankelijkheid door scheiding fysisch en logisch ontwerp
8) Betere toegang en toegangstijden

DBMS : Nadelen

1) Complexiteit (alhoewel, vooral op fysisch niveau dan)
2) Omvang (is dit zo?)
3) Kost (aanschaf, onderhoud, hardware, opleiding, …)
4) Performantie daalt (t.o.v. flat-file systemen)
   - Door normalisatie
   - Dit is op te vangen door fysische optimalisatie
5) Kwetsbaarder als centraal systeem
   - Op te vangen door gedistribueerde DBMS

DBMS : Ontstaan

Ontstaan vanuit de nood aan beheersysteem voor databanken (vergelijkbaar met ontstaan besturingssystemen)

Oorspronkelijk : experimentele modellen

- netwerkmodel
- inverted list model
- hiërarchisch model

Deze modellen stonden dicht bij fysisch ontwerp en bestandsbeheer

DBMS : Evolutie

Relationeel model
- Belangrijkste (r)evolutie in databankwereld
- Eerste model op solide basis (verzamelingenleer)
- Momenteel nog steeds meest gebruikte model
- Blijft waarschijnlijk nog een tijdje zo!

Nieuwe modellen (experimenteel)
- OO-DBMS
- Fuzzy (vage) DBMS
- Temporele en historische DBMS
- XML?

Relationeel model

Logisch model uit een paper van E. F. Codd (1970)

1) Bekijkt alle gegevens enkel en alleen als tabellen (verzameling van records)

2) Alle bewerkingen gebeuren op tabellen (en resulteren eventueel in tabellen)

DB praktijk	DB theorie	programmeren	wiskunde	OOP
tabel	tabel	tabel/array/bestand	verzameling	type/klasse
rij	tuple	record	element	variabele/object
kolom	attribuut	veld	eigenschap	member

cardinaliteit = aantal rijen

graad = aantal kolommen

Relationele DMBS'en

Product	Producent
Oracle 9i	Oracle
SQL Server	Microsoft
mySQL	
DB2	IBM
Access	Microsoft (desktop!)

Anderen : Informix, Postgres, Ingres, …

SQL

- Ontwikkeld door IBM, beschouwd als de de-facto standaard
- Gestandardiseerd (ANSI en ISO) in SQL92 en SQL99

  Een implementatie van de standaard noemen we een dialect
- WAT niet HOE (in tegenstelling tot 3GL's)
- Gesteund op relationele algebra
- Is een DSL (data sublanguage) bestaande uit

  DML : Data manipulatie taal
  DDL : Data definitie taal
  DCL : Data controle taal
- Kan 'embedded' worden binnen hostlanguage (PL/SQL, T-SQL, Jet-SQL, …) (impedance mismatch!)
- Dynamische SQL
- Niet case-sensitive

SQL

DDL

data definitie

operaties

- CREATE
- ALTER
- DROP

objecten

- DATABASE
- TABLE
- INDEX
- VIEW

DML

data manipulatie

INSERT
gegevens toevoegen
UPDATE
gegevens wijzigen
SELECT
gegevens opvragen
DELETE
gegevens verwijderen

DCL

data controle

access control

- GRANT
- REVOKE

transaction control

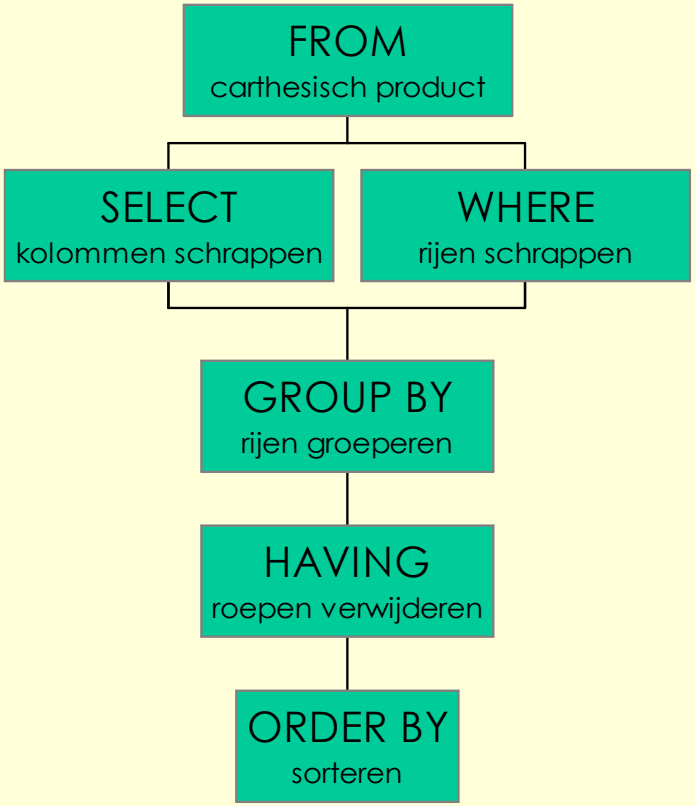
- COMMIT
- ROLLBACK
- SAVEPOINT

system control

session control

SELECT statement

CLOSED OPERATION



Syntax

```
SELECT    ...
FROM      ...
WHERE     ...
GROUP BY  ...
HAVING    ...
ORDER BY  ...
```

Voorbeeld

kleuren

kleurnummer	kleur
100	Geel
200	Blauw
300	Groen

```
SELECT   kleur
FROM     kleuren
WHERE    kleurnummer = 200
```

SELECT

kleurnummer	kleur
~~100~~	~~Geel~~
200	Blauw
~~300~~	~~Groen~~

WHERE

FROM clausule (VERPLICHT)

- Syntax

  FROM <*tabelnaam*> [*alias*] [, <*tabelnaam*> [*alias*]] [, …]

- Maakt het carthesisch product van alle tabellen die worden opgesomd
- Tabel-aliassen zijn bruikbaar in gans de opdracht

  Naast korte(re) schrijfwijze, vooral van belang bij multitable queries
- In Oracle is dual de ledige tabel

  Dit maakt het mogelijk om enkel het resultaat van uitdrukkingen te bekijken via SQL vb. SELECT *SYSDATE* FROM dual
- Een tabel kan meerdere keren voorkomen in de from-clausule (met verschillende aliassen!)

  Maakt carthesisch product van een tabel met zichzelf

SELECT clausule (VERPLICHT)

- Laat toe om kolommen te schrappen uit resultaat
- Syntax

  SELECT [DISTINCT | ALL] <*uitdrukking*> [AS *alias*] [, <*uitdrukking*> [AS *alias*]] [, …]
- Kolom-aliassen zijn zichtbaar in gans de opdracht (dialecten!)

  voor ontkoppeling van kolomnamen en voor berekende velden
- DISTINCT elimineert dubbele rijen (zijn die er?)

  - slaat op gans de select (niet enkel de 1ste kolom)

  - schrapt rijen en hoort dus logisch gezien bij de WHERE thuis
- ALL toont alle rijen (ook de dubbele)

  ALL hoef je nooit te schrijven, het is de standaardkeuze

  Niet te verwarren met het predicaat ALL (zie later)
- Een asterisk (*) staat voor alle kolommen (<> een statische opsomming van alle kolommen)

SELECT clausule (VERVOLG)

- Uitdrukkingen wordt gemaakt op basis van

  1/ literals (letterlijke waarden/constanten)

  - getalwaarden genoteerd zoals in C++ vb. –12, 23.05
  - strings genoteerd tussen enkele quotes vb. ‘een string’

    OPM: In de meeste systemen gelden ook de dubbele quotes “ en ”, maar niet noodzakelijk als synoniem voor de enkele quotes!

  2/ kolommen (of aliassen)

  - meest algemene vorm

    *<tabelnaam | alias>.<kolomnaam | alias | *>*
  - de tabelprefix hoeft meestal niet geschreven te worden

  3/Rekenkundige operatoren

  - operatoren * / + - (ook unair) en haakjes (zoals in C++)
  - concatenatieoperator platformafh. vb. | | , & , +, …

  4/ Functies

  Zijn ingebouwd en platformafhankelijk (niet gestandardiseerd)

- *Afgeleide* of *berekende* kolommen noemen we *procesdata* en worden best van een alias voorzien

Voorbeeld

producten

naam	prijs	korting
computer	1500	10
printer	300	15

SELECT naam, prijs*(1-korting/100) AS nettoprijs
FROM producten

naam	nettoprijs
computer	1350
printer	255

WHERE (NIET VERPLICHT)

- Laat toe om rijen te schrappen
- Syntax

  WHERE <*logische uitdrukking*>
- De uitdrukking wordt voor elke rij geëvalueerd

  indien waar wordt de rij getoond, anders niet
- Rekenkundige operatoren

  * / + - (ook unair)
- Relationele operatoren

  > = < <> <= >=
- Logische operatoren

  AND OR en NOT (XOR geen standaard, in principe niet nodig)

Voorbeeld

producten

naam	prijs	korting
computer	1500	10
printer	300	15

SELECT naam, prijs*(1-korting/100) AS nettoprijs
FROM producten
WHERE 250 < nettoprijs AND korting < 15

naam	nettoprijs
computer	1350
~~printer~~	~~255~~

WHERE (VERVOLG - PREDICATEN)

- Range test (niet strikt nodig)

  <*uitdrukking*> [NOT] BETWEEN <*waarde*> AND <*waarde*>

  grenzen inclusief!

- Set membership (niet strikt nodig)

  <*uitdrukking*> [NOT] IN (<*waarde*>[, <*waarde*>[, …]])

  behoort tot, is element van

- Pattern matching

  <uitdrukking> [NOT] LIKE '*patroon*' ESCAPE '*escape-karakter*'

  wildcards of jokertekens : _ juist 1 karakter (?)

  (dialecten!!!) % 0 of meer karakters (*)

- NULL test (3–value logic in SQL)

  <*uitdrukking*> IS [NOT] NULL

  niet met vergelijkingsoperator = !!!

- TOP n en TOP n% (geen standaard)

  niet gebruiken om een waarde te zoeken in combinatie met order by

Voorbeeld

producten

naam	prijs	korting
computer	1500	10%
printer	300	15%

SELECT naam, prijs*(1-korting/100) AS nettoprijs
FROM producten
WHERE korting NOT LIKE '15#%' ESCAPE '#'

naam	nettoprijs
computer	1350
printer	255

Voorbeeld

producten

naam	prijs	korting
computer	1500	<null>
printer	300	15%

SELECT naam, prijs*(1-korting/100) AS nettoprijs
FROM producten
WHERE korting IS NOT NULL AND prijs IS NOT NULL

naam	nettoprijs
computer	<null>
printer	345

ORDER BY (NIET VERPLICHT)

- Laat toe om de resulterende rijen te ordenen

  deze clausule wordt dus altijd als allerlaatste uitgevoerd
- Syntax

  ORDER BY <*uitdrukking*|*nummer*> [ASC|DESC][, <*uitdrukking*|*nummer*> [ASC|DESC]][, …]
- ASC ordent oplopend, DESC dalend

  ASC wordt, als standaardwaarde, meestal niet geschreven

  LET OP met de volgorde, 2<10 MAAR '10'<'2'
- Kolommen die geen naam of alias hebben kunnen met hun volgnummer aangeduid worden (te vermijden!)

  - het nummer is het volgnummer in de select-clausule (niet het volgnummer in de oorspronkelijke tabel)

  - de eerste kolom heeft nummer 1, …

ORDER BY (VERVOLG)

- Kolommen hoeven niet in de select te staan

  - hoewel sommige dialecten dit eisen

  - kolommen die niet in de select staan kunnen uiteraard NIET met een volgnummer aangeduid worden

- Er wordt gesorteerd van links naar rechts

  sorteren op de eerste kolom, gelijke waarden sorteren op de volgende kolom, enz …

- NULL waarden

  zijn groter of kleiner dan alle andere waarden (afh. van de implementatie)

Voorbeeld

producten

naam	prijs	korting
printer	300	15
computer	1475	20
computer	1500	15

SELECT naam, korting
FROM producten
ORDER BY naam, prijs DESC

naam	korting
computer	15
computer	20
printer	15

AGGREGAATFUNCTIES (1)

- Standaard aggregaatfuncties

  MIN MAX AVG SUM en COUNT (=aantal)
- SYNTAX

  <*aggregaatfunctie*> ([DISTINCT | ALL] <*uitdrukking*>)
- Enkel in de select (en having) clausule
- Deze functies werken op 1 (afgeleide) kolom
- Berekenen totalen voor een groep (rijen)

  - keren dus 1 waarde terug, berekend over gans de groep
- Zonder group by

  - er is maar 1 groep, die alle rijen bevat

  - groepen wordt herleid tot 1 rij in het resultaat, m.a.w. hier worden alle rijen herleid naar 1 rij

  - in de SELECT kunnen kolommen ENKEL binnen aggregaatfuncties voorkomen, van zodra er 1 aggregaatfunctie gebruikt wordt (men zegt ook dat elk onderdeel van de select **single-valued** moet zijn)

AGGREGAATFUNCTIES (VERVOLG)

- COUNT(*) telt alle rijen
- SUM en AVG enkel voor numerieke kolommen
- Null waarden worden niet in rekening gebracht
  met uitzondering van COUNT(*), die alle **rijen** telt (speciaal geval)
- Combinatie met DISTINCT mogelijk
  - distinct staat binnen de aggregaatfunctie!
  - elimineert dubbele **waarden** vóór de functie wordt uitgerekend
  - heeft geen effect op MIN en MAX
  - de standaard zegt dat DISTINCT maar 1 keer zou mogen voorkomen in dezelfde query (dialecten spreken dit tegen)
- Aggregaatfuncties kunnen NIET genest worden

Voorbeeld

producten

naam	prijs	korting
printer	300	15
computer	1475	20
computer	1500	15

SELECT COUNT(DISTINCT naam) AS productaantal,
SUM(prijs*(1-korting/100)) AS totaal

FROM producten

productaantal	totaal
2	2710

= 300*0.85 + 1475*0.8 + 1500*0.85

Voorbeeld

producten

naam	prijs	korting
printer	300	15
computer	1475	20
computer	1500	15

SELECT naam, SUM(prijs*(1-korting/100)) AS totaal, SYSDATE AS datum
FROM producten

productaantal	totaal	datum
printer	2710	08/10/02
computer	2710	08/10/02
computer	2710	08/10/02

Carthesisch product

- Zoals vroeger reeds aangehaald wordt **eerst** een **carthesisch product** gemaakt van de tabellen in de FROM clausule (de komma is dus quasi een operator)
- Het carthesisch product van 2 verzamelingen is …

  … de verzameling van alle koppels elementen van beide verzamelingen

  $A \times B = \{(a_i,b_j) \mid a_i \in A, b_j \in B\}$ (= A,B in SQL met komma als x-operator)
- Rekenregels

  $n_r(A \times B) = n_r(A) \times n_r(B)$

  $n_k(A \times B) = n_k(A) + n_k(B)$
- Het carthesisch product van n verzamelingen is …
- Levert meestal veel zinloze combinaties (meestal!)

  - oplossing : zinloze rijen schrappen met where

  - dit levert oude stijl join-expressies met '=' (zie later)

  - n tabellen veronderstelt n-1 join-expressies

Voorbeeld

producten

naam	prijs	categorie
printer	300	1
computer	1475	2
computer	1500	2

categorieën

nr	naam
1	printers
2	computers

producten.naam	prijs	categorie	nr	categorieën.naam
printer	300	1	1	printers
computer	1475	2	1	printers
computer	1500	2	1	printers
printer	300	1	2	computers
computer	1475	2	2	computers
computer	1500	2	2	computers

SELECT *

FROM producten, categorieën **WHERE** categorie = nr

Voorbeeld

objecten

naam	kleurnummer
Lucht	200
Gras	300

X

kleuren

kleurnummer	kleur
100	Geel
200	Blauw
300	Groen

SELECT *
FROM objecten, kleuren

WHERE objecten.kleurnummer = kleuren.kleurnummer

naam	objecten. kleurnummer	kleuren.kleurnummer	kleur
~~Lucht~~	~~200~~	~~100~~	~~Geel~~
Lucht	200	200	Blauw
~~Lucht~~	~~200~~	~~300~~	~~Groen~~
~~Gras~~	~~300~~	~~100~~	~~Geel~~
~~Gras~~	~~300~~	~~200~~	~~Blauw~~
Gras	300	300	Groen

JOINS

- JOIN

  Algemene term voor het samenvoegen van data uit verschillende tabellen
- Verschillende gradaties

  Gerangschikt naar restrictiviteit (allemaal deelverzameling van de hogere)

cross-join

carthesisch product (geen restricities)

(hoewel standaard wordt meestal de komma gebruikt)

θ -join

cross-join met restrictie operator θ

equi-join

θ-join waarbij θ de operator = is

natural join

equi-join op gelijknamige kolommen (waarvan er slechts 1 getoond wordt)

Voorbeeld

producten

ID	naam	prijs
1	printer	300
2	computer	1475
3	computer	1500

θ Is hier '>'

P1.naam	P1.prijs	P1.ID	P2.ID	P2.naam	P1.prijs
~~printer~~	~~300~~	~~1~~	~~1~~	~~printer~~	~~300~~
computer	1475	2	1	printer	300
computer	1500	3	1	printer	300
~~printer~~	~~300~~	~~1~~	~~2~~	~~computer~~	~~1475~~
~~computer~~	~~1475~~	~~2~~	~~2~~	~~computer~~	~~1475~~
computer	1500	3	2	computer	1475
~~printer~~	~~300~~	~~1~~	~~3~~	~~computer~~	~~1500~~
~~computer~~	~~1475~~	~~2~~	~~3~~	~~computer~~	~~1500~~
~~computer~~	~~1500~~	~~3~~	~~3~~	~~computer~~	~~1500~~

SELECT P1.naam, P1.prijs, P2.naam, P2.prijs
FROM producten P1, producten P2 **WHERE** P1.ID > P2.ID

Voorbeeld

producten

pnaam	prijs	categorienr
printer	300	1
computer	1475	2
computer	1500	2

categorieën

categorienr	cnaam
1	printers
2	computers

producten.pnaam	prijs	categorienr	categorieën.cnaam
printer	300	1	printers
computer	1475	2	computers
computer	1500	2	computers

SELECT *

FROM producten NATURAL JOIN categorieën

JOINS (VERVOLG)

- OUTER JOIN

  - combineert ALLE rijen langs 1 zijde met z'n tegenhanger langs de andere zijde (of met null waarden)

  - drie vormen : left, right en full outer joins

  - een voorbeeld is hier nodig

  - kan soms dmv speciale operator in de where (gebruik af te raden)

- SELF JOIN

  theoretische term voor een tabel die met zichzelf wordt samengebracht

- INNER JOIN

  - term voor een gewone join, bedoeld om het verschil te accentueren met een outer-join

  - dit is niet zonder meer een theoretische term, sommige dialecten laten de syntax INNER JOIN toe als synoniem voor de syntax JOIN

JOINS (VERVOLG)

- Oude stijl

  met join-conditie in de where (onafh. van dialecten)
- Nieuwe stijl

  in de from genoteerd
- SYNTAX

  FROM <*tabel*> [CROSS |
  INNER |
  NATURAL |
  LEFT [OUTER] |
  RIGHT [OUTER] |
  FULL [OUTER]] JOIN <*tabel*>
  [ON <*kolom*> = <*kolom*>]

  deze syntax is zeer gevoelig voor dialecten!!!
- Deze syntax kan genest worden (haakjes gebruiken!)

Voorbeeld

producten

naam	prijs	categorie
printer	300	1
computer	1475	2
computer	1500	3

categorieën

nr	naam
1	printers
2	computers

producten.naam	prijs	categorie	nr	categorieën.naam
printer	300	1	1	printers
computer	1475	2	2	computers
computer	1500	3	<null>	<null>

SELECT *

FROM producten LEFT (OUTER) JOIN categorieën ON categorie = nr

Voorbeeld

Uit het labo :
Geef alle titels van boeken die een korting geven op een boek dat (zonder korting) meer dan € 100 kost

```
SELECT    kopen.titel
FROM      boeken kopen, boeken op, promoties
WHERE     promoties.kopen = kopen.ID
AND       promoties.op = op.ID
AND       op.prijs > 100
```

Dit is geen self-join!

TIPS

- Meestal worden niet alle tabellen, nodig voor het oplossen van een vraag, in de vraag vermeld.

  - toch moet je alle noodzakelijke tabellen in de join-ketting erbij betrekken

  - N tabellen betekent meestal N-1 join-uitdrukkingen

- Opmaak van queries (2 mogelijkheden)

  - vertrek van wat je hebt en laat onderweg vallen wat je niet nodig hebt

  - vertrek van wat je wilt en betrek onderweg erbij wat je nodig hebt

  - vertrek van een voorbeeldoplossing (query by example of QBE)

GROUP BY (NIET VERPLICHT)

- Laat toe om groepen te maken
- SYNTAX

  GROUP BY <*uitdrukking*>[, <*uitdrukking*>][, …]
- Rijen met dezelfde waarde voor de uitdrukking(en) horen bij dezelfde groep
- Alle rijen uit een groep worden vervangen door 1 rij in het resultaat (werkwoord 'to condense')

  - bijgevolg heeft het geen zin meer om rij-specifieke zaken op te gaan vragen via de select, van zodra group by aanwezig is (single-valued)

  - m.a.w. vooral gebruik van aggregaatfuncties
- Gebruik van aggregaatfuncties

  subtotalen per groep
- Men spreekt van een *grouped* query en van *grouping* columns

GROUP BY (VERVOLG)

- Integratie met select clausule

  - zoals reeds gesteld moet elk onderdeel van de select single-valued zijn (per groep!)

  - alle kolommen die in de select voorkomen moeten bijgevolg in de group by voorkomen, tenzij ze binnen een aggregaatfunctie gebruikt worden (niet omgekeerd!)

  - uiteraard zijn uitdrukkingen zonder verwijzingen naar kolommen niet aan dergelijke regel onderhevig

- NULL waarden

  - worden als gelijk beschouwd in de group by

  - er is dus ook de groep NULL

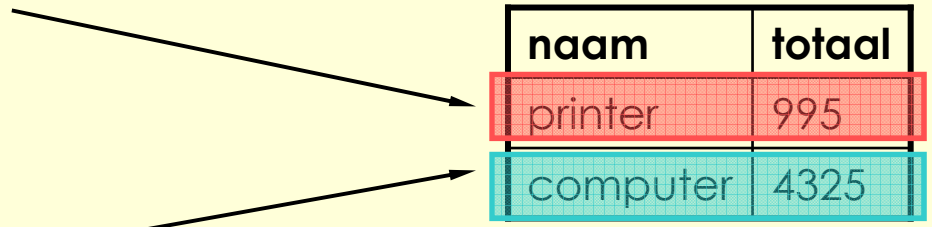
- Meerdere uitdrukkingen in de group by

  over elke unieke combinatie van alle kolommen in de group by wordt een groep gemaakt

Voorbeeld

producten

naam	prijs	categorie
printer	300	1
printer	320	1
printer	375	2
computer	1350	<null>
computer	1475	<null>
computer	1500	3



naam	totaal
printer	995
computer	4325

SELECT naam, SUM(prijs) AS totaal~~, prijs~~

FROM producten

GROUP BY naam

Voorbeeld

producten

naam	prijs	categorie
printer	300	1
printer	320	1
printer	375	2
computer	1350	<null>
computer	1475	<null>
computer	1500	3

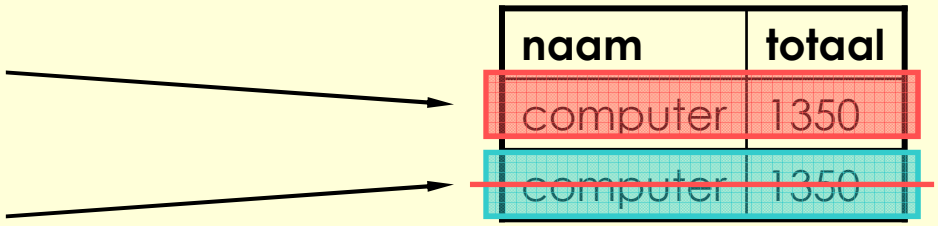
naam	totaal
printer	620
printer	375
computer	2825
computer	1500

SELECT naam, SUM(prijs) AS totaal
FROM producten
WHERE categorie IS NOT null
GROUP BY naam, categorie

Voorbeeld

producten

naam	prijs	categorie
computer	1350	<null>
computer	1350	<null>
computer	750	3
computer	600	3



naam	totaal
computer	1350
computer	1350

SELECT DISTINCT naam, SUM(DISTINCT prijs) AS totaal
FROM producten
GROUP BY naam, categorie

HAVING (NIET VERPLICHT)

- Laat toe om groepen te schrappen
- SYNTAX

  HAVING *<logische uitdrukking>*
- De uitdrukking wordt voor elke groep geëvalueerd

  indien 'waar' wordt de groep getoond, anders niet
- Aggregaatfuncties

  - laten toe groepen te evaluëren op totalen

  - in de praktijk is minstens 1 aggregaatfunctie wenselijk (anders kan gewoon where gebruikt worden!)
- HAVING enkel bij GROUP BY mogelijk

  having is voor groepen, wat where is voor rijen

  strikt genomen kan zelfs bewezen worden dat having nooit nodig is
- Kolommen in de having **moeten** opgenomen worden in de group by **ofwel** in een aggregaatfuntie

  m.a.w. tenzij ze in een aggregaatfunctie zijn opgenomen

Voorbeeld

producten

naam	prijs	categorie
printer	300	1
printer	320	1
printer	375	2
computer	1350	<null>
computer	1475	<null>
computer	1500	3

naam	Totaal
printer	620
printer	375
computer	2825
computer	1500

SELECT naam, SUM(prijs) AS totaal
FROM producten
WHERE categorie IS NOT null
GROUP BY naam, categorie
HAVING COUNT(*)>1

SUBQUERIES

- Definitie

  Een select-query binnen een andere query (deze hoeft geen select te zijn, tenzij hij op zijn beurt een subquery is)
- Synoniemen : geneste queries of subselects

  dit kan op meer dan 1 niveau (subqueries in subqueries, enz …)
- Men spreekt van **inner** (de subquery) en **outer** queries (de omvattende queries)

  niet te verwarrren met inner en outer joins
- Order by niet toegelaten

  tenzij in de buitenste (outermost) query
- Toegelaten in where en having

  hoewel scalaire subqueries (zie later) ook in de select kunnen voorkomen en ook de from soms toegelaten wordt (tabel)subqueries te bevatten (in bepaalde dialecten)
- Subquery altijd als rechteroperand

SUBQUERIES (VERVOLG)

- Een subquery kan al dan niet **gelinkt** zijn aan elk van de omvattende queries
- Dit gebeurt door te verwijzen naar kolommen/tabellen/aliassen uit deze omvattende queries

  indien nodig voorafgegaan door qualifiers
- Zijn strikt genomen niet nodig

  sommigen implementeren ze zelfs niet wegens inperformant
- Worden tussen ronde haakjes geplaatst
- Gelinkte subqueries zijn altijd duurder dan niet gelinkte

  gelinkte worden immers telkens opnieuw uitgevoerd (voor elke rij van de hoofdquery)

SCALAIRE SUBQUERIES

- Leveren 1 waarde op

  of een resultaat van maximaal 1 rij en 1 kolom

- Kunnen overal gebruikt worden waar een scalaire waarde verwacht wordt

  - dit ondermeer in de select, where, …

  - dialecten kunnen dit beperken!

  - soms eist een dialect dat een scalaire waarde verzekerd is. Een gewone select kan verboden worden, ook al levert hij toevallig één waarde op. Aggregaatfuncties en unieke kolommen verzekeren 1 waarde.

- Gebruik

  - meestal als waarde in een expressie

(slechte) Voorbeelden

SELECT naam, prijs – ~~AVG(prijs)~~
FROM producten
WHERE prijs > ~~AVG(prijs)~~

SELECT naam, prijs – (**SELECT** AVG(prijs) **FROM** producten)
FROM producten
WHERE ~~(**SELECT** AVG(prijs) **FROM** producten)~~ < prijs

Voorbeelden

```
SELECT  naam
FROM    producten
WHERE   categorie =     (       2       )
```

```
SELECT  naam, prijs     –       (       1091.67         )
FROM    producten
WHERE   prijs           >       (       1091.67         )
```

RIJ/TABEL SUBQUERIES

- Rij subqueries leveren één rij op
  een resultaat met maximaal 1 rij, mogelijks meerdere kolommen
- Tabel subqueries leveren een tabel op
  meerdere rijen en kolommen mogelijk
- Kunnen niet langer overal gebruikt worden
- Predicaat [NOT] EXISTS (tabel(sub)queries)
  - gaat na of een subquery resultaten oplevert (vergelijkt met ∅)
  - 'er bestaat (g)een enkele …' (∃)
- Predicaat [NOT] IN (vereist een tabelquery met 1 kolom!)
  - gaat na of een waarde in een query teruggevonden wordt
  - 'is element' van operator (∈ en ∉)
- Predicaten ANY en ALL (vereist een tabelquery met 1 kolom!)
  - 'er bestaat …' (∃) (ISO stelt ook SOME voor als synoniem voor ANY)
  - 'voor alle …' (∀)
  - worden gecombineerd met een (relationele) operator

Voorbeelden

```
SELECT titel
FROM   boeken
WHERE  ID IN   ( SELECT   boek, betaald
                 FROM     aankopen
                 WHERE    lidnummer IN ( SELECT   lidnummer
                                         FROM     leden
                                         WHERE    woonplaats IN ( 122, 345 )))
SELECT naam
FROM   producten
WHERE  prijs  >  ALL  ( SELECT     prijs
                        FROM       producten
                        WHERE      categorie = 2
                        ORDER BY   prijs )
AND    categorie = 1
```

Voorbeelden

```
SELECT titel,    ( SELECT   SUM (prijs)
                   FROM     aankoopdetails
                   WHERE    boek = boeken.ID  )  AS totaal
FROM   boeken

SELECT naam
FROM   leden L
WHERE  NOT EXISTS    ( SELECT     *
                       FROM       aankoopdetails, aankopen
                       WHERE      lidnummer = L. lidnummer
                       AND        aankoop = aankopen.ID
                       AND        boek = 1256)
```

Voorbeelden

```
SELECT  naam
FROM    aankopen A, leden
WHERE   NOT EXISTS    ( SELECT     *
                        FROM       boeken B
                        WHERE      NOT EXISTS ( SELECT  *
                                                FROM    aankoopdetails
                                                WHERE   boek = B.ID
                                                AND     A.ID = aankoop ))
AND     A.lidnummer = leden.lidnummer
```

Geef alle leden die in één aankoop alle boeken hebben gekocht

Voorbeelden

```
SELECT  naam
FROM    leden
WHERE   NOT EXISTS   ( SELECT    *
                       FROM      boeken B
                       WHERE     NOT EXISTS ( SELECT  *
                                              FROM    aankoopdetails,
                                                      aankopen
                                              WHERE   boek = B.ID
                                              AND     A.ID = aankoop
                                              AND     A.lidnummer =
                                                      leden.lidnummer ))
```

Geef alle leden die alle boeken al eens hebben gekocht (niet in 1 aankoop daarom)

SET OPERATORS

- Werken in op verzamelingen van records
  - lijmen dus (het resultaat van) verschillende queries aan elkaar
  - deze moeten zogenaamd '**union-compatible**' zijn
  - dit betekent zelfde aantal kolommen met identieke datatypes en lengtes
  - het predicaat ALL laat dubbele rijen toe
- INTERSECT [ALL]

  doorsnede (∩)
- EXCEPT [ALL]
  - verschil (/ of -)
  - soms genaamd MINUS (dialecten!)
- UNION [ALL]

  -unie (∪)
  - in praktisch alle dialecten ondersteund (in tegensteling tot except en intersect)
- CORRESPONDING [BY <kolom>[, <kolom>][, …]]

  voert de set-operator uit op opgegeven of gelijknamige kolommen

Voorbeelden

```
(
        SELECT *
        FROM   woonplaatsen WP, werknemers WN
        WHERE  WN.woonplaats = WP.ID
)
UNION   CORRESPONDING BY naam
(
        SELECT *
        FROM   woonplaatsen WP, klanten K
        WHERE  K.woonplaats = WP.ID
)
```

SQL (nabeschouwingen)

- Veelzijdig karakter van de SELECT

  hoge aantal redundante oplossingen voor eenzelfde vraag wordt vaak als nadeel beschouwd van SQL
- Joins zijn duur

  Neem dus geen onnodige tabellen mee
- Subqueries

  - kunnen in bepaalde gevallen tot meer duidelijke queries leiden, die beter de vraag representeren

  - mogen echter geen automatisme worden

  - vooral gelinkte subqueries zijn (vaak onnodig) duur
- Predicaten

  zijn doorgaans duurder dan gewone relationele uitdrukkingen en joins

INSERT (DML)

- Laten toe rijen in te voeren

  tenzij bij views zal een insert altijd op 1 basistabel betrekking hebben
- Syntax

  INSERT INTO *tabel* [<*kolom*>[, <*kolom*>][, …]]

  {VALUES (<*waarde*> [, <*waarde*>][, …]) | SELECT-statement}
- Kolomlijst is optioneel

  - in dat geval wordt de volgorde van de kolommen bij creatie verondersteld

  - we raden dit gebruik af gezien de definitie van een tabel in het relationeel model (kolomvolgorde onbepaald) (schrijf dus altijd de kolomlijst)

  - is nodig indien ergens standaardwaarden (**defaults**) worden verondersteld
- Waardenlijst

  - een waarde voor elke kolom in de kolomlijst (desnoods de literal **null**)

  - sommige kolommen hebben geen standaardwaarde

  - sommige kolommen mogen geen null-waarden bevatten

  - sommige (combinaties van) kolommen moeten uniek zijn

UPDATE (DML)

- Laat toe om rijen aan te passen
- Syntax

  UPDATE <*tabel*>
  SET <*kolom*> = <*uitdrukking*>[, <*kolom*> = <*uitdrukking*>][, …]
  [WHERE <*logische uitdrukking*>]

- Werking

  - zonder where-clausule worden alle rijen aangepast

  - anders worden enkel die rijen aangepast die aan de where-conditie voldoen

  - alle kolommen worden aangepast met de (nieuwe) waarde, het resultaat van de uitdrukking

  - het is zonder meer mogelijk de waarden van dezelfde en andere kolommen te gebruiken in de opbouw van de uitdrukking

- Opmerking

  Het is vaak geen slechte zaak om voorafgaand een select * uit te voeren met dezelfde where-conditie, zeker in een real-life situatie

DELETE (DML)

- Laat toe om rijen te verwijderen
- Syntax

  DELETE FROM <*tabel*>

  [WHERE <*logische uitdrukking*>]
- Werking

  - zonder where-clausule worden alle rijen gewist

  - anders worden enkel die rijen gewist die aan de where-conditie voldoen
- Opmerking

  Het is vaak geen slechte zaak om voorafgaand een select * uit te voeren met dezelfde where-conditie, zeker in een real-life situatie

Voorbeelden

INSERT INTO producten (prijs, naam) **VALUES** (1250, ‘computer’)

```
INSERT INTO  producten ( prijs, naam )
SELECT       prijs*2 , naam
FROM         producten
WHERE        prijs > 1000

UPDATE       producten
SET          prijs = prijs * 1.05
WHERE        categorie = 2

DELETE
FROM         producten
WHERE        naam LIKE ‘p%’
```

QBE

- Query by Example
- Een grafische techniek waarbij men de gebruiker afschermt voor de tekstgebaseerde SQL
- Men ‘tekent’ een voorbeeld van wat men wil
- Voor meer complexe constructies kan men het gebruik van SQL niet vermijden
- Wordt uiteraard vertaald naar SQL

Relationeel model

- Voorgesteld door E. F. Codd in 1970
- Gebaseerd op verzamelingenleer en predicatenlogica
- Basisobjectieven
  - Data-onafhankelijkheid
    - volledige abstractie van de interne datastructurering
    - op die manier worden applicaties onafh. van interne opslag
    - laat een setgebaseerde universele manipulatietaal toe (SQL)
  - Redundantie vermijden en consistentie verbeteren

    redundantie = zelfde informatie op meer dan 1 plaats

    consistentie = er zijn geen contradicties in de informatie

    normalisatie is de naam van het proces met dit doel (zie later)
  - Formalisering van data-semantiek
    - niet alleen data maar ook de betekenis capteren
    - ter herinnering : informatie = data + betekenis

Relationeel model (VERVOLG)

- Uitgangspunt
  - Alle gegevens worden conceptueel voorgesteld als tabellen (relaties) en enkel en alleen als tabellen (relaties)
  - Het relationeel model manifesteert zich dus op conceptueel en extern niveau, maar stelt geen eisen aan de interne opslag (fysisch of intern niveau)
  - Principieel is er geen verschil tussen het begrip **relatie** en het begrip **tabel**, hoewel men nogal eens een tabel durft stellen als de fysieke (2D-)voorstelling van een relatie d.m.v. rijen en kolommen, wat informeel betekent

    relatie = tabel van rijen en kolommen
- Kernbegrip in het relationeel model is dus tabel of relatie

Relationele tabel

- Onderdelen (definities)

  tabel = een benoemde verzameling van tuples

  attribuut = een benoemde kolom (+domein!)

  tuple = een rij of element

  domein = verzameling (toegelaten) waarden (voor een attribuut)

  graad = aantal attributen

  cardinaliteit = aantal tuples

- Een relationele tabel is dus een benoemde verzameling tuples met gelijke attributen, elk met een unieke naam (binnen deze tabel)

  gelijke attributen ≠ gelijke attribuutwaarden
  = zelfde naam en domein

Relationele tabel (VERVOLG)

- Nog enkele definities

  intensie van een tabel = de beschrijving

  (m.a.w. de attributen)

  extensie van een tabel = de feitelijke rijen

  cel = snijpunt kolom met rij

  relationele databank = benoemde verzameling van (genormaliseerde) relaties met een unieke naam (binnen deze databank)

  men noemt een relatie of tabel **unair**, **binair**, **ternair** of **n-air** naargelang z'n graad

  quaternair, enz … worden meestal niet gebruikt (vanaf graad 4 : n-air)

- Graad en intensie zullen normaal dus minder frequent wijzigen dan cardinaliteit en extensie

Relationele tabel (VERVOLG)

- Eigenschappen (deels uit de definitie 'verzameling')

  1/ alle *tuples* in een relatie zijn *uniek*

  2/ alle *cellen* bevatten juist 1 *atomaire* (ondeelbare) *waarde*

  3/ *attributen* hebben (conceptueel) *geen volgorde*

  4/ *tuples* hebben (conceptueel) *geen volgorde*

  5/ de waarden van een attribuut behoren tot éénzelfde domein

  6/ Een *relatie* heeft een *unieke naam* (binnen de db)

  7/ Elk *attribuut* heeft een *unieke naam* (binnen de relatie)

Eigenschap 1

1/ alle *tuples* in een relatie zijn *uniek*

- In een verzameling zijn alle elementen uniek
- Wordt in een RDBMS gewaarborgd door ‘sleutels’ (zie later)

Het meervoud sleutels slaat hier op meerdere verschillende types

- De meeste DBMS-en laten dubbele rijen helaas wel toe

Dit is vooral uit praktische overwegingen om tijdelijke resultaten te kunnen materializeren maar is regelrecht tegenstrijdig met het relationeel model!

- Dit mag niet verkeerdelijk verward worden met dubbele rijen die eventueel resulteren uit een select-query

Hoewel de rijen in een tabel uniek zijn, hoeven de rijen in een query-resultaat dat niet te zijn

Eigenschap 2

2/ alle *cellen* bevatten juist 1 *atomaire* (ondeelbare) *waarde*

- ≠ Een element is het atoom van een verzameling!

Deze eis is er niet in verzamelingenleer (op elementeigenschappen)

- Wordt meestal probleemloos afgedwongen door elk RDBMS door de 2D-voorstelling van een tabel

Helaas kan hier altijd tegen worden gezondigd bij zeer generieke types (zoals karakter –en stringtypes, binaire data, …) als hierbinnen betekenis wordt gegeven aan een indeling (dergelijk systeem werkt schijnbaar goed door de aanwezigheid van ingebouwde functie en klassieke procedurele programmastructuren)

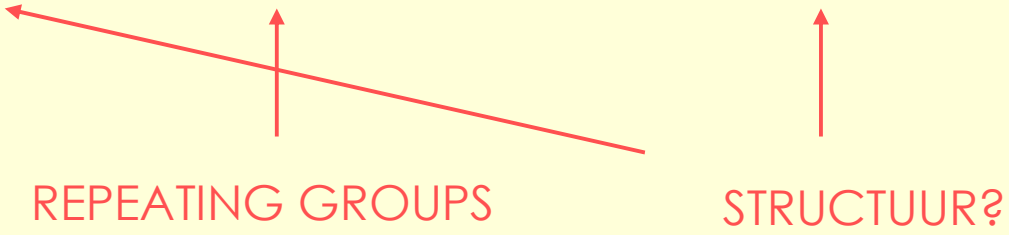
- Dit komt meteen neer op de eerste normaalvorm

Afgekort als 1NV of 1NF (first normal form)

- Deze regel kan omgekeerd ook leiden tot zeer ver doorgedreven opsplitsing (tot in het absurde)

Voorbeeld

Naam	Groepen	Woonplaats
Petronius Arbiter	1;2;3	Tervurenplein 7, 9000 Gent
Fox Mulder	1;3	Monacoplein, Antwerpen
E. F. Codd		8400 0ostende, E. Vlietinckstraat 34



Eigenschap 3

3/ *attributen* hebben (conceptueel) *geen volgorde*

- Fysisch uiteraard wel

En de fysische structuur kan repercussies hebben op de performantie

- Elementen in een verzameling hebben eigenschappen

Maar elementen (rijen) zijn de atomen van een verzameling

- Ga nooit uit van een eventuele volgorde!
- Het plaatswisselen van attributen geeft dus opnieuw dezelfde relatie

Eigenschap 4

4/ tuples hebben (conceptueel) *geen volgorde*

- Elementen in een verzameling hebben geen volgorde
- Fysisch uiteraard wel

En de fysische structuur kan repercussies hebben op de performantie

- Ga nooit uit van een eventuele volgorde!

Eigenschap 5

5/ de waarden van een attribuut behoren tot éénzelfde domein

- Een domein is een verzameling waarden

En een verzameling kan dan weer als een tabel vertaald worden

- Praktisch
  - Ingebouwde (meestal enkelvoudige) datatypes (ook standaard)
  - Een tabel defineert ook een verzameling waarden of objecten
  - De mogelijkheid eigen datatypes te definiëren is vaak niet of slechts gedeeltelijk ondersteund maar is altijd op te vangen door het definiëren van een nieuwe tabel
  - Samengestelde datatypes stroken eigenlijk niet echt met eigenschap 2 maar zijn soms aanwezig, wat dan eigenlijk impliceert dat we met een OO-DBMS of OO-RDBMS te maken hebben

Eigenschap 6

6/ Een *relatie* heeft een *unieke naam* (binnen de db)

- Deze eigenschap is vrij vanzelfsprekend

twee verschillende variabelen met dezelfde naam zijn immers niet van elkaar te onderscheiden

- Deze naam hoeft niet uniek te zijn over verschillende databanken

praktisch biedt in dat geval qualificatie m.b.v. de db-naam een oplossing, doch relationeel is er geen verband tussen verschillende databanken

- Bepaalde regels voor naamgeving kunnen door een dialect worden opgelegd

- Hoewel dialecten bepaalde leestekens toelaten in de naam is het gebruik af te raden

Voorbeelden zijn de spatie en het koppelteken (spatie bvb verboden in ISO)

Eigenschap 7

7/ Elk *attribuut* heeft een *unieke naam* (binnen de relatie)

- Deze eigenschap is vrij vanzelfsprekend

Twee verschillende variabelen met dezelfde naam zijn niet van elkaar te onderscheiden

- We weten echter vanuit SQL dat deze naam niet uniek hoeft te zijn binnen de databank

In dat geval biedt qualificatie m.b.v. de tabelnaam een oplossing

- Bepaalde regels voor naamgeving kunnen door een dialect worden opgelegd

- Hoewel dialecten bepaalde leestekens toelaten in de naam is het gebruik af te raden

Voorbeelden zijn de spatie en het koppelteken

Datatypes

- Datatypes zijn enkelvoudig
  in een zuiver relationele omgeving
- Datatypes zijn meestal ingebouwd
- Datatypes kunnen ingedeeld worden in
  - Alfanumerieke
  - Numerieke
    - Vlottende komma (floating point)
    - Vaste komma (fixed point)
    - Gehele
  - Booleaanse
  - Datum/tijd
  - Binaire (BLOB = binary large object)
- Datatypes kunnen een lengte en/of precisie hebben
- Taak : vgl de datatypes tussen mySQL, SQL Server, Oracle en de standaard

Relationele sleutels

- Nodig om uniciteit van rijen te waarborgen
- Een sleutel behoort altijd tot 1 relatie/tabel
  Er zijn geen sleutels die zich over meerdere tabellen spreiden
- Supersleutel (**superkey**)
  - Een verzameling van (1 of meer) attributen, die uniek is voor elke rij
  - Vooral van theoretisch belang (komt praktisch gezien niet voor)
- Kandidaatsleutel (**candidate key**)
  - Een **minimale** supersleutel (m.a.w. **irreducibel** of **irreducible**)
  - Er bestaat m.a.w. geen enkel echte deelverzameling van deze sleutel die op zijn beurt een supersleutel/uniek is (echte deelverzameling = proper subset)
- Primaire sleutel (**primary key**)
  - De kandidaatsleutel gekozen om rijen uniek te identificeren
  - Dit is dus een keuze uit de kandidaatsleutels
  - Criteria hierbij kan zijn de grootte (aantal attributen)
  - Hoewel er meerdere kandidaten kunnen zijn is er maar 1 de primaire

Relationele sleutels (VERVOLG)

- Verwijssleutel (**foreign key**)

  - Een verzameling van (1 of meer) attributen, die verwijst naar een kandidaatsleutel van een (andere of dezelfde) tabel

  - Het woordje sleutel is hierbij gevaarlijk! Een verwijssleutel is dus eigenlijk geen sleutel maar verwijst naar ('targets') een sleutel!

  - Een verwijssleutel hoeft dus helemaal niet uniek te zijn!

  - De relatie waarnaar verwezen wordt noemt men soms de **home relation**

- Als een sleutel uit meer dan 1 attribuut bestaat noemen we hem **samengesteld** (**composite**)
- De andere (niet primaire) kandidaatsleutels noemt men ook wel eens **alternate keys**
- Alle sleutels zet men onder 1 noemer relationele sleutels (relational keys)

Relationele sleutels (VERVOLG)

- Eigenschap 1 vertaald zich dus als : elke relatie moet een primaire sleutel hebben
- Een kandidaatsleutel kan nooit null bevatten!

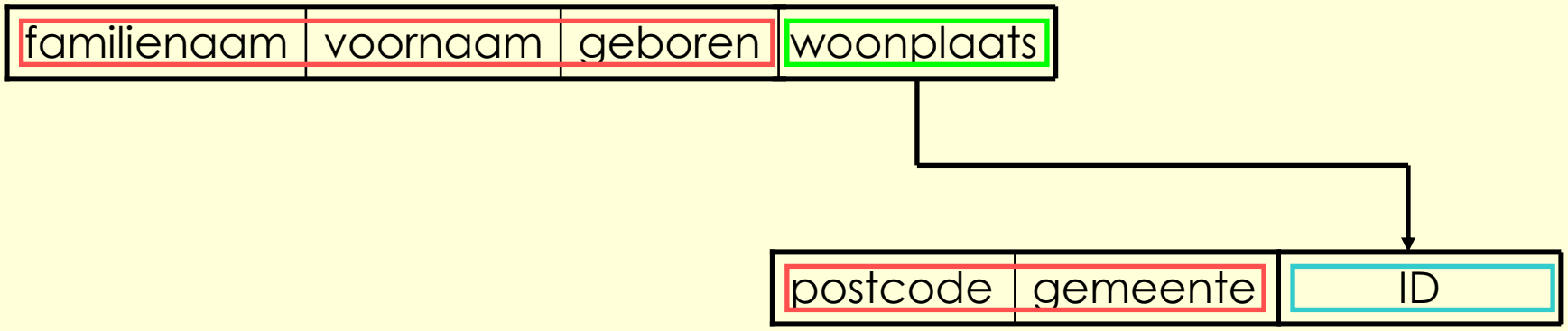
  - Volgt uit de definitie, want als een null toegelaten zou zijn, dan zou dit betekenen dat deze niet minimaal is

  - Strikt genomen is de term null-waarde al contradictorisch

Keuze van sleutels

- Kies een sleutel die een werkelijke betekenis heeft (dit zal als een vloek klinken bij nogal wat DBA's)
- Voer dus niet zomaar een nummering in om een primaire sleutel te hebben (tenzij deze nummering ook in de werkelijkheid wordt doorgedrukt)
- Je kunt altijd een sleutel vooropstellen, desnoods alle attributen samen (of anders heb je velden te weinig!)
- Een artificiële sleutel vermijd het gebruik van samengestelde verwijssleutels en maakt sleutellengte data-onafhankelijk
- Een artificiële sleutel alleen lost het uniciteitsprobleem niet op maar verlegd het (een artificiële kandidaatsleutel kan echter nuttig zijn)
- Kies de kleinste sleutel (andere kandidaatsleutels komen ook in de DDL aan hun trekken)
- Het legendarisch veld ID en types zoals autonummering zijn nuttig maar gevaarlijk voor de onwetende
- Neem de meest waarschijnlijke sleutel (vb. afspraken dokters/patiënten)

Voorbeeld



Nadelen

- een samengestelde verwijssleutel zorgt voor zeer veel redundantie

- het aanpassen van dergelijke sleutelwaarden is dikwijls geen sinecure

- het gebruik van een verwijssleutel naar een primaire sleutel (met betekenis) zorgt meestal voor een sterk data-afhankelijk en weinig performant gebruik van secundair geheugen

  vb. als meer mensen in Sint-Pieters-Kapelle wonen dan in Mol, …

- oplossing : we voeren een artificiële kandidaatsleutel in (met de naam ID) Maak echter niet de fout deze als primaire sleutel aan te duiden!

Relationele Integriteit

- Data-onafhankelijke integriteit
  - Domein-integriteit (domain constraints)

    - een attribuut moet tot een bepaald domein behoren
  - Entiteit(s)-integriteit (entity integrity)

    - elke rij moet uniek zijn

    - een kandidaatsleutel mag geen null bevatten
  - Referentiële integriteit (referential integrity)

    - een verwijssleutel moet naar een geldige kandidaatsleutel(waarde) verwijzen (of volledig null zijn)
- Data-afhankelijke integriteit
  - Bedrijfsregels (enterprise constraints)

    ook nog business-rules genoemd

    (typisch de 1-k type relaties waarbij k een exact gekend getal is)

Relationele Integriteit (VERVOLG)

- Wat met het wissen van rijen waar naar verwezen werd?
  - cascading deletes
  - cascading nulls
  - set defaults (slecht idee)
  - no action (verbod?)
- Wat met het wijzigen van de waarde van een kandidaatsleutel waar naar verwezen wordt?
  - cascading updates
  - cascading nulls
  - set defaults (slecht idee)
  - no action (verbod?)

  - kan meestal vermeden worden door dit te verbieden (of een artificiële kandidaatsleutel in te voeren)

Soorten relaties

- Een relatie kan benoemd worden in 2 richtingen
- Indeling naar het aantal enteiten (≠tabellen)

unaire relaties	vb. is baas van
binaire relaties	vb. woont in
ternaire relaties	vb. product/project/persoon
quaternaire relaties	
…	
n-aire relaties	

- Indeling naar het aantal tuples die langs beide zijden deelnemen

1 – 1 relaties	(one-to-one)	vb. heeft als partner (?)
1 – n relaties	(one-to-many of 1 - ∞)	vb. persoon/woonplaats
m – n relaties	(many-to-many of ∞ - ∞)	vb. factuur/product
1 – k relaties	(geen onderdeel van het relationeel model!)	

- Een ternaire relatie kan niet vervangen worden door 3 binaire (**connection trap**!)

Soorten relaties (VERVOLG)

	unair	binair
1 – 1	kandidaatsleutel(s) (1)	
1 – n	verwijssleutel/kandidaatsleutel	
m – n	extra tabel voor de relatie	

(1) bij binaire 1-1 relaties is de verwijssleutel ook een kandidaatsleutel

Om **m-n relaties** (en hun eigenschappen) voor te stellen hebben we een **extra tabel** nodig

Dit is een informele indicatie van het feit dat 'relatie' en 'tabel' één en hetzelfde begrip zijn

Normalisatie

- Doel
  - Vermijden van redundantie ('one fact in one place')
  - Reduceren van opslagruimte
  - Verhogen van consistentie en integriteit
- Deze doelstellingen zijn rechtstreeks gerelateerd aan een aantal update-problemen (Eng. **update anomalies**)

  insert - anomalies
  deletion - anomalies
  modification - anomalies
- Hoewel normalisatie een proces is dat formeel beschreven kan worden en zich baseert op sleutels en functionele afhankelijkheden tussen attributen, kunnen al een aantal ad-hoc regels worden gegeven

Voorbeeld

voornaam	familienaam	geboren	postcode	gemeente

- Toevoeg-problemen

  - we kunnen geen woonplaats toevoegen zonder persoon toe te voegen

  - we moeten telkens de gegevens van de woonplaats (hopelijk correct) opnieuw invullen

- Verwijder-problemen

  - als we alle personen uit een gemeente verwijderen zijn we ook de informatie omtrent die gemeente kwijt

- Aanpas-problemen

  - als we de gegevens van een woonplaats willen wijzigen moet dit gebeuren bij alle personen (uit onze tabel) die er wonen

Normalisatie (VERVOLG)

Informele regels

- Zorg voor een duidelijke scheiding van de verschillende entiteiten

In voorgaand voorbeeld zijn postcode en naam van de gemeente geen eigenschappen van personen maar van gemeentes

- Zorg ervoor dat alle attributen

enkel en alleen (voldoende voorwaarde)

éénduidig (nodige voorwaarde)

kunnen worden afgeleid uit de primaire sleutel

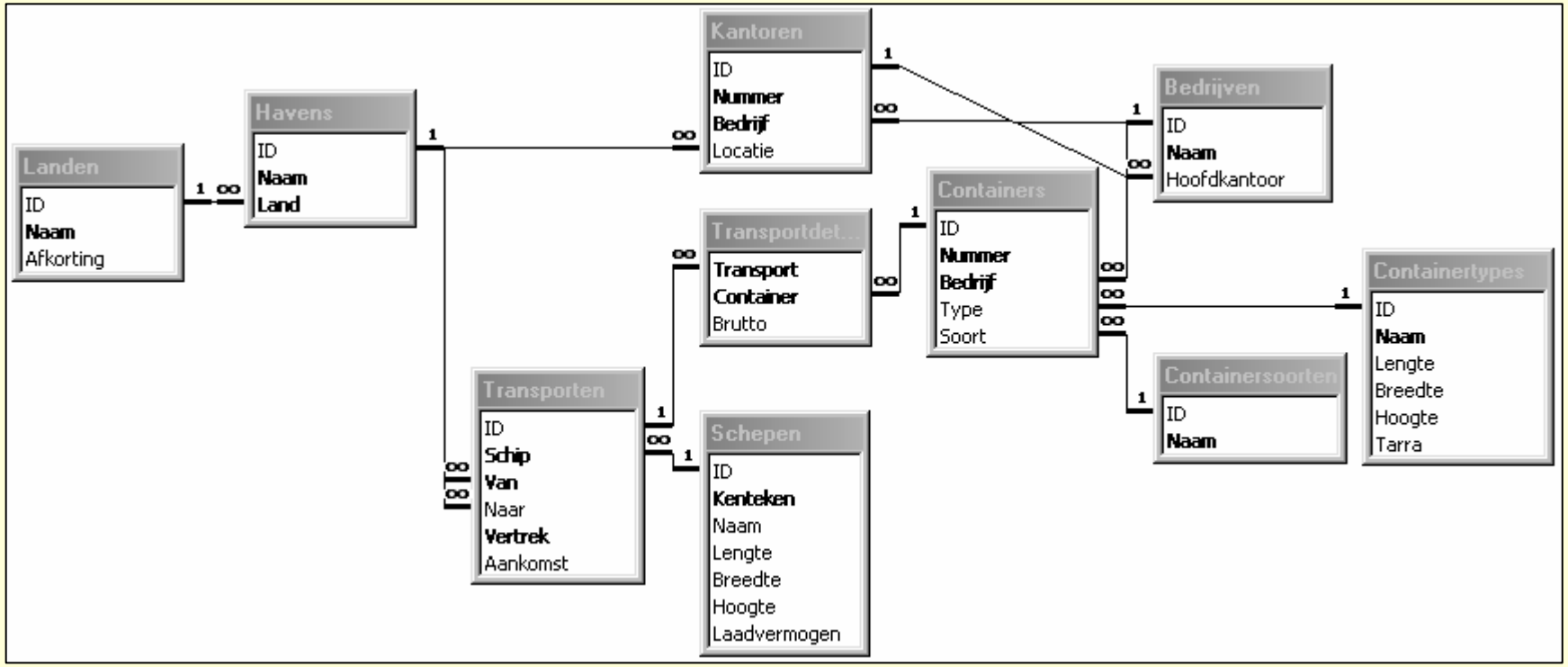
Als de waarden van de primaire sleutel gekend zijn, dan moeten alle attribuutwaarden gekend zijn (is helaas meestal voldaan)

Als de naam van de woonplaats gekend is en bijgevolg ook de postcode dit zou zijn dan zijn er andere afhankelijkheden dan deze t.o.v. de primaire sleutel (deze moeten gebroken worden)

Oefening

In het kader van wereldwijde internationalisering van de havens wil een overkoepelend orgaan één logistieke databank in het leven roepen. Ons wordt gevraagd de analyse te doen. Reders (firma's) aller landen hebben containers van verschillende types en soorten. Types zijn bijvoorbeeld kleine containers, grote containers, ... Types maken dus vooral een onderscheid tussen de containers op basis van hun afmetingen en hun gewicht. Soorten specifiëren eerder de functionaliteit van de containers. Zo zijn er droge containers, koelcontainers, ... Containers hebben een nummer dat uniek is binnen de firma (eigenaar). Een lege container heeft, afhankelijk van zijn type, een bepaald leeg gewicht (tarra). Over de ganse wereld verspreid hebben firma's kantoren die zich altijd in een haven bevinden. Het is uiteraard van belang dat men van een bepaald kantoor kan weten in welk land het is gevestigd. Kantoren krijgen ter identificatie een uniek nummer van hun firma. Van alle kantoren die een firma rijk is, wordt er één aangeduid als hoofdkantoor. Havens, landen en firma's hebben voor zo ver we weten een unieke naam. Het is misschien toch veiliger dat we er voor zorgen dat twee havens dezelfde naam kunnen dragen over gans de wereld. De logistieke dienst wil nu transporten kunnen ingeven waarbij men oorsprong en bestemming opgeeft alsook wanneer vertrek en aankomst geschieden. Op een transport kunnen vanzelfsprekend verschillende containers (zelfs van verschillende firma's) een plaatsje krijgen. Elk transport heeft echter een maximum volume en gewicht die niet mogen overschreden worden. Het spreekt voor zich dat dit kan gecontroleerd worden aan de hand van de afmetingen van de containers alsook hun bruto gewicht. Ter herinnering netto = bruto - tarra.

Oplossing



- Dit schema werd met MS Access 2000 geconstrueerd
- Er bestaat een standaard om relatieschema's voor te stellen

  ERD = **Entity Relationship Diagrams** (entiteiten/relaties)
- Enteiten die voor hun bestaan volledig afhankelijk zijn van andere entiteiten, noemen we **zwakke enteiten** (de primaire sleutel bevat een verwijssleutel)

DDL

	SCHEMA	DOMAIN	TABLE	VIEW	INDEX
CREATE	X	X	X	X	X
DROP	X	X	X	X	X
ALTER		X	X		

Opmerkingen:

- Er is ook nog zoiets als een ASSERTION
  een uitbreiding van de CHECK clausule uit CREATE TABLE over meerdere tabellen
- Bijna alle dialecten voorzien een ruimer gebruik voor de CREATE en DROP opdrachten
  vb. CREATE/DROP USER/TABLESPACE/
  CREATE/DROP DATABASE (vergelijkbaar met een schema)
- DDL heeft weinig meer om het lijf dan syntax (kookboeksysteem)

CREATE TABLE

```
create table aankopen
(
    id              int             not null        unique          ,
    lidnummer       int             not null                        ,
    datum           timestamp       not null                        ,
    betaling        varchar(15)     null                            ,
    betaald         float           not null        default 0       ,
    constraint AankopenPK           primary key (lidnummer, datum)  ,
    foreign key (lidnummer) references leden(lidnummer)
            on update       cascade
            on delete       cascade                                 ,
    constraint BetaaldPositief      check (betaald>=0)              ,
    constraint BetaaldFormaat       check (betaling IN
    ('VISA','CASH','CREDITCARD'))
);
```

CREATE TABLE (VERVOLG)

```
create table aankoopdetails
(
    aankoop             int                         not null  ,
    boek                int                         not null  ,
    aantal              smallint        default  1  not null  ,
    prijs               float                       not null  ,
    primary key (aankoop, boek)                                 ,
    constraint AankoopdetFK foreign key  (boek) references boeken(id),
    foreign key    (aankoop) references aankopen(id)            ,
    constraint PrijsPositief          check   (prijs>=0)        ,
    constraint AantalPositief         check   (aantal>0)        ,
    constraint MaxBoekenPerAankoop
                    check ( NOT EXISTS (      SELECT aankoop
                                              FROM  aankoopdetails
                                              GROUP BY aankoop
                                              HAVING count(*)>5       ))
);
```

VIEWS

- Base tables vs. virtual tables

  - Een **basis relatie** is een benoemde relatie waarvan de tuples fysisch zijn opgeslagen in de databank

  - Een **virtuele relatie** of **view** is het dynamisch resultaat van één of meerdere relationele bewerkingen (SQL) op andere virtuele en/of basistabellen

  - Een virtuele tabel wordt niet opgeslagen maar (telkens opnieuw) gegenereerd op het moment dat hij nodig is (vandaar *dynamisch*)

  - Een virtuele tabel zal dus ook een naam moeten krijgen, echter niet het resultaat maar de definitie wordt opgeslagen in de databank

  - De manier bij uitstek om een view te definiëren is uiteraard met SQL

  - Bijgevolg is een virtuele tabel te vergelijken met een opgeslagen (en dus benoemde) sql-query, die zich als tabel gedraagt

  - Bewerkingen op virtuele tabellen worden vertaald naar bewerkingen op basistabellen

  - Uiteraard zullen dus niet alle views **updatable** blijken

VIEWS (VERVOLG)

- Doel (voordelen)
  - Laat integriteitscontrole toe (check option)
  - Flexibel en krachtig beveiligingssysteem
  - Andere view is andere 'kijk' op dezelfde gegevens (customization)
  - Vereenvoudiging van complexe bewerkingen op basisrelaties
  - Hoge abstractiegraad en data-onafhankelijkheid

  vb. een tabel verticaal opspitsen, tabelstructuur wijzigen
- Syntax

  CREATE VIEW <*naam*> [(<kolomnaam>[, …])] AS <*subselect*>
  [WITH [CASCADED | LOCAL] CHECK OPTION]

  DROP VIEW <*naam*> [RESTRICT | CASCADE]

  - de subselect noemt men de **defining query**
  - 'with check option' controleert de rijen op de where condities alvorens ze toe te voegen of te wijzigen (rijen die de view binnenkomen of verlaten noemt men **migrating rows**)
  - CASCADED EN LOCAL hebben betrekking op view-hiërarchie
  - RESTRICT (standaard) laat drop niet toe indien views op deze view bestaan, CASCADE dropt alle afhankelijke views ook

VIEWS (VERVOLG)

- Updatable

  - Een view is updatable als hij is gebaseerd op slechts één basisrelatie en diens primaire sleutel (of een kandidaatsleutel) bevat

  - Men maakt in de literatuur een onderscheid tussen **theoretically updatable** en **partial updatable**

  - Constructeurs hebben allemaal hun eigen restricties omtrent de *updatability* van views en hun eigen techieken om deze updatable te maken

  vb. zo kent Oracle instead-of triggers op views (zie later)
- Horizontale vs. verticale views
- Nadelen

  update beperkingen, structuur beperkingen, performantie
- Materialized views

  Laten toe dat het resultaat van een view toch fysisch wordt opgeslagen (performanter doch niet dynamisch)

INDEXEN

- Rijen hebben geen volgorde
  - uiteraard is er een fysische volgorde
  - helaas is er maar 1 fysische volgorde mogelijk
- Pointerkettingen
  - Laten toe andere volgordes te doorlopen zonder fysisch te herschikken
- Een **index** is de relationele term voor dergelijke pointerketting
  - een index wordt gelegd op kolommen (van een basisrelatie!) waarop veel wordt geordend of gezocht
  - te veel indexen hebben het omgekeerde effect
  - elke index moet bijgeschaafd worden bij updates en inserts
- Syntax

  CREATE [UNIQUE] INDEX <*naam*>
  ON <*tabel*> (<*kolom*> [ASC | DESC] [, …])

SEQUENCES

- Gebruik (bedoeling)
  - Genereren van unieke nummers
  - Deze hoeven niet elkaar oplopend op te volgen
  - Gaten worden doorgaans niet opgevuld
  - Zijn dus met aandacht te gebruiken voor nummeringen die men werkelijk wil doordrukken (zoals factuurnummers e.d.)
  - In de eerste plaats dus bedoeld voor artificiële nummering
- Voorbeelden

  Access – Autonumber (type)
  - reeks of willekeurige getalen
  - laat niet toe manueel waarden in te geven

  Oracle – SEQUENCE (objecten)

  CREATE SEQUENCE AankoopSeq
  START WITH 1 INCREMENT BY 1 CACHE 30;
  in een insert … AankoopSeq.NEXTVAL of AankoopSeq.CURRVAL

PL-SQL

- Nut/toepassingen

  Bijna elke constructeur voorziet in een procedurele programmeertaal aan boord van het DBMS. Naast concurrentiële en commerciële drijfveren zijn de belangrijkste toepassingen

  - Stored procedures - subroutines worden (in gecompileerde vorm) in de databank opgeslagen. Dit laat toe eigen functies te definiëren.

  - Triggers – Vergelijkbaar met event-handlers bij een GUI wordt een trigger in een database uitgevoerd naar aanleiding van een gebeurtenis (insert, update, shutdown, …)

- Discussie

  Het kamp is verdeeld in voorstanders van databasetoepassingen aan boord van de database en voorstanders van een business-layer die volledig buiten de DBMS zit. Snelheid en platformafhankelijkheid wegen hier weer tegen elkaar op.

PL-SQL (VERVOLG)

- Geen dialect van SQL
- Uitbreiding van SQL met procedurele programmeertaal (PL is afkomstig van Ada)
  Aan boord van de database server (en sommige clients)
- PL-SQL is specifiek voor Oracle
  - Elke constructeur ontwikkelt z'n eigen procedurele taal
  - SQL Server heeft de taal T-SQL
  - Access heeft VBA aan boord
- Basisstructuur is de 'blok'
  - Anonieme blokken
  - Gelabelde blokken (vergelijkbaar met anonieme blokken)
  - Subroutines : procedures of functies
  - Triggers

PL-SQL (VERVOLG)

- Het is mogelijk DML SQL statements in PL-SQL op te nemen
- DDL SQL kan niet rechtstreeks maar dynamisch via het EXECUTE IMMEDIATE statement
- PL-SQL is wat men noemt een **host-language**, SQL is dan de DSL (**data sub language**) men spreekt van **embedded sql** (≠dynamic sql)
- Er is sprake van een **impedance mismatch** tussen SQL en een host-taal zoals PL-SQL

  - SQL is *set-oriented*, terwijl een procedurele programmeertaal meestal *record-oriented* is

  - oplossing hiertoe is de introductie van wat men een **cursor** noemt

PL-SQL (VERVOLG)

```
<<aLabel>>
DECLARE /* Moet weg indien sectie niet bestaat */
   v_Num1       NUMBER  := 1;
   v_Num2       NUMBER  := 2;
   v_String1    VARCHAR2(15)   := ‘Hello’;
   v_String2    VARCHAR2(15)   := ‘World!’;
BEGIN
   INSERT INTO temp_table (num_col, char_col)
        VALUES (v_Num1, v_String1);
   SELECT char_col
        INTO v_String1
        FROM temp_table
        WHERE num_col = v_Num1;
   /* set serveroutput on */
   DBMS_OUTPUT.PUT_LINE(v_String1 || ‘ ‘ || v_String2);
EXCEPTION /* Moet weg indien sectie niet bestaat */
   WHEN others THEN
        Raise_application_error(-20000, ‘Een fout is opgetreden');
END aLabel;
```

PL-SQL (VERVOLG)

- Operator

  ** + - * / = != < > <= >= := | |

  IS NULL, NOT AND OR LIKE BETWEEN

- Variabelen

  - PL-SQL kent meer datatypes dan SQL

  vb. BOOLEAN

  - %TYPE bepaald het datatype van een kolom

  vb. boeken.ISBN%TYPE

  - %ROWTYPE bepaald het recordtype van een tabel

  vb. boeken%ROWTYPE

- Programmastructuren

  IF … THEN … ELSIF … THEN … ELSE … END IF;

  CASE … WHEN … THEN … ELSE … END CASE;

  LOOP … END LOOP; in combinatie met EXIT WHEN of IF … THEN EXIT END IF;

  FOR … IN [REVERSE] *van* .. *tot* LOOP … END LOOP;

  WHILE … LOOP … END LOOP;

Embedded SQL

- Enkel DML en Transaction control
- DDL kan via **execute immediate** (DML ook)

```
DECLARE
    v_SQLString  VARCHAR2()   := 'INSERT INTO temp_table (
          (num_val, char_val) VALUES ('';
    v_Num  NUMBER             := 10;
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE temp_table (
          num_val      NUMBER,
          char_val     CHAR(10))';
    v_SQLString := v_SQLString || v_Num || ',''Hello'')';
    EXECUTE IMMEDIATE v_SQLString;
END;
-- Hier gebeurt impliciete conversie van datatypes
```

Embedded SQL (VERVOLG)

```
DECLARE
   v_Boek    boeken%ROWTYPE;
   v_Titel   boeken.titel%TYPE;
BEGIN
   SELECT *
       INTO v_Boek
       FROM boeken
       WHERE ISBN = ‘0201711591‘;
   SELECT titel
       INTO v_Titel
       FROM  boeken
       WHERE ISBN = ‘0201711591‘;
   DBMS_OUTPUT.PUT(v_Titel || ‘ = ‘ || v_Boek.titel);
   COMMIT;
END;
```

Embedded SQL (VERVOLG)

- Karaktervergelijking
  - blank-padded algoritme (fixed size strings vb. CHAR)
  - non-blank-padded algoritme (variable length strings vb. VARCHAR)
- TAAK : zoek deze algoritmes op en vergelijk ze!!!

Transacties

- Een transactie is een ondeelbaar geheel van operaties binnen de DBMS die in hun totaliteit slagen of mislukken
- Het slagen of mislukken kan ook gestuurd worden via de opdrachten COMMIT [WORK] en ROLLBACK [WORK]
- Men kan transacties opdelen via SAVEPOINTS

  - een rollback tot aan een savepoint is mogelijk maar laat de transactie open staan

  - een savepoint blijft actief na een rollback (opnieuw bruikbaar)
- SQL*Plus doet een COMMIT bij afsluiten
- Bovendien is er de optie autocommit

  Voert een commit uit na elke SQL opdracht
- PRAGMA AUTONOMOUS_TRANSACTION

  - maakt van een blok een autonome transactie

  - niet bruikbaar voor sub-blokken (alleen voor buitenste top-level blok)

  - de blok moet afgesloten worden met COMMIT of ROLLBACK (zie * p. 119)

Transacties (VERVOLG)

```
BEGIN
   INSERT INTO temp_table (char_col)
       VALUES (‘Eerste’);
   SAVEPOINT A;
   INSERT INTO temp_table (char_col)
       VALUES (‘Tweede’);
   SAVEPOINT B;
   INSERT INTO temp_table (char_col)
       VALUES (‘Derde’);
   SAVEPOINT C;
   -- ROLLBACK TO A;
   -- ROLLBACK TO B;
   COMMIT;
END;
```

Transacties (VERVOLG)

```
CREATE OR REPLACE PROCEDURE Autonoom AS
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO temp_table VALUES (-10,'sub');
   -- ROLLBACK TO A
   -- Foutief, savepoint is lokaal aan parent-blok
   COMMIT;
   -- Indien geen commit of rollback foutmelding (zie * p. 117)
END Autonoom;

BEGIN
   INSERT INTO temp_table VALUES (-10,'parent');
   SAVEPOINT A;
   Autonoom;
   -- Ondanks de rollback wordt de procedure commit
   ROLLBACK;
END;
```

Cursors

- Wat
  - het antwoord op impedance mismatch
  - een handle of pointer naar de context area of recordset
  - meestal intern, maar manipuleerbaar via methodes of subroutines
- Stappen
  1. Declaratie
  2. Openen voor een bepaalde query (kan ook vastliggen bij definitie) Dit plaatst de cursor wijzend naar het eerste record
  3. Fetch en navigatie (meestal in een procedurele lus)
  4. Sluiten
- Er zijn impliciete en expliciete cursors
  - alle andere SQL statements impliceren een verborgen cursor
  - wij behandelen uiteraard de voor ons interessante expliciete vorm
  - de impliciete cursor noemt SQL (hoeft niet gedefinieerd, geopend of gesloten te worden) en begrijpt eveneens de cursorattributen (zie volgende slide)

Cursors (VERVOLG)

- Access / VBA (DAO)

```
Dim db          AS Database
Dim rs          AS Recordset
Set db = Opendatabase("c:demo.mdb")
' Currentdb binnen Access duidt de huidige db aan
Set rs  = db.Openrecordset("SELECT * FROM Leden WHERE " & _
        " voornaam='" & Inputbox(…) & "'")
Do While Not rs.EOF
        Debug.Print rs!Familienaam
        rs.Movenext
Loop
```

- Oracle / PL-SQL zie volgende slide
- Cursors laten makelijk parameterisatie en dus dynamische sql toe
- Cursorattributen in PL-SQL

  %FOUND %NOTFOUND %ISOPEN %ROWCOUNT (aantal fetched!)

Cursors (VERVOLG)

```
CREATE OR REPLACE PROCEDURE Procedure (
   p_Woonplaats IN Woonplaatsen.Naam%TYPE,
   p_Postcode    IN Woonplaatsen.Postcode%TYPE ) RETURN Number
AS
   v_WoonplaatsID        woonplaatsen.ID%TYPE;
   v_Familienaam         leden.familienaam%TYPE;
   v_Voornaam            leden.voornaam%TYPE;
   CURSOR c_Leden IS
        SELECT   voornaam, familienaam FROM leden
        WHERE    woonplaats = v_WoonplaatsID; -- late binding
BEGIN
   SELECT ID
        INTO             v_WoonplaatsID FROM        woonplaatsen
        WHERE            postcode = p_Postcode AND naam = p_Woonplaats;
   OPEN  c_Leden;
   LOOP
        FETCH c_Leden INTO v_Voornaam, v_Familienaam;       -- OPGELET !!!
        EXIT WHEN c_Leden%NOTFOUND;                         -- OPGELET !!!
        DBMS_OUTPUT.PUT_LINE (v_Familienaam || ' ' || v_Voornaam);
   END LOOP
   CLOSE c_Leden;
   -- v_WoonplaatsID kan gewijzigd worden en de cursor opnieuw geopend
   RETURN 1;
END;
```

Cursors (VERVOLG)

```
…
OPEN  c_Leden;
FETCH c_Leden INTO v_Voornaam, v_Familienaam;
WHILE c_Leden%FOUND LOOP
     DBMS_OUTPUT.PUT_LINE (v_Familienaam || ‘ ‘ || v_Voornaam);
     FETCH c_Leden INTO v_Voornaam, v_Familienaam;
END LOOP;
CLOSE c_Leden;
…


…
-- declaratie v_Lid, open, close en fetch zijn impliciet
FOR v_Lid IN c_Leden LOOP
     DBMS_OUTPUT.PUT_LINE (v_Lid.Familienaam || ‘ ‘ ||
v_Lid.Voornaam);
END LOOP;
…
```

Cursors (VERVOLG)

En verder …

UPDATE CURSORS

Laten wijzigen van data toe

…FOR UPDATE [OF <*kolom*>[,…]] [NOWAIT | WAIT *n*]
…
UPDATE … WHERE CURRENT OF c_cursor

CURSOR VARIABELEN

- Laten toe de query gekoppeld aan een cursor dynamisch te wijzigen
- Er zijn **constrained** en **unconstrained** cursorvariabelen

Triggers

- 3 Soorten

  DML triggers

  INSTEAD-OF triggers

  system triggers

- Indeling

  2 levels : row-level vs. statement level

  2 timings : before vs. after

  3 statements : insert, update, delete

- Toepassingen

  complexe integriteitsregels

  auditing / logging

  automatisatie en batch-opdrachten

  automatische publicatie

Triggers (VERVOLG)

```
create or replace trigger MaxBoekenAankoop
before insert ON aankoopdetails
for each row
declare
 v_num NUMBER;
 too_much_books EXCEPTION;
begin
 DBMS_OUTPUT.PUT_LINE('MaxBoekenAankoop');
 SELECT count(*) INTO v_num FROM Aankoopdetails WHERE
   Aankoop=:new.Aankoop;
 DBMS_OUTPUT.PUT_LINE(v_num || ' aankoopdetail(s) voor aankoop '
   || :new.Aankoop);
 if v_num>=5 then
    RAISE too_much_books;
 end if;
exception
    WHEN too_much_books THEN
   Raise_application_error(-20000, 'Maximum aantal aankoopdetails
   is 5');
end MaxBoekenAankoop;
```

Triggers (VERVOLG)

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab
  FOR EACH ROW
WHEN (:new.Empno > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff  := :new.sal  - :old.sal;
    dbms_output.put('  Old salary: ' || :old.sal);
    dbms_output.put('  New salary: ' || :new.sal);
    dbms_output.put_line('  Difference ' || sal_diff);
END;
```

FD's (FA'en)

- Functionele afhankelijkheid FA

  Functional dependency FD

  - Als R een relatie is en X en Y zijn arbitraire deelverzamelingen van de verzameling attributen van R, dan zegt men dat een Y **functioneel afhankelijk** is van X als en slechts als met een waarde voor X, op elk moment, precies één waarde voor Y overeenstemt. (stilzwijgend veronderstellen we een relatievariabele)

  - We zeggen ook dat Y bepaald wordt door X.

  - X noemen we de **determinant**, Y de **afhankelijke**.

- Notatie

  Stel R = {postcode, naam, land} waarbij R dus woonplaatsen voorstelt

  {postcode, naam}→ {land}

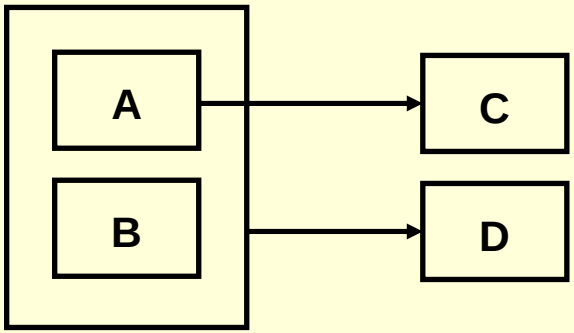
  Bij singletons laat men meestal de accolades vallen

  {postcode, naam}→ land

FD's (FA'en) (VERVOLG)

- Grafische voorstelling

  grafisch kunnen we een FA voorstellen d.m.v. een **FD diagram**. Een voorbeeld:



A $\rightarrow$ C

{A,B} $\rightarrow$ D

- Triviale FA'en

  - we noemen een FA **triviaal** als de afhankelijke een deelverzameling is van de determinant

  - voorbeeld {A,B} $\rightarrow$ A

FD's (FA'en)

- Semantiek

  - een functionele afhankelijkheid weerspiegelt eigenlijk een integriteitsregel

  - bijgevolg zijn functionele afhankelijkheden een manier om uit te drukken wat data betekent

  - een functionele afhankelijkheid is uiteindelijk een many-to-one relatie tussen determinant en afhankelijke

  - als dusdanig een afbeelding van veel op 1, vandaar het woord *functie* in de naam

- Toepassing

  - normalisatie

  - het minimaliseren van het aantal integriteitsregels te controleren door het DBMS

FD's (FA'en) (VERVOLG)

Afleiden van FA'en

uit FA'en kunnen nieuwe/andere FA'en afgeleid worden.

hiertoe gebruiken we de **inferentieregels/axioma's van Armstrong**:

1. *Reflexiviteit*

$$B \subset A \quad \Rightarrow \quad A \rightarrow B$$

opmerking: dit is juist de definitie van een triviale FA

2. *Vermeerdering (augmentation)*

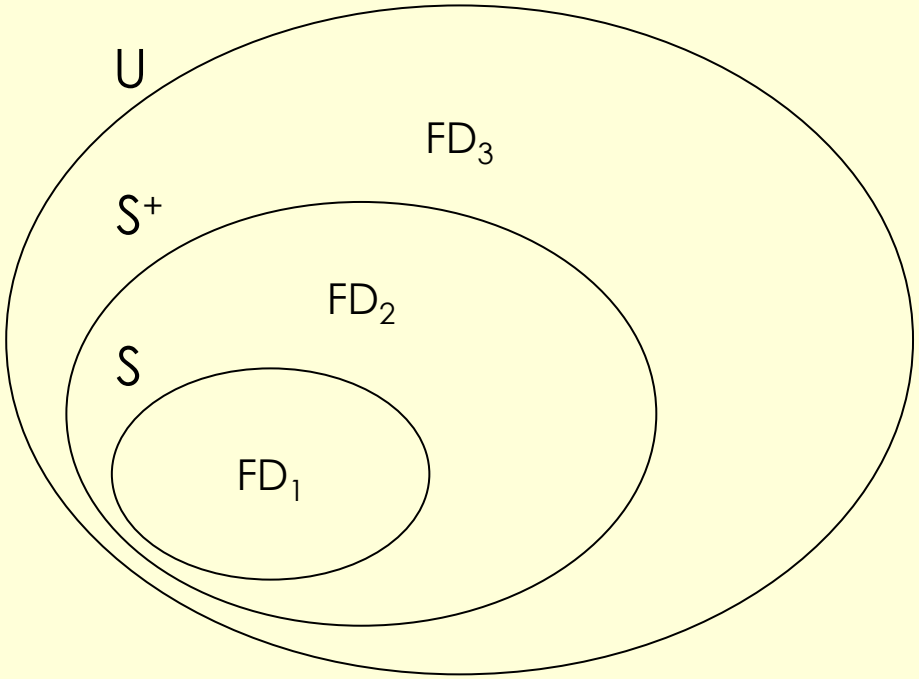
$$A \rightarrow B \quad \Rightarrow \quad AC \rightarrow BC$$

opmerking: AC betekent {A,C} of nog $A \cup C$

3. *Transitiviteit*

$$\begin{array}{l} A \rightarrow B \\ B \rightarrow C \end{array} \quad \Rightarrow \quad A \rightarrow C$$

Deze set regels noemen we **sound** en **complete**. Elke afleidbare FA uit een verzameling FA'en kan worden afgeleid m.b.v. deze regels en omgekeerd.



R	=	relatie van attributen $\{A_1 \ldots A_n\}$
U	=	verzameling van alle mogelijke FA'en, geschreven met de attributen van relatie R (bevat dus 2^{2n} mogelijke FA'en, nl. 2^n det x 2^n afh)
S	=	een willekeurige deelverzameling van U
S^+	=	alle FA'en afleidbaar uit S met de drie axioma's
'sound'	=	elke FA afgeleid uit S met de 3 axioma's ligt binnen S^+
'complete'	=	elke FA uit S^+ is afleidbaar uit S met de 3 axioma's

FD's (FA'en) (VERVOLG)

Afgeleide regels (verderop wordt hun irrelevantie aangetoond)

4. Zelfbepaling (self-determinantion)

$A \rightarrow A$

5. Decompositie

$A \rightarrow BC \quad \Rightarrow \quad A \rightarrow B,\ A \rightarrow C$

6. Unie

$A \rightarrow B,\ A \rightarrow C \quad \Rightarrow \quad A \rightarrow BC$

7. Compositie

$A \rightarrow B,\ C \rightarrow D \quad \Rightarrow \quad AC \rightarrow BD$

8. Eenheidsstelling (Darwen's general unification theorem)

$A \rightarrow B,\ C \rightarrow D \quad \Rightarrow \quad A \cup (C\text{-}B) \rightarrow BD$

Voorbeeld (VERVOLG)

Te bewijzen

$A \rightarrow B$

$C \rightarrow D \quad \Rightarrow \quad A \cup (C\text{-}B) \rightarrow BD$

Bewijs

1) $A \rightarrow B$	(gegeven)
2) $C \rightarrow D$	(gegeven)
3) $A \rightarrow B \cap C$	($B \cap C \subset B$ en 1)
4) $C\text{-}B \rightarrow C\text{-}B$	(zelfbepaling)
5) $A \cup (C\text{-}B) \rightarrow (B \cap C) \cup (C\text{-}B)$	(unie van 3 en 4)
6) $A \cup (C\text{-}B) \rightarrow C$	(vereenvoudiging)
7) $A \cup (C\text{-}B) \rightarrow D$	(transitiviteit 6 en 2)
8) $A \cup (C\text{-}B) \rightarrow B \cup D$	(compositie 1 en 7)

Voorbeeld (VERVOLG)

Te bewijzen

$A \rightarrow BC$
$B \rightarrow E$
$CD \rightarrow EF$

$\Rightarrow$ $AD \rightarrow F$

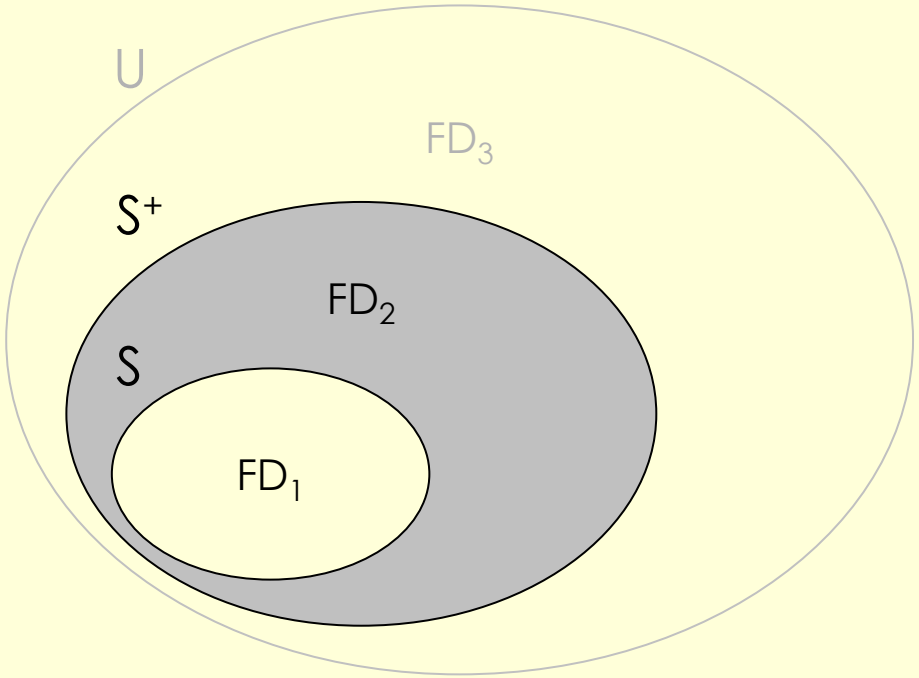
Bewijs

1) $A \rightarrow BC$	(gegeven)
2) $A \rightarrow C$	(decompositie 1)
3) $AD \rightarrow CD$	(vermeerdering 2)
4) $CD \rightarrow EF$	(gegeven)
5) $AD \rightarrow EF$	(transitiviteit 3 en 4)
6) $AD \rightarrow F$	(decompositie 5)

FD's (FA'en) (VERVOLG)

Sluiting (closure) van een stel FA'en

- we weten nu dat we FA'en kunnen afleiden uit andere
- we definiëren de **sluiting van een verzameling FA'en** *S* als de verzameling S^+ van alle FA'en die afgeleid kunnen worden uit de gegeven verzameling
- deze sluiting berekenen op een exhaustieve manier is niet aan te raden, N atributen leveren namelijk 2^{2N} mogelijk FA'en (bovengrens)
- het bewijs van afleidbaarheid van een FA uit een verzameling FA'en *S* of m.a.w. het bewijs dat die FA een element is van de sluiting S^+ van *S* kan niet met de axioma's als er geen afleidbaarheid is! Bovendien is deze manier *ad hoc* en zeker niet voor automatisatie geschikt
- gelukkig kunnen we nagaan of een FA element is van de sluiting zonder deze sluiting te berekenen (zie verder)
- daartoe voeren we een nieuwe definitie in (zie volgende slide)



- Om S te kunnen minimaliseren gaan we S^+ minimaliseren
- S^+ is echter te groot om exhaustief te berekenen
- We zullen echter zien dat het volstaat van een FA te weten of hij afleidbaar is uit S (of m.a.w. behoort tot S+)

-Sterker nog, het blijkt niet nodig om S+ te kennen om te weten of een FA er element van is. Daartoe voeren we volgende definitie (en algoritme) in.

FD's (FA'en) (VERVOLG)

Sluiting (closure) van een stel attributen onder een verzameling FA'en

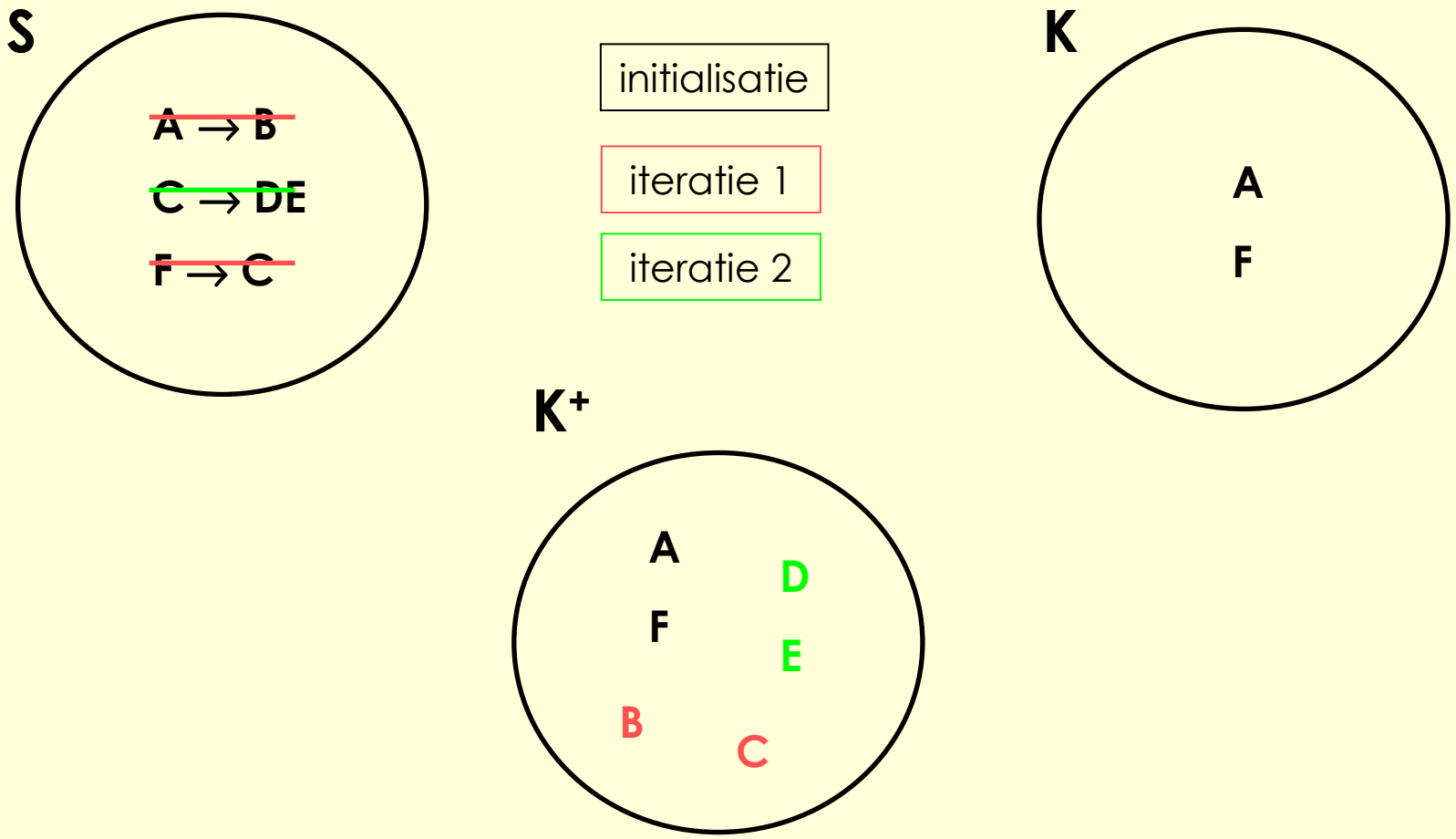
- we definiëren de **sluiting van een deelverzameling attributen** *K* van een relatie R **onder een verzameling FA'en** *S* als de verzameling K^+ van alle attributen die functioneel afhankelijk zijn van (bepaald kunnen worden uit) de gegeven verzameling K onder de geldende FA'en van S
- deze sluiting mag geenzins verward worden met de sluiting van een verzameling FA'en!!! (zie vorige slide)
- hiervoor bestaat een algoritme
- we zullen zien dat deze sluiting ons een middel (algoritme) verleent om zaken over de sluiting van een verzameling FA'en te weten te komen zonder deze effectief te hoeven berekenen
- we geven eerst het algoritme en daarna de onmiddellijke toepassingen ervan

Algoritme

In pseudocode ziet het algoritme er als volgt uit

```
AttributeSet getClosure (AttributeSet K, FDSet S)
{
      AttributeSet Ks = K;
      changed = true;
      while (changed)
      {
                changed = false;
                for (int i=0;i<S.getCount();i++)
                {
                          if (isSubSet(det(S[i]),Ks) && !isSubSet(afh(S[i]),Ks))
                          {
                                    Ks += afh(S[i]));
                                    //optimalisatie: 'schrap' S[i] uit S
                                    changed = true;
                          }
                }
      }
      return Ks;
}
```

Voorbeeld



FD's (FA'en) (VERVOLG)

Betekenis

- indien $K^+ = R$ (m.a.w. alle attributen van R bevat), dan is K een supersleutel (niet noodzakelijk een kandidaatsleutel!)

- om te weten of K ook een kandidaatsleutel is, kan hetzelfde algoritme gebruikt worden, er mag immers geen enkele echte deelverzameling van K bestaan die op zijn beurt een supersleutel is!

- als een verzameling K' een supersleutel is, dan is elke superset van K' dat ook, bijgevolg kun je best beginnen de grootste deelverzamelingen eerst te controleren

vb. als ABC een supersleutel is dan controleren we eerst AB, BC en AC of ze op hun beurt een supersleutel zijn

- indien $Y \subset X^+$ onder S, dan is $X \rightarrow Y \in S^+$ of $S \Rightarrow X \rightarrow Y$

- het volstaat echter het algoritme al af te breken van zodra $Y \subset X^+$

FD's (FA'en) (VERVOLG)

Twee verzamelingen regels S_1 en S_2 hoeven evenwel niet identiek te zijn om hetzelfde te betekenen (i.e. hetzelfde te impliceren).

Indien $S_1^+ \subset S_2^+$ noemt men S_2 een **cover** van S_1. Elke FA die geïmpliceerd wordt door S_1 wordt dan ook geïmpliceerd door S_2.
Indien *ook* S_1 een **cover** van S_2 is dan kan het niet anders dan dat $S_1^+ = S_2^+$. S_1 en S_2 noemt men op dat ogenblik **equivalent**. Twee equivalente sets regels zijn dus zonder meer uitwisselbaar binnen een DBMS die dergelijke regels moet waarborgen (controleren).

Men kan makkelijk bewijzen of twee verzamelingen FA'en al dan niet equivalent zijn. Het volstaat te bewijzen dat elke FA : $X \rightarrow Y$ van S_1 behoort tot S_2^+ of m.a.w. dat $Y \subset X^+$ onder S_2 en omgekeerd dat elke FA van S_2 behoort tot S_1^+. Principieel betekent dit dat je het algoritme n1+n2 keer zult moeten uitvoeren (n_i is hierbij het respectieve aantal FA'en van S_i)

Voorbeeld (VERVOLG)

Te bewijzen

S_1:

$A \rightarrow B$

$AB \rightarrow C$

$D \rightarrow AC$

$D \rightarrow E$

$\equiv$

S_2:

$A \rightarrow BC$

$D \rightarrow AE$

Bewijs

1) $B \subset A^+$ onder $S_2 = \{A,B,C\}$
2) $C \subset \{A,B\}^+$ onder $S_2 = \{A,B,C\}$
3) $\{A,C\} \subset D^+$ onder $S_2 = \{A,B,C,D,E\}$
4) $E \subset D^+$ onder $S_2 = \{A,B,C,D,E\}$
5) $\{B,C\} \subset A^+$ onder $S_1 = \{A,B,C\}$
6) $\{A,E\} \subset D^+$ onder $S_1 = \{A,B,C,D,E\}$

FD's (FA'en) (VERVOLG)

Het is uitermate interessant de verz. regels die een DBMS moet controleren tot een minimum te herleiden, daarom definiëren we een

Minimale set van FA'en (irreducible set of FD's)

een irreducibele verz. FA'en voldoet aan de volgende 3 eisen

1. de afhankelijke van elke FA is minimaal (i.e. een singleton)
2. de determinant is minimaal, men noemt de FA dan **links-minimaal**
3. geen enkele FA kan uit de verz. weggelaten worden zonder de sluiting van de verzameling te wijzigen (i.e. zonder een niet-equivalente verz. te bekomen)

De engelse term voor minimaal is **irreducible**. Het is makkelijk in te zien dat een minimale set die wordt afgeleid van een bestaande set hiermee equivalent zal zijn. Let op, er is echter niet zoiets als dé minimale set. Het is best mogelijk dat 2 verschillende (niet identieke) verzamelingen toch equivalent zijn en beiden minimaal!

FD's (FA'en) (VERVOLG)

Werkwijze om tot een minimale set te komen.

1. de afhankelijke van elke FA is minimaal (i.e. een singleton)

   dit kan probleemloos door elke FA waarvan de afh. geen singleton is te vervangen door FA'en die enkel singletons als afh. hebben m.b.v. decompositie

2. de determinant is minimaal, men noemt de FA dan **links-minimaal**

   dit kan met het algoritme door van elk element van de determinant na te gaan of het kan weggelaten worden (het volstaat echter NIET enkel na te gaan of het afh. Is van de andere elementen van de determinant!)

   een interessante stelling mag dan wel zijn

   $AB \rightarrow C$ en $A \rightarrow B$ $\Rightarrow$ $A\not{B} \rightarrow C$

   er geldt omgekeerd nog steeds niet dat

   $AB \rightarrow C$ en $A \rightarrow C$ $\Rightarrow$ $A \rightarrow B$

FD's (FA'en) (VERVOLG)

Werkwijze om tot een minimale set te komen (vervolg).

3. geen enkele FA kan uit de verz. weggelaten worden zonder de sluiting van de verzameling te wijzigen (i.e. zonder een niet-equivalente verz. te bekomen)

we controleren van elke FA of hij behoort tot de sluiting van de rest (i.e. S exclusief de FA zelf want die impliceert uiteraard zichzelf)met behulp van het algoritme

Voorbeeld (VERVOLG)

Bepaal een minimale cover voor S met R{A,B,C,D,E,F}

AB	→ C	AB	→ C	AB	→ C
C	→ A	C	→ A	C	→ A
BC	→ D	BC	→ D	BC	→ D
ACD	→ B	~~A~~CD	→ B	CD	→ B
BE	→ C	BE	→ C	BE	→ C
CE	→ FA	CE	→ F	CE	→ F
CF	→ BD	CE	→ A	CE	→ A
D	→ EF	CF	→ B	CF	→ B
		CF	→ D	CF	→ D
		D	→ E	D	→ E
		D	→ F	D	→ F

A^+ onder S = {A}

B^+ onder S = {B}

C^+ onder S = {A,C}

$\{A,C\}^+$ onder S = {A,C}

$\{A,D\}^+$ onder S = {A,D,E,F}

$\{C,D\}^+$ onder S = {A,B,C,D,E,F}

E^+ onder S = {E}

F^+ onder S = {F}

Voorbeeld (VERVOLG)

Bepaal een minimale cover voor S met R{A,B,C,D,E,F}

AB	→ C	{A,B}⁺	onder de andere FA'en = {A,B}	AB → C
C	→ A	C⁺	onder de andere FA'en = {C}	C → A
BC	→ D	{B,C}⁺	onder de andere FA'en = {A,B,C}	BC → D
~~CD~~	~~→ B~~			BE → C
BE	→ C	{C,D}⁺	onder de andere FA'en = {A,B,C,D,E,F}	CE → F
CE	→ F	{B,E}⁺	onder de andere FA'en = {B,E}	CF → B
~~CE~~	~~→ A~~	{C,E}⁺	onder de andere FA'en = {A,B,C,D,E,F}	D → E
CF	→ B	{C,E}⁺	onder de andere FA'en = {A,C,E,F}	D → F
~~CF~~	~~→ D~~	{C,F}⁺	onder de andere FA'en = {A,C,D,E,F}	
D	→ E	{C,F}⁺	onder de andere FA'en = {A,B,C,D,E,F}	
D	→ F	D⁺	onder de andere FA'en = {D,F}	
		D⁺	onder de andere FA'en = {D,E}	

Normalisatie

- Wat

  - Het proces om redundantie te vermijden en aldus consistentie te verbeteren (we proberen dus herhaling te vermijden)

  - Dit proces moet omkeerbaar zijn! (zie later)

  - Ter vgl. *security* zorgt er voor dat iemand doet wat hij/zij mag doen, *integriteit* zorgt er voor dat hij/zij dit correct doet

- Hoe

  decompositie van relaties in meerdere relaties d.m.v. de **projectie-operator** uit de relationele algebra (dit is de algebra die werkt met verzamelingen van records). Het is ook deze operator die achter de select-clausule in SQL schuil gaat.

  (re)compositie of het terug samenstellen van de oorspronkelijke relatie uit z'n projecties gebeurd dan d.m.v. de **join-operator** uit de relationele algebra (wel bekend uit de from-clausule in SQL).

Normalisatie (VERVOLG)

- **verliesloze** vs. **verlieslatende** decomposities

  - een **lossless** of **non-loss** decompositie is er één waarbij het proces van decompositie *omkeerbaar* (*reversible*) is. De join van de decomposities levert m.a.w. terug dezelfde relatie (opgelet een verlies van informatie kan zich uiten in méér records/data).

  - is dit niet zo dan noemt men de decompositie verlieslatend of **lossy**.

  - het spreekt voor zich dat we in het normalisatieproces alleen verliesloze decomposities mogen gebruiken!

- stelling van Heath

  - wat maakt een decompositie nu geschikt (=verliesloos)?

  - hier komen FA'en op de proppen

  - als R {A,B,C} een relatie is waarvoor de FA : A $\rightarrow$ B geldt, dan is R gelijk aan de join van z'n projecties {A,B} en {B,C}

  - het mag dus duidelijk zijn dat een verliesloze decompositie er één is waarbij geen FA'en verloren gaan

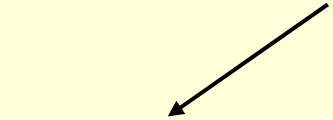
Voorbeeld

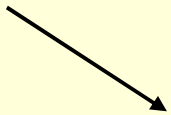
Stel dat : R {ID, naam, code} met

ID → naam

ID → code

ID	naam	code
1	zwart	0
2	zwart	1
3	blauw	255
4	groen	65025





ID	naam
1	zwart
2	zwart
3	blauw
4	groen

ID	code
1	0
2	1
3	255
4	65025

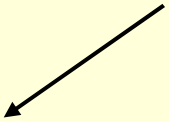
Voorbeeld

Stel dat : R {ID, naam, code} met

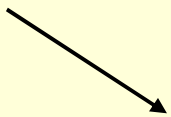
ID → naam

ID → code

ID	naam	code
1	zwart	0
2	zwart	1
3	blauw	255
4	groen	65025
1	zwart	1
2	zwart	0



ID	naam
1	zwart
2	zwart
3	blauw
4	groen



code	naam
0	zwart
1	zwart
255	blauw
65025	groen

Normalisatie (VERVOLG)

- normaalvormen

  - het normalisatieproces drukt zich uit in een aantal **normaalvormen** die wij hieropvolgend zullen bespreken.

  - er zijn 6 normaalvormen die de naam dragen 1NF, 2NF, 3NF, BCNF, 4NF en 5NF

  - NF is de afkorting van **normal form** (ofte normaalvorm idd.)

  - zo staat 2NF dus voor second normal form of tweede normaalvorm

  - de BCNF volgt deze naamgeving niet doordat hij later is ingevoegd als aanvulling op 3NF (die niet afdoende bleek te zijn in bepaalde gevallen)

  - elke normaalvorm impliceert alle lagere normaalvormen met extra restricties. Zo zal een relatie die in 3NF staat vanzelfsprekend ook in 2NF en dus in 1NF staan.

Normalisatie (VERVOLG)

- normaalvormen

  - meestal gaat men in de praktijk minstens tot 3NF/BCNF de hogere vormen 4NF en 5NF zijn heel wat exotischer en veelal automatisch voldaan door de relaties tot in BCNF te brengen

  - 4NF steunt op het begrip **multivalued dependency MVD** (een uitbreiding van het begrip FD). 'Meerwaardige' of 'samengestelde' afhankelijkheid zouden in de Nederlandse taal hiervoor in aanmerking kunnen komen.

  - 5NF steunt op het begrip **join dependency JD** (op zijn beurt een uitbreiding van het begrip MVD). Misschien moeten we de term 'samengestelde afhankelijkheid' hiervoor reserveren? (Hou het bij het Engels ;-)). Een JD is dus de meest algemene vorm van afhankelijkheid in het relationeel model.

  - 5NF noemt men bijgevolg ook wel eens **PJ/NF** (**projection-join normal form**)

  - voor de volledigheid vernoemen we (uit een stoffig theorieboek) ook nog **domain-key normal form DK/NF,** die evenwel niet in het rijtje thuishoort, en **restriction-union normal form** of **(3,3)NF** tussen BCNF en 4NF. Enkel ter volledigheid. Ik betwijfel of DBA's ze zelfs kennen.

1 NF

- Definitie

  - Een relatie staat in 1NF als en slechts als alle onderliggende domeinen (lees types van kolommen) enkel scalaire waarden bevatten

  - Dit slaat in feite op het atomair zijn van alle attributen

  - Meestal kan hiertegen niet gezondigd worden als enkel ingebouwde enkelvoudige types gebruikt worden voor de kolommen (let op sommige systemen laten toe zelf nieuwe samengestelde types te gebruiken dit heeft hen per definitie eerder de titel *object-relationeel* eerder dan zuiver relationeel)

  - Kortom in een zuiver relationeel systeem kan hiertegen niet gezondigd worden en vertekt elke relatie dus van 1NF

  - Verwar atomair zijn niet met het begrip **repeating groups**, dit slaat op het herhalen van gegevens binnen dezelfde tabel, redundantie dus. Iets waar normalisatie (de hogere vormen) mee probeeert af te rekenen. Atomair zijn is dus slechts een eerste stap.

2 NF

- Nieuwe definitie

  B is **volledig FA** van A (*full FD* of *FFD*) als B FA is van A maar van geen enkele echte deelverzameling van A, is dit niet het geval dan noemt men B **partieel afhankelijk** van A dit is dezelfde definitie als *minimaal afhankelijk* of van een *linksminimale FA (irreducible dependent* en *left-irreducible)*

- Definitie

  - Een relatie staat in 2NF als en slechts als 1NF voldaan is en niet-sleutelattributen volledig FA zijn van de primaire sleutel of m.a.w. elk niet-sleutel attribuut is FFD van een kandidaat sleutel

  - Het is duidelijk dat 2NF automatisch voldaan is wanneer er de primaire sleutel enkelvoudig is (i.e. een singletons). Helaas is dit vaak een verdediging van de mensen die artificiële primaire sleutels maken. Noem het eerder een slecht excuus. Of nog, meestal/vaak zijn primaire sleutels niet enkelvoudig.

  - Normaliseren naar 2NF betekent dus praktisch het verwijderen van partiele FA'en of het minimaliseren van de primaire sleutel.

  - Transitieve FA'en van de primaire sleutel verwijderen we pas in de hogere normaalvormen

3 NF

- Nieuwe definitie

  C is **transitief afhankelijk** van A als er B bestaat waarvoor geldt dat

  $A \rightarrow B$ en $B \rightarrow C$ (uiteraard aangenomen dat A niet FA is van B noch C)

- Definitie

  - Een relatie staat in 3NF als en slechts als ze 2NF voldoet (en bijgevolg ook 1NF) en geen enkel niet-sleutelattribuut transitief afhankelijk is van de primaire sleutel (meer algemeen een kandidaat sleutel)

  - Normaliseren naar 3NF betekent dus praktisch het verwijderen van transitieve FA'en t.o.v. de primaire sleutel.

3 NF (VERVOLG)

- Alternatieve definitie

  - Een relatie staat in 3NF als en slechts als de niet-sleutel attributen:

  1/ mutueel onafhankelijk zijn

  2/ volledig (irreducible) afhankelijk zijn van de primaire sleutel (2NF)

BCNF

- Definitie

  - Een relatie in 3NF staat automatisch in BCNF tenzij volgende 3 voorwaarden voldaan zijn:

    1/ twee of meer kandidaatsleutels

    2/ die bovendien samengesteld zijn

    3/ en elkaar overlappen

  - In veel gevallen (waar er dus geen overlappende samengestelde kandidaatsleutels zijn) is BCNF gelijkwaardig met 3NF

  - Heel wat korter is de volgende definitie: Een relatie staat in BCNF als elke determinant een kandidaatsleutel is

  - De afkorting BCNF staat zoals gezegd voor **Boyce - Codd Normal Form**

ERD's

- Entity/Relationship diagrammen
- Symbolen
- Zwakke entiteiten