KU Leuven
Campus De Nayer
Industrieel ingenieur
Opleiding Elektronica-ICT
3e Bachelor — Schakelprogramma

# Databanken: Aanvullingen

Academiejaar 2019-20

Lesgever: J. Vennekens
Cursus ism. H. Crauwels

# Inhoudsopgave


**1 SQL: aanvullingen** **1**
- 1.1 Versies . . . 1
- 1.2 Gegevens . . . 1
- 1.3 Domeinen en tabellen . . . 1
- 1.4 Views . . . 2
- 1.5 DDL statements voor wijzigingen in structuren . . . 3

**2 Analyse van gegevens: normalisatie** **5**
- 2.1 Eerste normaalvorm. . . . 5
- 2.2 Tweede normaalvorm. . . . 6
- 2.3 Derde normaalvorm. . . . 7
- 2.4 Normalisatie: een eindpunt? . . . 8
- 2.5 Een tweede voorbeeld op normalisatie. . . . 8
- 2.6 Oefening. . . . 9

**3 Information retrieval: aanvullingen** **10**
- 3.1 Indexeren van tekst . . . 10
- 3.2 Tonen van de tekst . . . 11
- 3.3 Parameters bij een context index . . . 11
- 3.4 Thesaurus . . . 16
- 3.5 Classificeren van documenten . . . 20

**4 Architectuur van een DBMS** **24**

**5 Architectuur van Oracle.** **26**
- 5.1 Database versus instance . . . 26
- 5.2 Oracle instance . . . 26
- 5.3 Oracle database . . . 30
- 5.4 Verwerking van een SQL statement . . . 37
- 5.5 Enkele queries op de data dictionary views . . . 38

**6 Het fysisch niveau: data** **40**
- 6.1 Bestandsorganisatie. . . . 40
- 6.2 Oracle database . . . 44

**7 Het fysisch niveau: indexen** **48**
- 7.1 Index-Sequentiële bestanden. . . . 48
- 7.2 $B^+$-Boom bestanden. . . . 51
- 7.3 Hashing. . . . 55
- 7.4 Indexen voor speciale doeleinden . . . 57
- 7.5 Gereedschap voor de oefeningen . . . 59

**8 Transaction management** **60**
- 8.1 ACID . . . 60
- 8.2 Serializeerbaarheid . . . 60
- 8.3 Two-Phase Locking . . . 63
- 8.4 Timestamp gebaseerde concurrentie controle . . . 67
- 8.5 Oefeningen . . . 68
- 8.6 Voorbeelden waarom concurrentie controle nodig is . . . 69
- 8.7 Isolatieniveau van een transactie . . . 70
- 8.8 Concurrency controle in Oracle . . . 71

**9 Databaseherstel** **73**
9.1 Recovery via reprocessing . . . . . 73
9.2 Recovery via rollback/rollforward . . . . . 73

**A Oracle SQL*Plus** **75**

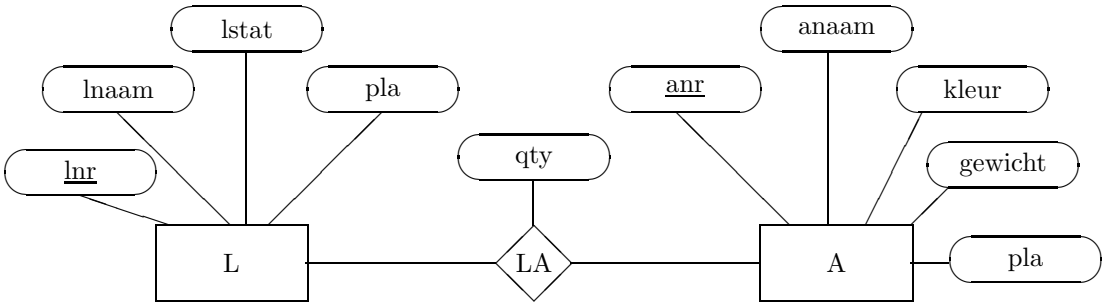
**B DBMS: oefeningen** **79**
B.1 Beschrijving van de gegevens . . . . . 79
B.2 Formatteringsaspecten van sql*plus . . . . . 79
B.3 Fysische organisatie . . . . . 79
B.4 Transaction management . . . . . 80
B.5 Fysische dataorganisatie . . . . . 82
B.6 DBA Views . . . . . 83
B.7 Van databankgegevens naar een xml-bestand en omgekeerd . . . . . 83
B.8 Information Retrieval . . . . . 84
B.9 Indexen . . . . . 85
B.10 sqldev . . . . . 86

# 1 SQL: aanvullingen

## 1.1 Versies

Er zijn verschillende dialecten van SQL. De twee belangrijkste standaards zijn ANSI SQL (jaren 80) en een aangepaste SQL-92 of SQL2 (van 1992), met bijvoorbeeld een HAVING in een subquery. Actueel wordt SQL2 uitgebreid met nieuwe elementen zoals recursie, triggers en objecten. Dit zou moeten leiden tot de nieuwe standaard SQL3.
De meeste leveranciers van database management systemen hebben elk hun eigen versie ontwikkeld. Hierin zijn steeds de mogelijkheden van de originele ANSI standaard aanwezig. Daarnaast bevatten ze een aantal elementen uit SQL2, maar nogal dikwijls lichtjes aangepast, bijvoorbeeld in de syntax. Meestal zijn er ook extra's bovenop SQL2, waaronder eventueel een aantal elementen uit de voorgestelde SQL3 standaard.
In dit hoofdstuk worden een aantal elementen besproken in verband met de *gegevensintegriteit* in een databank: het garanderen van bepaalde beperkingen en de beveiliging op het vlak van gebruikerstoegang.

## 1.2 Gegevens




leveranciers **L**

| lnr | lnaam | lstat | pla |
|---|---|---|---|
| L1 | Jan | 20 | Peulis |
| L2 | Marc | 10 | Hever |
| L3 | Sara | 30 | Hever |
| L4 | Luc | 20 | Peulis |
| L5 | Els | 30 | Schriek |

klanten **K**

| knr | knaam | kstat | pla |
|---|---|---|---|
| K1 | Sam | 40 | Peulis |
| K2 | Marc | 50 | Hever |
| K3 | Sara | 60 | Hever |
| K4 | Els | 20 | Peulis |
| K5 | Jan | 30 | Schriek |

lev-art **LA**

| lnr | anr | qty |
|---|---|---|
| L1 | A1 | 300 |
| L1 | A2 | 200 |
| L1 | A3 | 400 |
| L1 | A4 | 200 |
| L1 | A5 | 100 |
| L1 | A6 | 100 |
| L2 | A1 | 300 |
| L2 | A2 | 400 |
| L3 | A2 | 200 |
| L4 | A2 | 200 |
| L4 | A4 | 300 |
| L4 | A5 | 400 |

(gemid: 258.33)

artikels **A**

| anr | anaam | kleur | gewicht | pla |
|---|---|---|---|---|
| A1 | schroef | rood | 26 | Peulis |
| A2 | as | groen | 31 | Hever |
| A3 | bout | blauw | 17 | Grootlo |
| A4 | bout | rood | 24 | Peulis |
| A5 | moer | blauw | 12 | Hever |
| A6 | vijs | rood | 29 | Peulis |

## 1.3 Domeinen en tabellen

Een beperking leggen op de mogelijke waarden voor een attribuut, kan door een *domein* te declareren.

```
CREATE DOMAIN LD      CHAR(5);
CREATE DOMAIN AD      CHAR(6);
```

```
CREATE DOMAIN NAAM    CHAR(20);
CREATE DOMAIN STAD    CHAR(20);
CREATE DOMAIN STATUS  SMALLINT;
CREATE DOMAIN QTY     INTEGER;
CREATE DOMAIN MAS     INTEGER;

CREATE DOMAIN KLEUR   CHAR(6)  DEFAULT '???'
       CONSTRAINT geldige_kleuren
       CHECK ( VALUE IN
             ( 'rood', 'geel', 'blauw', 'groen', '???' ) ) ;
```

Zoals in het voorbeeld aangegeven, kan een beperking op de waarde van een attribuut gerealiseerd worden met een *domein* declaratie. Er is dan echter geen naam van het attribuut gekend; dit wordt opgevangen door het keyword VALUE te gebruiken
Deze domeinen kunnen dan gebruikt worden als datatype voor de attributen.

```
CREATE TABLE L  (  lnr      LD,
                   lnaam    NAAM,
                   lstat    STATUS  DEFAULT 10,
                   pla      STAD,
                   PRIMARY KEY ( lnr )  );
CREATE TABLE K  (  knr      LD,
                   knaam    NAAM,
                   kstat    STATUS  DEFAULT 20,
                   pla      STAD,
                   PRIMARY KEY ( knr )  );
CREATE TABLE A  (  anr      AD,
                   anaam    NAAM,
                   kleur    KLEUR  DEFAULT 'rood',
                   gewicht  MAS,
                   pla      STAD,
                   PRIMARY KEY ( anr )   );
```

## 1.4 Views

Een *view* is een virtuele tabel: het resultaat van een select krijgt een naam en kan vanaf dan bekeken worden als een nieuwe tabel. Het belang van een view heeft te maken met afschermen van informatie: bijvoorbeeld een gebruiker die alleen de rode artikels mag zien en geen andere. Een view kan gezien worden als een functie in wiskundige betekenis: gegeven de argumenten, bereken het resultaat. De inverse hiervan, gegeven het resultaat en bereken de argumenten, is wel moeilijk te realiseren.

```
CREATE VIEW rooddeel (anr, anaam, wt, stad)
    AS SELECT A.anr, A.anaam, A.gewicht, A.pla
       FROM A
       WHERE A.kleur = 'rood';
```

| anr | anaam | wt | stad |
|---|---|---|---|
| A1 | schroef | 26 | Peulis |
| A4 | bout | 24 | Peulis |
| A6 | vijs | 29 | Peulis |

De view `rooddeel(anr,anaam,wt,stad)` kan door de gebruiker als een nieuwe tabel gezien worden.

```
CREATE VIEW ah
  AS SELECT LA.anr, SUM(LA.qty) AS tothoev
     FROM LA
     GROUP BY LA.anr ;
```

| | | |
|---|---|---|
| A1 | 300 + 300 | 600 |
| A2 | 200 + 400 + 200 + 200 | 1000 |
| A3 | 400 | 400 |
| A4 | 200 + 300 | 500 |
| A5 | 100 + 400 | 500 |
| A6 | 100 | 100 |

Bij de uitvoering van een SQL statement dat naar een view refereert, wordt door het DBMS de view vervangen door haar definitie: het statement wordt "herschreven". Het resulterende herschreven statement wordt dan uitgevoerd; de view wordt dus niet eerst uitgerekend, deze blijft steeds virtueel. Bij een SELECT is herschrijven steeds mogelijk.

```
SELECT anr                          SELECT anr
FROM rooddeel                       FROM A
WHERE wt > 25;                      WHERE gewicht > 25 AND kleur = 'rood';
```

Er zijn "updatable" en "non-updatable" views. Wanneer er een $1-1$ correspondentie bestaat tussen de rij in de originele tabel en de view, kan de update uitgevoerd worden.

```
UPDATE rooddeel                     UPDATE A
SET wt = 454 * wt;                  SET gewicht = 454 * gewicht
                                    WHERE kleur = 'rood';
```

Er kunnen echter problemen optreden wanneer door de DBMS het SQL statement herschreven wordt.

```
UPDATE ah                           UPDATE LA
SET tothoev = tothoev + 1;          SET ???
```

## 1.5 DDL statements voor wijzigingen in structuren

### 1.5.1 Aanpassingen aan de tabelstructuur

De structuur van een tabel kan gewijzigd worden met de **alter table**-instructie.
Veronderstel dat de spelerstabel geen kolom bondsnummer zou hebben, dan zouden we deze kunnen toevoegen met de volgende instructie.

```
ALTER TABLE spelers
ADD bdnr char(4)
```

We kunnen een kolom ook verwijderen.
We verwijderen ter illustratie de kolom huisnummer.

```
ALTER TABLE spelers
DELETE hnr
```

We kunnen het datatype van kolommen wijzigen. Hiervoor dienen alle waarden van de kolom te voldoen aan de regels van het nieuwe datatype. Of nog, alle waarden van de kolom moeten te transformeren zijn naar het nieuwe datatype.
Het voorbeeld verandert het datatype van de kolom bondsnummer in integer.

```
ALTER TABLE spelers
MODIFY bdnr integer
```

### 1.5.2 Aanpassingen aan beperkingen

Alle bovenvermelde beperkingen kunnen een naam krijgen. Dit wordt gerealiseerd door de beperking te beginnen met het keyword CONSTRAINT gevolgd door een naam, zoals bij de definitie van het domein KLEUR.

```
CREATE TABLE A ( anr     char(5) CONSTRAINT sleu PRIMARY KEY,
                 anaam   char(20),
                 kleur   char(8) CONSTRAINT kleurok
                     CHECK ( kleur IN ('rood','geel','groen','blauw') ),
                 gewicht integer  CONSTRAINT gewichtok
                     CHECK ( gewicht < 100 ),
                 pla     char(20),
                 CONSTRAINT blauwgewicht
                      CHECK ( kleur = 'blauw' OR gewicht > 20 )   );
```

Met behulp van deze namen kan bij aanpassingen naar de juiste beperking gerefereerd worden.
Weglaten: `ALTER TABLE A  DROP CONSTRAINT gewichtok;`
Terug bijvoegen: `ALTER TABLE A  ADD CONSTRAINT gewichtok CHECK ( gewicht < 100 );`
Merk op dat de beperking nu niet attribuut- maar tuple gebaseerd is. Een attribuut-gebaseerde CHECK beperking achteraf bijvoegen, kan niet. Om dit te realiseren moet men met domeinen werken:

```
CREATE DOMAIN MAS integer
             CONSTRAINT gewichtok  CHECK ( VALUE < 100 );
ALTER DOMAIN MAS  DROP CONSTRAINT gewichtok;
ALTER DOMAIN MAS  ADD CONSTRAINT gewichtok CHECK ( VALUE < 100 );
```

Ook assertions kunnen verwijderd worden:

```
DROP ASSERTION niettezwaar;
```

# 2 Analyse van gegevens: normalisatie

Normalisatie wordt gebruikt om een aantal problemen op het gebied van gegevensonafhankelijkheid en redundantie op te lossen. De normalisatie stappen worden besproken aan de hand van een voorbeeld. Het nut van normalisatie wordt duidelijk gemaakt door telkens te wijzen op de problemen (qua gebruik en onderhoud van de database), welke er kunnen bestaan wanneer de gegevens in een bepaalde vorm voorkomen.
Tabel 1 geeft als vertrekbasis de niet genormaliseerde records, waarin een aantal gegevens (attributen) zijn opgenomen i.v.m. de programmeurs (entiteit) van het bedrijf.

PROGRAMMEUR

| PROGR. NR. | PROGR. NAAM | TEAM IDENT. | TEAM LOKAL. | TEAM NAAM | SPEC. KODE | SPEC. NAAM | SPEC. GRAAD |
|---|---|---|---|---|---|---|---|
| 1099 | JANSEN | 15 | B116 | BSP | 05<br>06 | ONTWERP<br>ANALYSE | A+<br>A |
| 2734 | JACOBS | 23 | K32 | STIV | 03<br>04 | TEST<br>KONTROL | B+<br>A |
| 4372 | DE WIT | 22 | K32 | ADOC | 03<br>02<br>04 | TEST<br>IMPLEM.<br>KONTROL | A<br>A+<br>C |
| 7299 | DAANS | 15 | B116 | BSP | 02 | IMPLEM | B+ |

Tabel 1: De niet genormaliseerde entiteit

Er gelden volgende externe beperkingen: elke programmeur kan slechts in 1 team werken en elk team heeft slechts 1 lokatie.
Een attribuut of een reeks attributen word(t)(en) de kandidaatsleutel genoemd, wanneer de waarden uniek de records identificeren; men zegt dat de sleutel de entiteit identificeert. De primaire sleutel is deze welke gekozen wordt om de entiteit te identificeren.
In het voorbeeld van tabel 1 zijn de attributen (data elements):

```
programmeur { progr.nr., progr.naam,
                team ident., team lokal., team naam,
                       {spec.kode, spec.naam, spec.graad} }
```

De voor hand liggende primaire sleutel is *progr.nr.*.

## 2.1 Eerste normaalvorm.

Een relatie (entiteit) in eerste normaalvorm is een tabel, waarin slechts één waarde op de intersektie van kolommen (attributen) en rijen (tuples) mag voorkomen. Groepen van waarden (repeating groups) zijn niet toegelaten.
Bekijken we tabel 1. Voor een gegeven waarde van de primaire sleutel kunnen de waarden van de secundaire (niet-sleutel) attributen niet uniek gedefinieerd worden. Dit geeft aanleiding tot volgende anomalieën:

- updaten : verandert bijv. De Wit van team, moeten we de team gegevens wijzigen in alle tuples, waarin De Wit voorkomt. Updaten zal dus veel redundant zoekwerk veroorzaken;
- er is geen garantie dat aan de gegeven beperkingen voldaan is, bijvoorbeeld dat elke programmeur tot slechts één team behoort;
- insertion: het is onmogelijk een nieuwe specialisatie toe te voegen zonder dat er een programmeur bestaat die deze specialisatie bezit, (bijvoorbeeld de specialisatie “projectbeheer” met kode 07);

- deletion: onderstel dat Jansen verwijderd wordt. Als gevolg daarvan verdwijnen alle gegevens van de specialisatie 05 en 06.

De oorspronkelijke entiteit bevat in feite gegevens over de programmeurs. spec.kode, spec.naam, spec.graad vormt de repeating group t.o.v. de attributen i.v.m. programmeur.

```
EERSTE NORMALISATIESTAP: VERWIJDER DE ZICH HERHALENDE DEELVERZAMELINGEN
                         (REPEATING GROUPS) IN EEN NIEUWE ENTITEIT.
```

Daartoe introduceren we een tweede entiteit *progr.spec.* In de oorspronkelijke entiteit *programmeur* zijn de repeating groups verwijderd. De twee nieuwe entiteiten worden in tabel 2 voorgesteld. Hierin vormen de dubbel onderlijnde attributen de primaire sleutel voor deze entiteiten.

entiteit PROGRAMMEUR

| PROGR. NR. | PROGR. NAAM | TEAM IDENT. | TEAM LOKAL. | TEAM NAAM |
|---|---|---|---|---|
| 1099 | JANSEN | 15 | B116 | BSP |
| 2734 | JACOBS | 23 | K32 | STIV |
| 4372 | DE WIT | 22 | K32 | ADOC |
| 7299 | DAANS | 15 | B116 | BSP |

entiteit PROGR.SPEC.

| PROGR. NR. | SPEC. KODE | SPEC. NAAM | SPEC. GRAAD |
|---|---|---|---|
| 1099 | 05 | ONTWERP | A+ |
| 1099 | 06 | ANALYSE | A |
| 2734 | 03 | TEST | B+ |
| 2734 | 04 | KONTROL | A |
| 4372 | 03 | TEST | A |
| 4372 | 02 | IMPLEM. | A+ |
| 4372 | 04 | KONTROL | C |
| 7299 | 02 | IMPLEM | B+ |

Tabel 2: Eerste normaalvorm

We moeten de occurrences van de entiteit *progr.spec.* uniek kunnen identificeren. Het verwijderen van de zich herhalende deelverzamelingen resulteert dus steeds in een samengestelde sleutel (compound key) voor de nieuwe entiteit, namelijk *prog.nr* en *spec.kode*.

## 2.2 Tweede normaalvorm.

Bekijken we de entiteit *progr.spec.*, dan zien we o.a. volgende anomalieën opduiken:

- redundantie, bijvoorbeeld spec.kode 03 met naam TEST;
- verwijderen van een programmeur, doet de specialisatie gegevens verdwijnen, indien hij de enige was in de beschouwde specialiteit, bijvoorbeeld bij het verwijderen van 1099 verdwijnen de specialisaties 05 en 06.

De oorzaak van deze anomalieën is het feit dat niet alle secundaire attributen functioneel afhankelijk zijn van de volledige samengestelde primaire sleutel.
Definiëren we het begrip functionele afhankelijkheid:

```
beschouw: de attributen X en Y van de entiteit E.
             Y is functioneel afhankelijk van X
```

```
(of X identificeert Y) in E als, op elk ogenblik,
met iedere waarde van X niet meer dan een waarde
van Y in verband staat onder E.
```

In het voorbeeld is het attribuut *spec.naam* in de entiteit *progr.spec.* niet functioneel afhankelijk van de volledige compound key, doch alleen van het key-attribuut *spec.kode*.

```
TWEEDE NORMALISATIESTAP: VERWIJDER DE ATTRIBUTEN DIE SLECHTS
                FUNCTIONEEL AFHANKELIJK ZIJN VAN EEN GEDEELTE
                VAN DE PRIMAIRE SLEUTEL IN EEN NIEUWE ENTITEIT.
```

We splitsen de entiteit *progr.spec.* dan ook op in twee entiteiten. Zie tabel 3. De entiteit *programmeur* staat reeds in de 2NV (bevat namelijk geen samengestelde sleutel).

entiteit PROGR.SPEC.

| PROGR. NR. | SPEC. KODE | SPEC. GRAAD |
|---|---|---|
| 1099 | 05 | A+ |
| 1099 | 06 | A |
| 2734 | 03 | B+ |
| 2734 | 04 | A |
| 4372 | 03 | A |
| 4372 | 02 | A+ |
| 4372 | 04 | C |
| 7299 | 02 | B+ |

entiteit SPEC.

| SPEC. KODE | SPEC. NAAM |
|---|---|
| 02 | IMPLEM. |
| 03 | TEST |
| 04 | KONTROL |
| 05 | ONTWERP |
| 06 | ANALYSE |

Tabel 3: Tweede normaalvorm

## 2.3 Derde normaalvorm.

Bekijken we de entiteit *programmeur* (tabel 2) van nabij, zien we de volgende anomalieën:

- insertion: een nieuw team kan niet geïntroduceerd worden tenzij er ten minste een programmeur lid is;
- deletion: indien een programmeur van team verandert of verwijderd wordt, gaan de gegevens van het team verloren, indien hij het enige teamlid was, wanneer bijvoorbeeld De Wit naar team 15 verhuist, dan verdwijnt de informatie over team 22;
- de beperking dat elk team slechts één lokatie heeft, is niet gegarandeerd;
- updaten van de team lokatie of het team nummer geeft aanleiding tot redundant zoekwerk, als bijvoorbeeld team 15 van lokaal verandert dan moet op twee plaatsen de waarde B116 aangepast worden.

De oorzaak is de transitieve afhankelijkheid.

```
Stel : X, Y en Z zijn attributen van de entiteit E:
          Y is funktioneel afhankelijk van X
          Z is funktioneel afhankelijk van Y
dan is: Z is transitief afhankelijk van X.
```

```
DERDE NORMALISATIESTAP: VERWIJDER DE ATTRIBUTEN, WELKE VOLLEDIG
       FUNCTIONEEL AFHANKELIJK ZIJN VAN EEN ANDER ATTRIBUUT IN DE
       ENTITEIT (DAT GEEN PRIMAIRE SLEUTEL IS), NAAR EEN NIEUWE ENTITEIT.
```

entiteit PROGRAMMEUR

| PROGR. NR. | PROGR. NAAM | TEAM IDENT. |
|---|---|---|
| 1099 | JANSEN | 15 |
| 2734 | JACOBS | 23 |
| 4372 | DE WIT | 22 |
| 7299 | DAANS | 15 |

entiteit TEAM

| TEAM IDENT. | TEAM LOKAL. | TEAM NAAM |
|---|---|---|
| 15 | B116 | BSP |
| 23 | K32 | STIV |
| 22 | K32 | ADOC |

Tabel 4: Derde normaalvorm

In de entiteit *programmeur* zijn de attributen *team lokal.* en *team naam* funktioneel afhankelijk van het attribuut *team ident.*, wat op zijn beurt afhankelijk is van de primaire sleutel *progr.nr.*. De entiteit *programmeur* wordt in twee entiteiten opgesplitst. Zie tabel 4.
Nieuwe teams kunnen nu zonder probleem aan de tweede tabel van tabel 4 toegevoegd worden zonder dat hiervoor een programmeur reeds lid moet zijn.

## 2.4 Normalisatie: een eindpunt?

Normalisatie is een stap-voor-stap omkeerbaar proces van vervanging van een gegeven verzameling van relaties (entiteiten) door opeenvolgende verzamelingen, waarin de relaties steeds een meer eenvoudige en meer effektieve struktuur hebben. Deze vereenvoudiging gaat echter gepaard met een verhoging van de *processing complexity*. Verschillende file accessen kunnen nodig zijn, daar waar in de niet-genormaliseerde struktuur één access volstond.
Het inbouwen van een zekere redundantie kan, i.v.m. performantie en eisen betreffende responstijden noodzakelijk zijn. Maar dan spreken we niet meer over de logische struktuur, maar over de opslagstruktuur. De logische struktuur kan, door eisen betreffende de toegang tot gegevens, niet zomaar overgenomen worden in de opslagstruktuur. Dit heeft te maken met de verschillende doelstellingen: normalisatie leidt tot een eenvoudig logisch model, niet noodzakelijk tot een efficiënt model.
De volgende stap is er dan ook op gericht een compromis te vinden tussen de *file complexity* (ingebouwde redundantie) en *processing complexity*. Het gevonden compromis is bepalend voor het prestatievermogen van de database, bijvoorbeeld de snelheid waarmee informatie kan teruggevonden worden.

## 2.5 Een tweede voorbeeld op normalisatie.

Een bestelformulier voor boeken (figuur 1) bevat volgende elementen:

- een opdrachtnummer
- een klantnummer, de naam en de woonplaats van de klant
- per besteld boek wordt aangegeven: het ISBN, de titel, de prijs/eenheid en het aantal besteld
- de datum en de totaalprijs.

Om tot de eerste normaalvorm te komen, moet de repeating group (de verschillende bestelde boeken) in een tweede entiteit geplaatst worden:

| | |
|---|---|
| ORDER | opdrachtnr, klantnr, klantnaam, woonplaats, datum, totaalprijs |
| REGEL | opdrachtnr, ISBN, titel, prijs/eenheid, aantal |

Bij het opstellen van de tweede normaalvorm merkt men op dat in de entiteit REGEL titel en prijs/eenheid functioneel afhankelijk zijn van slechts een gedeelte van de sleutel. Deze entiteit wordt dus gesplitst in twee entiteiten:

opdrachtnr: 23 klantnr: 1063

datum: 31/05/2003 naam: Appeltans

woonplaats: Diest

Bestellingen:

| | | | | |
|---|---|---|---|---|
| 1. | Bestandsorganisatie | 90-6233-075-4 | 5 | 16 |
| 2. | Productions and Operations Management | 0-13-725268-4 | 1 | 40 |

Totaal bedrag: 120 euro

Figuur 1: Bestelformulier

| | |
|---|---|
| REGEL | opdrachtnr, ISBN, aantal |
| BOEK | ISBN, titel, prijs/eenheid |

Bij de derde normaalvorm moet de entiteit ORDER nog gesplitst worden omdat klantnaam en woonplaats functioneel afhankelijk zijn van klantnr, dat op zijn beurt functioneel afhankelijk is van de sleutel opdrachtnr:

| | |
|---|---|
| ORDER | opdrachtnr, klantnr, datum, totaalprijs |
| KLANT | klantnr, klantnaam, woonplaats |

## 2.6 Oefening.

Op een rapport staan volgende gegevens vermeld:

- datum van de deliberatie; klas, opleiding, minor;
- naam en voornaam van de student;
- per vak: naam, gewichtsfaktor, behaalde punten op 20;
- behaalde procent en graad.

Breng deze gegevens in 3NV.

# 3 Information retrieval: aanvullingen

## 3.1 Indexeren van tekst

Verschillende soorten: CONTEXT en CTXCAT tekstindices.

```
create index boek_content on boek(tekst)
indextype is ctxsys.context;

create index boek_ctxcat on boek(tekst)
indextype is ctxsys.ctxcat;
```

Voordeel: synchronisatie van de gegevens in de basistabel en in de tekstindexen

Op basis van zo'n tekstindex: mogelijkheden van tekst zoeken sterk uitgebreid:
sleutelwoord contains (context index) en catsearch (ctxcat index) in de where van een select
Daarnaast: CTXRULE: bouwen van een inhoudgebaseerde document classificatie toepassing

Bij een content index worden vier tabellen gecreëerd:

```
select table_name from user_tables;
```

```
TABLE_NAME
-----------------
DR$BOEK_CONTENT$K
DR$BOEK_CONTENT$N
DR$BOEK_CONTENT$R
DR$BOEK_CONTENT$I
```

De K tabel bevat twee attributen: een DOCID en een TEXTKEY (een ROWID). Voor elke rij in boek is er een rij in deze K tabel.
De N tabel (met attributen NLT_DOCID en NLT_MARK) bevat geen rijen.
De R tabel (met attributen ROW_NO en DATA) bevat 22 rijen.
De I tabel bevat verschillende attributen:

```
Name                                      Null?    Type
----------------------------------------- -------- ----------------------------
TOKEN_TEXT                                NOT NULL VARCHAR2(64)
TOKEN_TYPE                                NOT NULL NUMBER(3)
TOKEN_FIRST                               NOT NULL NUMBER(10)
TOKEN_LAST                                NOT NULL NUMBER(10)
TOKEN_COUNT                               NOT NULL NUMBER(10)
TOKEN_INFO                                         BLOB
```

Deze tabel bevat 60 rijen.
Na aanpassingen in de basistabel, moet een content index hersynchroniseerd worden:

```
begin     ctx_ddl.sync_index('boek_content');     end;
```

**Toepassing: index set** . Vroeger: problemen bij een gecombineerde where met non-tekst en tekst elementen in de voorwaarde:

```
select titel, score(5) from boek
where auteur = 'mie groen'
      and contains(tekst, 'appelen', 5) > 0;
```

Oplossing: gebruikmaken van index set: de indexen in deze verzameling kunnen gemaakt zijn op de gestructureerde relationele kolommen en op de tekstkolommen.
Om zo'n index set te creëren, wordt gebruik gemaakt van de CTX_DDL package.

```
execute CTX_DDL.CREATE_INDEX_SET('boeken');
execute CTX_DDL.ADD_INDEX('boeken', 'auteur');
execute CTX_DDL.ADD_INDEX('boeken', 'datum');
```

Na het toevoegen van de non-tekst indexen kan een ctxcat index gecreëerd worden:

```
create index boek_ctxcat on boek(tekst)
indextype is ctxsys.ctxcat
parameters ('index set boeken' );
```

De resultaten van de text search kunnen nu geordend worden op basis van de datum:

```
select titel, score(5) from boek
where catsearch(tekst, 'appelen', 'auteur = ''mie groen''
                order by datum desc) > 0;
```

## 3.2 Tonen van de tekst

Hiervoor kunnen PL/SQL procedures uit o.a. ctx_doc package gebruikt worden.

```
1  declare a clob;
2          totaal integer;
3          i integer := 1;
4          n number := 80;
5          l varchar2(80);
6  begin
7     ctx_doc.filter('boek_content',1,a,FALSE);
8     totaal := dbms_lob.getlength(a);
9     while ( i < totaal ) loop
10         dbms_lob.read(a,n,i,l);
11         dbms_output.put(l);
12         i := i+n;
13    end loop;
14    dbms_output.new_line;
15    dbms_lob.freetemporary(a);
16 end;
```

Om bovenstaande procedure te parametriseren kan gebruikt gemaakt worden van een bind variabele:

```
var tnr number
exec :tnr := 2
```

De eerste lijn wordt dan ctx_doc. filter ('boek_content', :tnr, a,FALSE);

## 3.3 Parameters bij een context index

**Stopwoorden.** Creatie van een stopwoorden lijst:

```
begin
     ctx_ddl.create_stoplist('stopvz', 'BASIC_STOPLIST');
     ctx_ddl.add_stopword('stopvz', 'het');
     ctx_ddl.add_stopword('stopvz', 'de');
     ctx_ddl.add_stopword('stopvz', 'een');
     ctx_ddl.add_stopword('stopvz', 'om');
     ctx_ddl.add_stopword('stopvz', 'te');
     ctx_ddl.add_stopword('stopvz', 'via');
     ctx_ddl.add_stopword('stopvz', 'voor');
     ctx_ddl.add_stopword('stopvz', 'met');
     ctx_ddl.add_stopword('stopvz', 'bij');
     ctx_ddl.add_stopword('stopvz', 'en');
     ctx_ddl.add_stopword('stopvz', 'er');
```

```
    ctx_ddl.add_stopword('stopvz', 'zo');
    ctx_ddl.add_stopword('stopvz', 'niet');
    ctx_ddl.add_stopword('stopvz', 'wat');
    ctx_ddl.add_stopword('stopvz', 'ook');
end;
```

Gebruik van de stopwoorden lijst bij het maken van de index:

```
create index boek_content on boek(tekst)
       indextype is ctxsys.context
       parameters ('STOPLIST stopvz');
```

In de I tabel zijn de stopwoorden niet meer opgenomen; er zijn nog slechts 47 rijen.
Verwijderen van een stopwoorden lijst:

```
  begin    ctx_ddl.drop_stoplist('stopvz');    end;
```

**Stamwoorden.** Aangeven dat stammen op basis van de nederlandse taal gemaakt worden:

```
begin
    ctx_ddl.create_preference('vlaams','BASIC_LEXER');
    ctx_ddl.set_attribute('vlaams','index_stems','DUTCH');
end;
```

Index met stamwoorden:

```
create index boek_content on boek(tekst)
       indextype is ctxsys.context
       parameters ('LEXER vlaams STOPLIST stopvz');
```

De I tabel bevat 78 rijen; naast *tomaten* is ook het woord *tomaat* opgenomen.
Verwijderen van een lexer preferentie lijst:

```
    begin    ctx_ddl.drop_preference('vlaams');    end;
```

Om bij het zoeken stamwoorden en fuzzy matching te activeren, moet de wordlist preferentie gebruikt worden. Het enige type dat beschikbaar is, is BASIC_WORDLIST.
Belangrijkste attributen zijn:

- stemming: welke taal voor stemming gebruikt moet worden, bijv. `DUTCH`;
- fuzzy_match: welke fuzzy matching cluster gebruikt moet worden, bijv. `DUTCH`;

```
begin
    ctx_ddl.create_preference('stamvlaams', 'BASIC_WORDLIST');
    ctx_ddl.set_attribute('stamvlaams','FUZZY_MATCH','DUTCH');
    ctx_ddl.set_attribute('stamvlaams','STEMMER','DUTCH');
end;
```

Bij de creatie van de index:

```
create index boek_content on boek(tekst)
       indextype is ctxsys.context
       parameters ('Wordlist stamvlaams');


select titel,score(1) from boek
where contains(tekst,'fuzzy(beren, 50, 5, weight)',1) > 0
```

Vroegere syntax: contains(tekst, '?beren', 1)
fuzzy heeft vier parameters:

term : het woord (minstens 3 letters) waarop de fuzzy expansie uitgevoerd wordt; deze expansie bevat enkel woorden die in de index gevonden worden;

score : geëxpandeerde termen met een similariteitscore lager dan deze waarde worden niet beschouwd (tussen 1 en 80, default is 80);

numresults : maximum aantal termen dat in de expansie van de gegeven term kan beschouwd worden (tussen 1 en 5000, default is 100);

weight : resultaten worden gewogen volgens de similariteitscores (alternatieve waarde is no-weight).

Resultaat:

```
fruit eten          43
groenten            13
```

Wanneer een minimum score van 70 gevraagd wordt, wordt alleen het eerste document gevonden met een score van 39.
Wanneer de databank teksten in verschillende talen bevat, kunnen de stammen op basis van een automatisch gedetecteerde taal gemaakt worden:

```
begin
ctx_ddl.create_preference('vlaams','AUTO_LEXER');
ctx_ddl.set_attribute('vlaams','index_stems','YES');
end;
```

Index met stamwoorden:

```
create index boek_content on boek(tekst)
       indextype is ctxsys.context
       parameters ('LEXER vlaams STOPLIST stopvz');
```

De I tabel bevat 115 rijen.

### 3.3.1 Locatie van de tekst.

Bij een context index zijn er drie mogelijke locaties. Het type wordt bij de creatie van de index aangegeven met de DATASTORE parameter:

```
create index boek_content on boek(docs)
     indextype is ctxsys.context
     parameters ('DATASTORE CTXSYS.DEFAULT_DATASTORE');
```

- DIRECT_DATASTORE (of DEFAULT_DATASTORE): de tekst wordt volledig in de tabel opgenomen; mogelijke types voor de tekstkolom zijn VARCHAR2, CLOB, BLOB, CHAR, of BFILE; elke rij komt overeen met één document.
- FILE_DATASTORE: de tekst wordt extern in een bestand van het besturingssysteem bewaard; de bestandsnaam wordt in de tabel opgenomen.
- URL_DATASTORE: de tekst wordt extern in bestanden opgeslagen, die op het intranet of het internet gelokaliseerd zijn; de bijhorende URL wordt in de tabel opgenomen.

Bij een CTXCAT of CTXRULE index, is alleen de eerste techniek (volledig in de tabel) mogelijk.

**Tekst in externe bestanden.** Creatie van de tekst tabel en bijhorende index:

```
create table txtpdf
(
        pnr integer,
        titel varchar2(40),
```

```
        filenaam varchar2(40),
        constraint pdf_pk primary key (pnr)
);
insert into txtpdf values (10,'oText','/home/hcr/t/4sp/les.pdf');
insert into txtpdf values (20,'infret','/home/hcr/t/4cc/les.pdf');
create index pdf_ctx on txtpdf(filenaam)
    indextype is ctxsys.context
    parameters('datastore ctxsys.file_datastore');
```

Zoeken naar het woord *vlaams*:

```
select titel,score(1) from txtpdf
where contains(filenaam,'vlaams',1) > 0
```

In plaats van de volledige padnaam in de tabel op te nemen, kan een PATH attribuut bij de preferentie gedefinieerd worden:

```
begin
 ctx_ddl.create_preference('padpl','FILE_DATASTORE');
 ctx_ddl.set_attribute('padpl','PATH','/home/hcr/t/4sp:/home/hcr/t/4cc');
end;
```

**Tekst in webpagina's.** Creatie van de tekst tabel en bijhorende index:

```
create table txtweb
(
    wnr integer,
    titel varchar2(20),
    urlnaam varchar2(80),
    constraint web_pk primary key (wnr)
);
insert into txtweb values (17, 'db_nv',
                'http://spline/~hcr/db/nv.html');
insert into txtweb values (27, 'db_nvvb',
                'file:/home/hcr/public_html/db/nvboek.html');

create index web_ctx on txtweb(urlnaam)
     indextype is ctxsys.context
     parameters('datastore ctxsys.url_datastore
                    filter ctxsys.null_filter
                    section group htmgroep');
```

**Filteren van de tekst.** Heel wat documenten: beschikbaar in geformatteerde vorm (bijv. PDF, MS-Word, ...). Om zo'n documenten te indiceren moeten ze gefilterd worden naar tekst. Hiervoor is default de ctxsys. auto_filter beschikbaar: deze filter detecteert automatisch het formaat van het document en doet de omzetting naar tekst.
Voor pure tekst of HTML-pagina's is zo'n filtering niet nodig. Hiervoor kan bij de creatie van de index de NULL_FILTER gespecificeerd worden:

```
create index urlindex on webpag(htmlfile)
    indextype is ctxsys.context
    parameters('filter ctxsys.null_filter
                section group ctxsys.html_section_group');
```

Daarnaast zijn door gebruik te maken van de interne structuur van HTML document secties gedefinieerd en geïndiceerd. Dit laat toe om het bereik van een query te verengen, bijv. zoeken naar documenten die een bepaalde term in een specifieke sectie hebben (bijv. in `TITLE`).

Er zijn systeemgedefinieerde sectie definities voor HTML en XML. Het is ook mogelijk om zelf sectie definities te definiëren.

```
begin
   ctx_ddl.create_section_group('htmgroep', 'HTML_SECTION_GROUP');
   ctx_ddl.add_field_section('htmgroep', 'titel', 'TITLE');
   ctx_ddl.add_zone_section('htmgroep', 'hoofding', 'H1');
   ctx_ddl.add_zone_section('htmgroep', 'hoofding', 'H2');
   ctx_ddl.add_zone_section('htmgroep', 'hoofding', 'H3');
   ctx_ddl.add_zone_section('htmgroep', 'tekst', 'P');
end;

begin      ctx_ddl.drop_section_group('htmgroep');      end;
```

Zone sectie dat alle `CONTENT` attributen voor de `META` tag met `NAME` waarde gelijk aan *author* indiceert:

```
begin
    ctx_ddl.add_zone_section('htmgroep', 'auteur', 'meta@author');
end
```

Mogelijke zoekvraag: contains(urlnaam, 'hcr within auteur',1)

**Zone section.** Extract uit het `nv.html` document:

```
      <meta name="author" content="hcr" />
<h1 class="blauw"> Tweede normaalvorm</h1>
<h3 class="rood">Problemen:</h3>
```

Zoeken naar een woord dat in een hoofding staat:

```
select titel,score(1) from txtweb
where contains(urlnaam,'normaalvorm within hoofding', 1) > 0
```

Er kunnen meerdere zone sections in een document voorkomen en deze worden als aparte onderdelen beschouwd. Als gevolg hiervan heeft volgende vraag geen resultaat:

```
select titel,score(1) from txtweb
where contains(urlnaam,'(normaalvorm and problemen) within hoofding', 1) > 0
```

Bij een gelijkaardige vraag zonder de within operator worden wel documenten gevonden.

**Field section.** Bij het zoeken naar het woord normalisatie:

```
select titel,score(1) from txtweb
where contains(urlnaam,'normalisatie',1) > 0
```

is er geen resultaat, alhoewel het woord in de titel van beide webpagina's voorkomt.
De reden hiervoor is dat `TITLE` als een *field section* toegevoegd is aan de htmgroep. Zo'n *field section* wordt als een sub-document geïndiceerd, apart van de rest van het document.
Op die manier is een beperkende zoekvraag mogelijk:

```
select titel,score(1) from txtweb
where contains(urlnaam,'normalisatie within titel',1) > 0
```

Wanneer de tekst in zo'n *field section* toch als onderdeel van het omvattende document moet geïndiceerd worden, moet een vierde parameter met waarde **TRUE** toegevoegd worden:

```
ctx_ddl.add_field_section('htmgroep', 'titel', 'TITLE', TRUE);
```

## 3.4 Thesaurus

- een gereedschap voor vocabulaire controle
- bevat allerlei soorten informatie om indexers en zoekers te leiden naar gebruik van juiste termen zodat de kwaliteit van opvragen verbetert
- aangeven van voorkeurtermen: welke termen indexers en zoekers mogen gebruiken
  op die manier wordt het vocabularium beperkt zodat het gemakkelijker te voorspellen is welke woorden gebruikt zijn om een concept te indiceren
- aangeven van non-voorkeurtermen: woorden welke beter niet gebruikt worden
  het is mogelijk een non-voorkeurterm op te zoeken en dan de voorkeurterm te krijgen welke in de plaats kan gebruikt worden
  bij het opzoeken van een voorkeurterm in de thesaurus krijgt men ook de non-voorkeurtermen te zien: dit geeft een duidelijker idee omtrent de betekenis van een term
- semantische relaties: het met elkaar linken van verschillende voorkeurtermen: dit maakt de betekenis van een term duidelijker en kan de gebruiker helpen bij het kiezen van de juiste term
- bereik notitie is een definitie van een term en kan de gebruiker ook helpen bij het kiezen van de juiste term

Twee types { enumeratief: opsomming van alle voorkeurtermen
synthetisch: met regels om voorkeurtermen te construeren uit componenten

### 3.4.1 Constructie: verzamelen van termen

Bronnen:

- bestaande lijsten van termen: andere thesauri, indexen, woordenlijsten, ...
- teksten waaruit termen kunnen gehaald worden: titels, abstracts, volledige teksten met geïndiceerde elementen
- mensen: domeinspecialisten

Welke type termen:

- algemeen genoeg zodat het als index kan gebruikt worden voor een aantal documenten
  bijv. eigennamen worden best niet opgenomen in een thesaurus
- maar ook niet te algemeen, zodat het een index zou zijn in het overgrote deel van de documenten binnen het onderwerp van de thesaurus
  bijv. *nieuws* in een thesaurus om nieuwsitems te indiceren
- zelfstandige naamwoorden in volgende vorm

| | |
|---|---|
| meervoud bij telbare dingen | emmers |
| enkelvoud bij *massa* woorden | hout |
| enkelvoud voor processen, eigenschappen, voorwaarden | koeling, gewicht, armoede |
| geen omkeringen | radar antenne (ipv. antenne, radar) |
| met weglaten van voorzetsels | hogeschool wetenschap kunst |
| met weglaten van leestekens, diacritica, speciale tekens en afkortingen | cooperatieve programmas |

Termen met meer dan één betekenis:
een homograaf is een uitdrukking met dezelfde spelling maar met een andere betekenis;
een thesaurus maakt onderscheid tussen homografen door bijvoorbeeld een qualifier tussen haakjes toe te voegen *port (wine)*
(omwille van bovenstaande suggestie kunnen de haakjes ook weggelaten worden)

Introduceren van nieuwe termen:

1. brede concepttermen, bruikbaar bij brede zoekopdrachten
2. structurele termen: helpen de structuur van semantische relaties te verduidelijken
3. termen uit non-tekstueel materiaal, bijv. uit een foto of een prent

### 3.4.2 Voorkeurtermen en non-voorkeurtermen

Na het verzamelen van de woordenlijst moeten de equivalente termen bepaald worden: voor indexeren en zoeken worden equivalente termen beschouwd dezelfde betekenis te hebben en kunnen ze weergegeven worden door één voorkeurterm.
Equivalent omwille van:

- spelling: burokratie of bureaucratie
- synoniemen: automatisatie of mechanisatie
- quasi-synoniemen: in het reële leven duiden ze verschillende dingen aan, maar voor indexeren en zoeken is het beter ze te groeperen:
  * termen met overlappende betekenissen: school en hogeschool
  * term waarvan het bereik in dat van een andere term vervat is: ijzer en metaal
  * tegengestelden, wanneer de ene term waarschijnlijk relevant is bij een zoekopdracht voor de tegengestelde term: transparant en ondoorzichtig

| | |
|---|---|
| USE | non-voorkeurterm verbinden met voorkeurterm |
| UF (used for) | omgekeerde relatie |

Bijv. met de voorkeurterm *school*:

| | |
|---|---|
| hogeschool<br>USE school | school<br>UF hogeschool |

Voorkeurtermen worden gekozen op basis van dagdagelijks gebruik, breedte, minder dubbelzinnigheid, beknoptheid, meervoudsvorm (voor telbare dingen), consistensie, ...
Het is soms aangewezen om in plaats van een non-voorkeurterm meerdere voorkeurtermen samen te gebruiken. Bij de USE worden de verschillende voorkeurtermen vermeld en de UF referentie wordt gewoonlijk speciaal gemarkeerd:

| | | |
|---|---|---|
| sneeuwmobielen<br>USE voertuigen+sneeuw | sneeuw<br>UF+ sneeuwmobielen | voertuigen<br>UF+ sneeuwmobielen |

### 3.4.3 Semantische relaties

Meestal aanduiding van een hiërarchische relatie tussen voorkeurtermen
broader term (BT) en narrower term (NT):

1. soort: alle elementen van de klasse aangeduid door $B$ zitten ook in de klasse aangeduid door $A$ : $A$ is BT en $B$ dus NT
   bijv. *beesten* is een bredere term voor *katten*
   maar *huisdieren* is geen bredere term voor *katten*
2. klasse/lid: de NT benoemt een klassse met slechts één lid
   bijv. *hogescholen* is een bredere term voor *hogeschool voor wetenschap & kunst*
3. hiërarchisch geheel/onderdeel: alles wat in $B$ (NT) zit is onderdeel van iets dat in $A$ (BT) zit
   bijv. *hoofd* is een bredere term voor *neus* (in een medische thesaurus)
   maar *bossen* is geen bredere term voor *bomen*
4. geografisch geheel/onderdeel
   bijv. *Europa* is een bredere term voor *Zwitserland*
   maar *Zwitserland* is geen bredere term voor *meer van Genève*

**BT en NT referenties**

1. BT en NT zijn gewoonlijk inverse relaties

   Als | pennen<br>BT schrijfmaterialen | dan | schrijfmaterialen<br>NT pennen

2. polyhiërarchisch: een term kan meer dan één onmiddellijk bovenliggende bredere term hebben

   | sociale psychologie<br>BT psycholgie<br>BT sociologie |
   |---|

   daarmee worden ook discussies vermeden omtrent welk nu de beste bredere term is

3. niet elke hiërarchische relatie moet expliciet in de thesaurus opgenomen worden
   als X een bredere term is voor Y en Y een bredere term voor Z, dan hoeven er geen BT/NT referenties tussen X en Z opgenomen te worden

   | planten<br>NT fruit | en | fruit<br>NT vers fruit | maar niet | planten<br>NT vers fruit |
   |---|---|---|---|---|

**Gerelateerde termen** RT: wordt gebruikt voor een niet-hiërarchische relatie in de thesaurus zo'n RT relatie wordt best opgenomen wanneer een indexer of een zoeker die het gebruik van term X overweegt, ook best er aan herinnerd wordt dat de term Y ook bestaat

eigen inverse | pennen<br>RT calligrafie | | calligrafie<br>RT pennen |
|---|---|---|---|

Bepalen van gerelateerde relaties: in functie van een aantal categoriën:

| | |
|---|---|
| tijd | vrijetijdslectuur<br>RT vrijetijd |
| product | scheepsbouw<br>RT schip |
| toepassing | computers<br>RT tekstverwerking |
| onderdeel | voertuigen<br>RT wielen |

| | |
|---|---|
| plaats | vreemde talen<br>RT taallabos |
| oorzaak | vandalisme<br>RT vijandigheid |
| apparaat | schilderen<br>RT borstels |
| complement | ouders<br>RT kinderen |

### 3.4.4 Bereik notities

*Scope notes* zijn de meest gebruikte manier voor het aansturen om termen in een thesaurus te gebruiken:

1. geven van een definitie

   ```
   pleinvrees
   SN neiging om in problemen te geraken door een ruimtelijke positie
   ```

2. aangeven welke concepten ingesloten en welke uitgesloten zijn

   ```
   gemechaniseerd informatie opvragen
   SN omvat pre-computer methodes zoals ponskaart systemen

   beren
   SN met uitsluiting van panda's

   partij
   SN alleen voor politieke partijen, niet voor sociale samenkomsten
   ```

3. verwijzing naar andere termen
4. opnemen van bijkomende instructies voor de indexers

   ```
   hospitalisatie
   SN wijs ook termen toe voor de voorwaarden waarin patienten
      gehospitaliseerd worden

   gereedschap
   SN brede term, geef de voorkeur aan specifieke types van gereedschap
   ```

5. instructie voor synthese

```
geschiedenis
SN voegtoe als een onderdeel na termen die disciplines, activiteiten,
   levende wezens, ... aanduiden bijv. geiten-geschiedenis
```

Bereik notities moeten relevant zijn zowel voor de indexer als de vrager. De inhoud ervan moet iets toevoegen aan wat de term op zich al zegt. Een bereik notitie moet welgevormd zijn, bijvoorbeeld zonder spellingfouten en syntactisch correct geformuleerd.

### 3.4.5 Thesaurus in Oracle Text

Een thesausus kan gecreëerd en ingevuld worden met het ctx_thes package:

```
begin
    ctx_thes.create_thesaurus('default',FALSE);
    ctx_thes.create_relation('default','eten','syn','voedsel');
    ctx_thes.create_relation('default','voedsel','nt','fruit');
    ctx_thes.create_relation('default','appelen','bt','fruit');
end;
```

Verwijderen kan met hetzelfde package:

```
    exec  ctx_thes.drop_thesaurus('default');
```

In plaats van de naam `default` kan een andere naam gekozen worden, maar dan moet bij het gebruik van deze thesaurus telkens deze naam vermeld worden. De FALSE parameter geeft aan de toegevoegde elementen hoofdletterongevoelig zijn.
Een andere manier om een thesaurus te creëren is met behulp van `ctxload`:

```
ctxload -user xxx/xxx -thes -name default -file ischat.txt
```

In plaats van `-thes` kan ook `-thesdump` gebruikt worden: De ingeladen thesaurus wordt dan geëxporteerd naar het bestand gespecificeerd bij de `-file` optie.
Het import bestand beschrijft de structuur van de thesaurus en bevat een reeks entries van de volgende vorm:

```
term
SYN synoniem
NT engere term
BT bredere term
RT gerelateerde term
USE geprefereerde term
SN tekst
```

Bij elke term kunnen meerdere lijnen met SYN, NT, BT en RT voorkomen. Er kan slechts één USE aangegeven worden (in plaats hiervan kan ook `PT` of `SEE` gebruikt worden). De lengte van zo'n term (één of meerdere woorden) is maximaal 255 letters (geen speciale tekens). Samengestelde termen (met een + teken) kunnen alleen als synoniem of als geprefereerde term aangegeven worden. Bij `SN` kan de tekst maximaal 2000 tekens lang zijn; bij meerdere lijnen moet elke lijn beginnen met `SN`.
Een hiërarchie (NT/BT relaties) kan ook op een geïndenteerde manier voorgesteld worden:

```
term
   NT1 engere term
      NT2 engere term
         NT3 engere term
      NT2 engere term
```

**Gebruik van een thesaurus** met behulp van de `ctx_thes` package:

```
declare  termen varchar2(200);
begin
    termen := ctx_thes.bt('bananen',3);
    dbms_output.put_line(' bredere termen voor bananen: ' || termen );
end;
```

De bt functie heeft drie argumenten: de term die opgezocht wordt, het aantal niveaus en de naam van de thesaurus (indien deze naam *default* is, kan deze weggelaten worden).
Er zijn gelijkaardige functies voor nt, pt, rt, sn, syn en tt (top term).
Als query operator in CONTAINS:

```
select titel, score(1) from boek
where contains(tekst, 'bt(bananen)', 1) > 0
```

Als tweede argument kan het aantal niveaus dat moet bekeken worden, aangegeven worden. Als derde argument kan de naam van de te raadplegen thesaurus gegeven worden (in dat geval moet het aantal te doorzoeken niveaus gegeven worden).

## 3.5 Classificeren van documenten

Probleem: iedereen wordt overladen met informatie. Het is dus belangrijk om documenten op te delen in klassen: die met bruikbare en die met niet-bruikbare informatie.

- sleutelwoord zoekmachines, maar beperkingen, bijv. geen onderscheid op basis van context: in veel talen hebben woorden en zinsneden meerdere betekenissen, een zoekvraag kan resulteren in een aantal documenten die niet gerelateerd zijn aan het gewenste topic.
- document classificatie op basis van inhoud; maar niet realiseerbaar door mensen voor grote volumes.
- regel-gebaseerde classificatie: de ontwerper schrijft enkele regels uit waarmee de classificatie zal gebeuren;
- superviseerde classificatie: classificatie regels worden gegenereerd op basis van een verzameling vooraf geclasseerde documenten;
- niet-superviseerde classificatie: clustering op basis van de inhoud.

### 3.5.1 Regel-gebaseerde classificatie

De gebruiker verzamelt de documenten, beslist welke catergoriën er zijn en formuleert regels om deze categoriën te definiëren.
* voordeel: nauwkeurig voor klein documentverzamelingen
* nadeel: moeizaam om juiste regels te definiëren voor grote documentverzamelingen

1. creëer een tabel voor de documenten, en voeg deze documenten toe;
2. creëer een regeltabel, met categorienaam en regels;
3. creëer een CTXRULE index op de regeltabel;
4. classificeer de documenten met het MATCHES statement.

**Voorbeeld.** Creatie van de tabellen:

```
create table cursus
(    cnr integer primary key,
     titel varchar2(40),
     datum date,
     filenaam varchar2(40),
     inhoud clob                    );
```

```
create table klassen
(  vnr integer primary key,
   klasse varchar2(40),
   vraag varchar2(400)        );
```

```
create table curs_klas
(
   vnr integer,
   klas_id integer
);
```

Inladen van de documenten met SQLLDR:

```
load data
infile *
into table cursus
fields terminated by '\t'
(
        cnr position(1) integer external,
        titel char,
        datum date "DD/MM/YYYY",
        filenaam char,
        inhoud lobfile (filenaam) terminated by eof
)
begindata
1       databanken      20/09/08        /home/hcr/t/3db/sli/infdb.pdf
2       Java    30/08/08        /home/hcr/t/3oo/sli/infoo.pdf
3       webjee  20/11/08        /home/hcr/t/jee/sli/infee.pdf
```

Definiëren van de klassen:

```
insert into klassen values
    (1, 'besturingssystemen', 'unix or windows or proces or bestand');
insert into klassen values
    (2, 'databanken', 'sql or tabel or rij or zoekvraag');
insert into klassen values
    (3, 'programmeertalen', 'java or C or sql or fortran');
```

Creatie van de CTXRULE index:

```
    create index curs_kls_idx on klassen(vraag)
    indextype is ctxsys.ctxrule;
```

Voor het classificeren van de documenten wordt de CLASSIFIER.DOEN PL/SQL procedure gebruikt. Deze procedure doorloopt de `cursus` tabel, matcht elk document met een categorie en schrijft het resultaat in de `curs_klas` tabel:

```
1  create or replace package classifier as procedure doen; end;
2  /
3  show errors
4  create or replace package body classifier as
5   procedure doen is
6    v_document     clob;
7    v_item         number;
8    v_doc          number;
9   begin
10      for doc in (select cnr, inhoud from cursus)
11      loop
12         v_document := doc.inhoud;
13         v_item := 0;
14         v_doc  := doc.cnr;
15           for c in (select vnr, klasse from klassen
16              where matches(vraag, v_document) > 0 )
```

```
17              loop
18                  v_item := v_item + 1;
19                  insert into curs_klas values (doc.cnr,c.vnr);
20              end loop;
21          end loop;
22      end doen;
23  end;
24  /
25  show errors
26  exec classifier.doen
```

Merk op. Queries voor `CTXRULE` zijn gelijkaardig aan `CONTAINS` queries:
operatoren: `AND`, `NEAR`, `NOT`, `OR`, `STEM`, `WITHIN`, en `THESAURUS`, ook zinsneden (`"object gericht ontwerpen"`) zijn mogelijk.

### 3.5.2 Superviseerde classificatie

Gelijkaardig aan regel-gebaseerde classificatie, maar het definiëren van de regels is geautomatiseerd.
- voordeel: automatisch definiëren van regels, erg bruikbaar bij grote documentverzamelingen
- nadeel: op voorhand documenten aan klassen toewijzen zonder regels;
- nadeel: regels misschien niet zo specifiek en nauwkeurig als degene die een ontwerper zelf schrijft.

Regels worden automatisch gegenereerd met behulp van de CTX_CLS.TRAIN procedure waarbij gebruikt gemaakt wordt van een verzameling *leer*documenten om de classificatieregels af te leiden.

### 3.5.3 Clustering

Documenten groeperen en categorie regels schrijven is geautomatiseerd. Er gebeurt een statistische analyse van de documenten en op basis van de inhoud worden documenten gecorreleerd met clusters.
- voordeel: je hoeft geen classificatie regels of voorgeclassificeerde documentenverzameling te voorzien;
- voordeel: patronen of gelijkaardigheden die je zelf niet voorzien had, kunnen ontdekt worden.
- nadeel: onverwachte groeperingen omwille van het intern algoritme
- nadeel: regels waarmee de clusters gecreëerd worden, zijn niet gekend;
- clustering operatie is CPU-intensief.

Met behulp van de CTX_CLS.CLUSTERING procedure: creëert een hiërarchie van document-groepen (cluster) en berekent relevantie scores.

**Voorbeeld.** Creatie van een CONTENT index op de inhoud, zonder deze effectief in te laden.

```
create index curs_clus on cursus(inhoud)
       indextype is ctxsys.context parameters('nopopulate');
```

Creatie van de resultaat tabellen (eventueel automatisch gegenereerd), attribuutnamen zijn voor-gedefinieerd:

```
create table clus_res
(
     docid NUMBER,
     clusterid NUMBER,
     score NUMBER
);
```

```
create table clusters
(
     clusterid NUMBER,
     descript varchar2(4000),
     label varchar2(200),
     sze    number,
     quality_score number,
     parent number              );
```

Zetten van preferenties:

```
exec  ctx_ddl.drop_preference('c_cluster');
exec  ctx_ddl.create_preference('c_cluster','KMEAN_CLUSTERING');
exec  ctx_ddl.set_attribute('c_cluster','CLUSTER_NUM','2');
```

Uitvoeren van clustering:

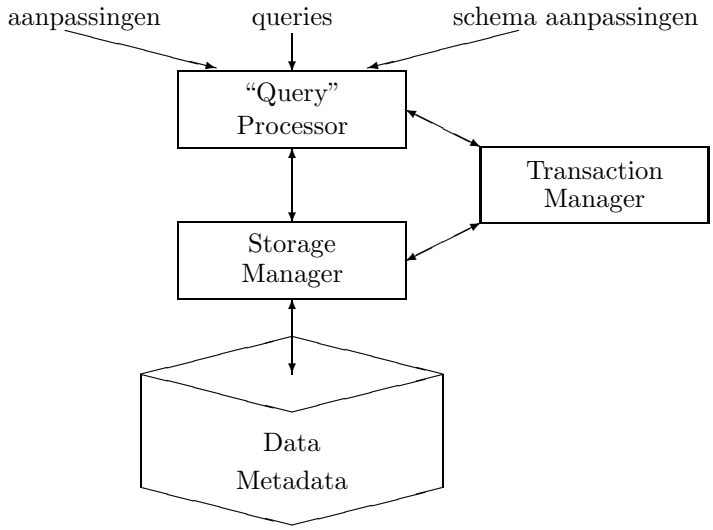
```
begin
   ctx_output.start_log('cluster_log');
   ctx_cls.clustering('curs_clus','cnr','clus_res','clusters','c_cluster');
   ctx_output.end_log;
end;
```

Resultaten:

| CLUSTERID | SZE | QUALITY_SCORE | PARENT | DOCID | CLUSTERID | SCORE |
|---|---|---|---|---|---|---|
| 1 | 0 | ,387537 | 0 | 2 | 2 | ,999999999 |
| 2 | 1 | 0 | 1 | 1 | 3 | ,999999871 |
| 3 | 2 | ,254536 | 1 | 3 | 3 | ,999999874 |

# 4 Architectuur van een DBMS

In figuur 2 worden de verschillende onderdelen van een DBMS getoond. Onderaan is de plaats voorgesteld waar de data gestockeerd wordt; gewoonlijk is dit één of meerdere disks. Deze component bevat niet alleen gewone, echte data maar ook *metadata*. Dit is informatie over de structuur van de data. Bij een R-DBMS bijvoorbeeld bevat de metadata de namen van de relaties, de namen van de attributen van deze relaties en de datatypes van deze attributen (integer, string, ...). Een DBMS bevat normaal ook *indexen* voor de data. Een index is een datastructuur die het zoeken van informatie in de databank versnelt.




Figuur 2: Belangrijkste componenten van een DBMS

**Storage manager.** Zijn taak bevat het ophalen van de gevraagde data uit de databank en het aanpassen van de informatie op aanvraag van de bovenliggende niveaus. In een eenvoudig DBMS is deze component gewoon het filesysteem van het onderliggende besturingssysteem. De naakte data wordt op disk gestockeerd waarbij het filesysteem gebruikt wordt dat normaal deel uitmaakt van het besturingssysteem. De storage manager vertaalt de verschillende DML statements in low-level filesysteem commando's en is dus verantwoordelijk voor de daadwerkelijke stockage, opvragen en aanpassen van de data in de databank. Omwille van de efficiëntie beheert een DBMS meestal zelf de data op de disk. Er zijn twee onderdelen:

**file manager** : beheert de locatie van de bestanden op de disk; levert het blok of de blokken van een bestand op aanvraag van de buffer manager;

**buffer manager** : stockeert het door de file manager geleverde blok in een pagina van het primair geheugen; dit blok blijft gedurende een bepaalde tijd in primair geheugen zodat andere queries deze data ook kunnen gebruiken zonder dat er van disk gelezen moet worden; na een tijd, wanneer er geen aanvragen voor dat blok meer blijken te zijn, wordt de pagina voor een ander net ingelezen blok gebruikt.

**Query processor.** Deze component doet meer dan queries afhandelen. Ook de vragen voor aanpassingen van de data en de metadata passeren via de query processor. Deze vragen worden meestal uitgedrukt in een taal van *hoog* niveau (bijv. SQL). De query processor vertaalt de vraag naar een reeks bevelen die naar de storage manager gestuurd worden, die ze dan zal uitvoeren.
Het moeilijkste deel is de *query optimisatie*: de keuze van een goede opeenvolging van data-aanvragen aan het storage systeem zodat snel de gevraagde data gevonden wordt. Hiervoor worden

indexen gebruikt, maar ook de volgorde waarin de verschillende stappen van een complexe query uitgevoerd worden is meestal bepalend voor de snelheid.

**Transaction manager.** Deze component is verantwoordelijk voor de integriteit van het systeem. Hij moet verzekeren dat verschillende queries die simultaan lopen niet met elkaar interfereren. Concurrentie controle: wanneer verschillende gebruikers de database gelijktijdig aanpassen, is de consistenstie van de data misschien niet meer gegarandeerd. Het is noodzakelijk voor het systeem om de interactie tussen de verschillende gelijktijdige gebruikers te controleren. Het systeem mag ook geen data verliezen, zelfs bij een systeemcrash.
Via de interactie met de query processor komt de transaction manager te weten op welke data de actuele queries operaties uitvoeren zodat conflicterende acties kunnen vermeden worden. Het is mogelijk om bepaalde queries of operaties uit te stellen zodat er geen conflicten optreden.
Er is ook interactie met de storage manager: voor de bescherming van de data moet er gewoonlijk een *log* bijgehouden worden van de veranderingen op de data. Bij een goede ordening van de operaties zal de log een lijst van een aanpassingen bevatten die na een systeemcrash terug kunnen uitgevoerd worden.

**Invoertypes.** Men kan vier types van gebruikers onderscheiden: *naïeve gebruikers* via applicatie interfaces, *applicatie programmeurs* via applicatieprogramma's, *gesophisticeerde gebruikers* via queries en *database adminstrators* die zich bezig houden met het schema van de databank.

**Queries** : vragen naar informatie. Zo'n vraag kan op twee manieren gegenereerd worden. Via een *generisch query interface* kunnen SQL statements ingetikt worden. Deze worden doorgegeven aan de query processor die een antwoord teruggeeft. Een andere manier zijn de *application program interfaces*. In een gebruiksvriendelijk programma (met GUI) kan de gebruiker aangeven welke gegevens gewenst zijn; het programma zet deze vraag zelf om in SQL statements die door de query processor uitgevoerd worden. Het resulaat wordt zo elegant mogelijk aan de gebruiker gepresenteerd.

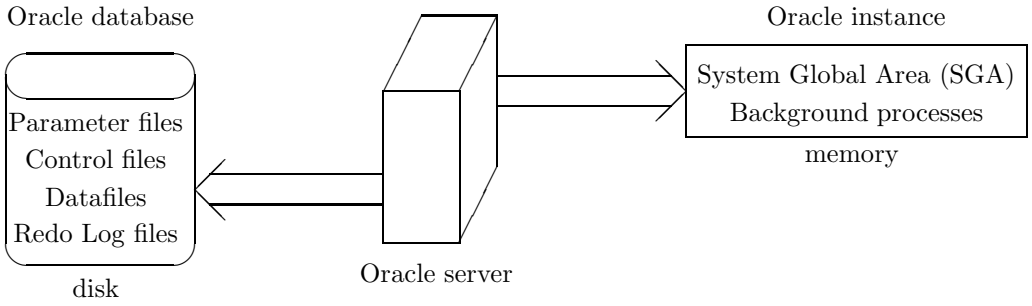
**Aanpassingen** : operaties om de gegevens te veranderen; eventueel zijn dit toevoegingen of worden er gegevens verwijderd. De manier waarop is zoals bij queries.

**Schema aanpassingen** : commando's die gewoonlijk gegeven worden door geauthoriseerd personeel, bijvoorbeeld de *database administrator*, die de toelating hebben om het schema aan te passen of een nieuwe databank te creëren.

# 5 Architectuur van Oracle.

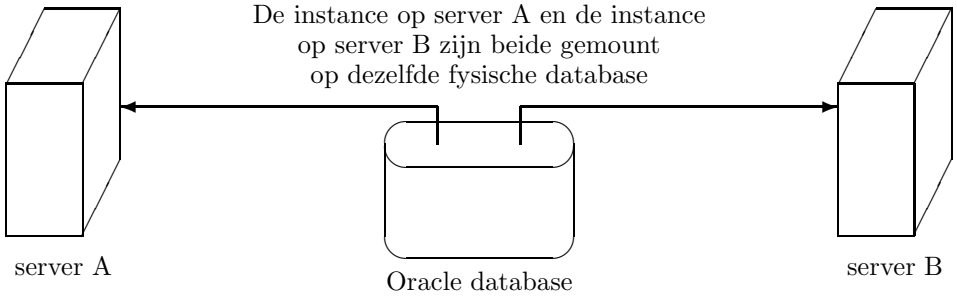
## 5.1 Database versus instance

De *database* is de data op disk, gestockeerd op bestanden van het onderliggende operating system, of eventueel in UNIX in raw bestanden. De *instance* bestaat uit het System Global Area (SGA) geheugen en de achtergrond processen. Een instance wordt geSTART door gebruik te maken van de Oracle Server Manager of de Oracle Enterprise Manager (OEM). De database wordt dan geMOUNT op de instance en tenslotte geOPENd. Gebruikers kunnen dan CONNECTeren naar de instance om de data in de database te raadplegen. Figuur 3 toont de basiscomponenten van een Oracle database en instance.




Figuur 3: Oracle database en Oracle instance

Behalve wanneer er gebruik gemaakt wordt van de Oracle Parallel Server (OPS) optie, is er een één-op-één mapping tussen instance en database. In de OPS wereld kan de database geMOUNT zijn op verschillende instances.

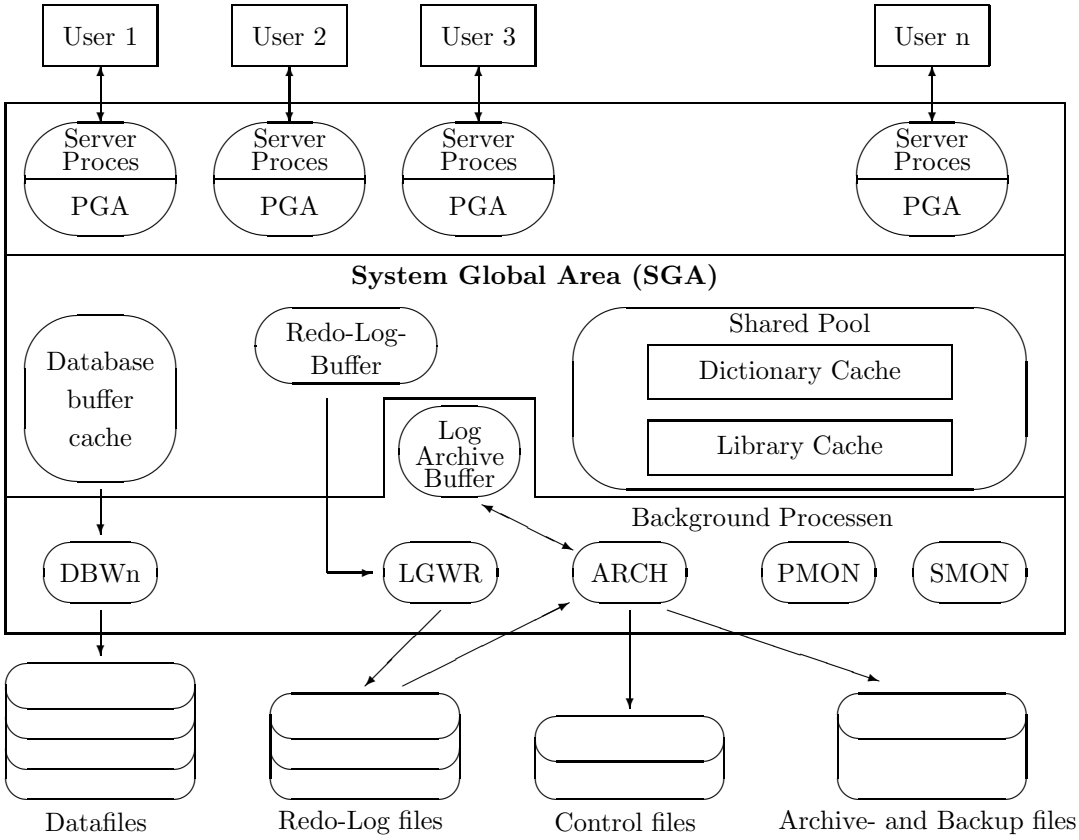



Figuur 4: Oracle Parallel Server

De Oracle DBMS server is gebaseerd op een *Multi-Server Architectuur*. De server is verantwoordelijk voor het verwerken van alle database activiteiten, zoals het uitvoeren van SQL statements, beheer van gebruikers en resources en het beheer van de opslagplaatsen (storages). Alhoewel er maar één copy van de programma code van de DBMS server in geheugen aanwezig is, wordt een logische server aan elke geconnecteerde gebruiker toegewezen. Figuur 5 illustreert de architectuur van een Oracle DBMS bestaande uit geheugenstructuren, processen en bestanden.

## 5.2 Oracle instance

De Oracle instance bestaat uit de Oracle processen en de bijhorende geheugenstructuren (zie figuur 6). Deze zijn nodig om toegang te hebben tot de bestanden die samen de Oracle database vormen en om Oracle gebruikers de data in deze bestanden te laten raadplegen.




Figuur 5: Oracle systeem architectuur

### 5.2.1 Processen

Dit zijn programma's die op een computer in uitvoering zijn, onder de controle van het operating system van de computer. Oracle creëert twee soorten processen: Oracle processen (achtergrond en server processen) en gebruikersprocessen.
De **achtergrondprocessen** voeren taken uit die anders door elk gebruikersproces dat met de database geconnecteerd is, zouden moeten uitgevoerd worden.

```
hcr 21494     1  0 08:16:32 ?          0:00 ora_qmnc_erp
hcr 21477     1  0 08:16:21 ?          0:00 ora_reco_erp
hcr 21473     1  0 08:16:21 ?          0:00 ora_ckpt_erp
hcr 21471     1  0 08:16:21 ?          0:00 ora_lgwr_erp
hcr 21483     1  0 08:16:22 ?          0:00 ora_s000_erp
hcr 21465     1  0 08:16:20 ?          0:00 ora_pmon_erp
hcr 21479     1  0 08:16:21 ?          0:00 ora_cjq0_erp
hcr 21498     1  0 08:16:34 ?          0:00 ora_mmnl_erp
hcr 21481     1  0 08:16:21 ?          0:00 ora_d000_erp
hcr 21496     1  0 08:16:34 ?          0:00 ora_mmon_erp
hcr 21475     1  0 08:16:21 ?          0:00 ora_smon_erp
hcr 21469     1  0 08:16:21 ?          0:00 ora_dbw0_erp
hcr 21502     1  0 08:16:43 ?          0:00 ora_q000_erp
hcr 21467     1  0 08:16:20 ?          0:00 ora_mman_erp
```

| Software code (Oracle executable) | | |
|---|---|---|
| Background processes | DBWn | |
| | LGWR | |
| | SMON | |
| | PMON | |
| | enz. | |
| System global area (SGA) | Database buffer cache | |
| | | Library cache |
| | | Dictionary cache (shared SQL area) |
| | Shared Pool | Control Structures |
| Server process 1 | Program global area (PGA) | Sort area |
| Server process 2 | Program global area (PGA) | Sort area |

Figuur 6: Een eenvoudige instance

**DBW*n* - database writer** (vroeger DBWR) is verantwoordelijk voor het effectieve schrijven van data vanuit de buffercache naar de fysische database bestanden van Oracle op het operating system niveau. Hierbij wordt gebruik gemaakt van het LRU algoritme; DBW*n* schrijft eerst de oudste, minst actieve blokken. Dus de frequent gevraagde blokken, zelfs de *dirty* blokken, blijven in primair geheugen. Er kunnen tot twintig DBW*n* processen gestart worden (DBW0, ..., DBW9, DBWa, ..., DBWj). Het aantal wordt bepaald door de `DB_WRITER_PROCESSES` parameter.

**LGWR - log writer** is verantwoordelijk voor het schrijven van de inhoud van de redo log buffer naar de log bestanden op disk, en voor het beheer van de log buffer. LGWR is één van de actiefste processen bij een instance met een hoge DML belasting. Een transactie wordt pas volledig afgesloten beschouwd wanneer de redo-informatie, met inbegrip van de commit record, door LGWR succesvol weggeschreven is in de redo log bestanden. Daarenboven kunnen *dirty* buffers in de buffercache niet weggeschreven naar de databestanden door DBW*n* totdat LGWR de redo-informatie weggeschreven heeft.

**SMON - system monitor** is verantwoordelijk voor:

- instance recovery bij een database instance startup (indien nodig, zoals bij na een system crash) door het toepassen van de entries in de online redo log bestanden op de databestanden;
- opruimen van onnodige temporary segmenten;
- *coalescing* van aaneensluitende vrije extents tot grotere extents.

**PMON - process monitor** doet het opkuiswerk bij een gebruikersverbinding die wegvalt, of wanneer een gebruikersproces om een andere reden faalt. De databasebuffercache en andere resources dat de gebruikersverbinding aan het gebruiken was, worden opgeruimd. Wanneer bijvoorbeeld een gebruikerssessie enkele rijen in een tabel aan het aanpassen is, waarbij een lock geplaatst is op één of meerdere rijen, en de spanning valt uit, dan zal de SQL*Plus sessie stoppen. Na enkele ogenblikken zal PMON ontdekken dat de verbinding niet langer bestaat en zal volgende taken uitvoeren:

- rollback bij niet-committed transacties die in uitvoering waren op het moment van de spanningsuitval;
- datablokken die bij de transacties betrokken zijn als beschikbaar markeren in de buffercache;
- vrijgeven van locks gezet door een gefaald of beeindigd proces;

- verwijderen van de procesid van het niet-meer-verbonden proces uit de lijst van actieve processen.

Net zoals SMON, wordt PMON periodiek wakker gemaakt, om na te gaan of deze nodig is.

**CKPT - checkpoint proces** is verantwoordelijk voor het aanpassen van de hoofdingen van het controlebestand en alle Oracle databestanden op het moment van een checkpoint, zodat ze weet hebben van de laatste succesvolle SCN *System Change Number*. Een checkpoint treedt automatisch op bij elke wissel van het redo log bestand. Op dat moment schrijft het DBWn proces alle *dirty* buffers weg, zodat het tijdstip van waarop instance recovery eventueel moet beginnen, recenter wordt. Op die manier vermindert de *Mean Time to Recovery* (MTTR).

**RECO - recoverer proces** wordt gebruikt door de Oracle gedistribueerde transaction facility om te herstellen bij het falen van gedistribueerde transacties door

- te connecteren naar andere databanken;
- het verwijderen van rijen die corresponderen met elke twijfelachtige transactie, uit elke transactie tabel van de databank.

**ARC*n* - archive proces** (optioneel). Het LGWR achtergrondproces schrijft op een cyclisch manier naar de redo-log bestanden. Wanneer het laatste redo-log bestand volledig opgevuld is, overschrijft LGWR de inhoud van het eerste redo-log bestand. Het is mogelijk om de database instance in *archive-log* mode te laten lopen. In dat geval copieert het ARC*n* proces redo-log entries naar archiefbestanden vooraleer de entries overschreven worden door het LGWR proces. Er kunnen tot tien ARC*n* processen gestart worden, bepaald door de `LOG_ARCHIVE_MAX_PROCESSES` parameter.

Een **serverproces** wordt gecreëerd om de aanvragen van een geconnecteerd gebruikersproces te behandelen.

```
hcr 21512     1  0 08:17:13 ?         0:00 oracleerp
                          (DESCRIPTION=(LOCAL=YES)(ADDRESS=(PROTOCOL=beq)))
hcr 21510 21076  0 08:17:13 pts/2     0:00 sqlplus /
```

In een **gebruikersproces** wordt één of ander applicatieprogramma (bijvoorbeeld SQL*Plus of een Pro*C programma) gestart.

### 5.2.2 Geheugen structuren

System Global Area (SGA), Program Global Area (PGA) en een sorteer gebied.
De *System Global Area* is een shared memory gebied dat door de instance gebruikt wordt om informatie te stockeren die gedeeld wordt tussen de database en de gebruikersprocessen. De belangrijkste componenten zijn:

**database buffer cache** : bevat actuele copies van datablocks van de database. Wanneer je een tabel aanpast, wordt de informatie eerst in de data buffer veranderd en pas later naar disk weggeschreven.

De beschikbare ruimte in de database buffer wordt beheerd op basis van een *least recently used* (LRU) algoritme: wanneer er vrije ruimte nodig is in de cache, worden de minst recent gebruikte blokken naar de databestanden uitgeschreven. De grootte van de database buffer cache heeft een belangrijke impact op de globale performantie van de database.

**shared pool** : bevat de SQL en PL/SQL statements die uitgevoerd worden. Elk uitgevoerd SQL statement wordt in de shared pool gestockeerd, wat Oracle toelaat om alle informatie die gegenereerd is omtrent het statement, te hergebruiken. Dit hergebruik resulteert in

beduidende verbeteringen in verwerkingssnelheid voor gebruikers in een *online transaction processing* (OLTP) omgeving die telkens dezelfde SQL statements uitvoeren.

De belangrijkste componenten van deze pool zijn de *dictionary cache* en de *library cache*. Informatie omtrent database objecten (metadata) wordt in de data dictionary tabellen gestockeerd. Wanneer de database nood heeft aan informatie, bijvoorbeeld om na te gaan of een tabelkolom die in een query gebruikt wordt, wel bestaat, worden de dictionary tabellen gelezen en de resulterende data wordt in de dictionary cache gestockeerd. Merk op dat alle SQL statements de data dictionary moeten raadplegen; door relevante delen ervan in de cache te bewaren, kan de performatie verhoogd worden.

De library cache bevat informatie omtrent recent uitgevoerde SQL statements, zoals de parse tree en het query execution plan. Indien hetzelfde statement verschillende keren uitgevoerd wordt, moet het niet telkens geparsed wordt en kan alle informatie om het statement uit te voeren uit de library cache gehaald worden.

**redo log buffer** : verzamelt de redo entries voor de online redo logs vooraleer ze naar disk worden weggeschreven. De buffer bevat informatie over wijzigingen in de datablokken in de database buffer. Terwijl de redo-log-buffer opgevuld wordt gedurende de wijzigingen aan de data, schrijft het log-writer proces informatie omtrent deze wijzigingen naar de redo-log bestanden. Deze bestanden worden na bijvoorbeeld een system crash gebruikt om de database te herstellen.

**large pool** : een optioneel gebied in de SGA en wordt gebruikt voor transacties die interageren met meer dan één databank of voor message buffers voor processen die parallelle queries uitvoeren.

**Java pool** : wordt gebruikt door de Oracle JVM (Java Virtual Machine) voor alle Java code en data in een gebruikerssessie.

**Streams pool** : nieuw in Oracle $10g$; voor het beheer van gedeelde data en gebeurtenissen in een gedistribueerde omgeving.

De *log-archive buffer* is optioneel en wordt gebruikt om tijdelijk redo-log entries te cachen die in speciale bestanden moeten gearchiveerd worden.
De *Program Global Area* is een memory gebied dat door één enkel Oracle gebruikersproces gebruikt wordt. De configuratie ervan hangt af van de connectie configuratie van de Oracle databank:

**shared server** : verschillende gebruikers delen een verbinding met de databank. Voordeel is het mindere geheugen gebruik op de server; nadeel is de mogelijke verslechterde responstijd voor gebruikers. De sessie informatie wordt in de SGA (en niet in de PGA) bijgehouden. Deze configuratie is ideaal voor een groot aantal gelijktijdige connecties met de databank waarbij infrequent kort-durende aanvragen gedaan worden.

**dedicated server** : elke gebruiker heeft zijn eigen verbinding met de databank; de sessie-memory zit in de PGA.

De PGA bevat het omgevingsgebied van de gebruiker (cursors, variabelen, ...) en procesinformatie. Daarnaast is er ook een sorteergebied dat gebruikt wordt wanneer bij de gebruikersvraag een sortering, een bitmap operatie of hash join operatie nodig is.
Met behulp van de `PGA_AGGREGATE_TARGET` en de `WORKAREA_SIZE_POLICY` parameters kan de DBA een totale grootte voor alle werkgebieden samen instellen en Oracle zal dan zelf het beheer doen en geheugen toewijzen aan de verschillende gebruikersprocessen.

## 5.3 Oracle database

### 5.3.1 Script om een database te creëren

Een voorbeeld van een parameterfile (`init.ora`):

```
# Cache and I/O
     db_block_size=8192
     db_cache_size=25165824
     db_file_multiblock_read_count=16
# Cursors and Library Cache
     open_cursors=300
# Database Identification
     db_domain=denayer.wenk.be
     db_name=erp
# Diagnostics and Statistics
     background_dump_dest=/u01/app/oracle/admin/erp/bdump
     core_dump_dest=/u01/app/oracle/admin/erp/cdump
     user_dump_dest=/u01/app/oracle/admin/erp/udump
# File Configuration
     control_files=("/u02/oradata/erp/control01.ctl",
          "/u02/oradata/erp/control02.ctl", "/u02/oradata/erp/control03.ctl")
     db_files=100
# Job Queues
     job_queue_processes=10
# Miscellaneous
     compatible=10.1.0.2.0
# NLS
     nls_language="ENGLISH"
     nls_territory="BELGIUM"
# Pools
     java_pool_size=125829120
     large_pool_size=8388608
     shared_pool_size=125829120
# Processes and Sessions
     processes=50
     sessions=60
# Security and Auditing
     os_authent_prefix=""
     remote_login_passwordfile=NONE
# Shared Server
     dispatchers="(PROTOCOL=TCP) (SERVICE=erpXDB)"
# Sort, Hash Joins, Bitmap Indexes
     pga_aggregate_target=25165824
     sort_area_size=65536
# System Managed Undo and Rollback Segments
     undo_management=AUTO
     undo_tablespace=UNDOTBS1
```

Een database bestaat normaal uit een SYSTEM tablespace die de data dictionary en andere interne tabellen, procedures, ... bevat en een tablespace (ROLLBACK) voor de rollback segmenten. Bijkomende tablespaces zijn de tablespace voor gebruikersdata (USERS), een tablespace voor tijdelijke query resultaten en tabellen (TEMP) en een tablespace (TOOLS) die gebruikt wordt door toepassingsprogramma's zoals SQL*Forms.
De grootte van de rollback tablespace en van de rollback segmenten hangt af van het type en de lengte van de transacties die normaal door de toepassingsprogramma's zullen uitgevoerd worden. Een rollback segment is een special type segment omdat het geen database objecten bevat maar een *before image* van aangepaste data die door de in uitvoering zijnde transactie nog niet gecommit is. Door gebruik te maken van de rollback segmenten kunnen aanpassingen ongedaan gemaakt worden. Oracle gebruikt rollback segmenten om "read consistency"te garanderen tussen

verschilende gebruikers. De rollback segmenten worden ook gebruikt om de "before image"van aangepaste rijen te herstellen in geval van een rollback van een lopende transactie.

1. Creatie van de databank, met specificatie van de filenaam en de grootte voor de SYSTEM tablespace. Deze SYSTEM tablespace hoeft niet al te groot te zijn, bijvoorbeeld tussen de 50 en de 80 megabytes. De lokatie kan best een andere disk zijn dan de disk voor de andere datafiles. Ook moeten de namen, locaties en groottes van de redo log files gespecificeerd worden (ten minste drie).
2. Creatie van de data dictionary views met behulp van het *catalog.sql* script.
3. Creatie van de objecten die nodig zijn voor de procedurele componenten van Oracle (bijvoorbeeld PL/SQL) met behulp van het *catproc.sql* script.
4. Creatie van een rollback segment in de SYSTEM tablespace en dit segment moet *online* gebracht worden. Dit rollback segment is nodig om andere tablespaces te kunnen creëren. Indien gewenst, kan het achteraf terug offline gezet worden en verwijderd worden.
5. Creatie van een rollback tablespace: rollback segmenten kunnen best in een specifieke tablespace gecreëerd worden. De tablespace moet voldoende groot zijn zodat de vier rollback segmenten erin kunnen opgenomen worden. Om I/O contention te vermijden kan de rollback tablespace best op een aparte disk geplaatst worden.
6. Creatie van één of meer rollback segmenten. De namen van deze segmenten moeten ook in het INIT.ORA bestand toegevoegd worden zodat ze gebruikt worden vanaf de volgende database startup.
7. Creatie van een TEMPORARY tablespace (default gebruikt Oracle de SYSTEM tablespace als tijdelijke tablespace voor elke gebruiker en dit kan best vermeden worden).
8. Creatie van bijkomende tablespaces, bijvoorbeeld één voor *tabellen* en één voor indexen. Het is aan te bevelen dat data en index tablespaces op verschillende disks gelokaliseerd worden.
9. Eventueel uitvoeren van bijkomende specifieke Oracle scripts.
10. Wijzigen van de SYSTEM account: default tablespace en TEMPORARY tablespace zijn SYSTEM, en het is beter om daar geen nieuwe objecten te creëren.
11. Wijzigen van de SYS account: default TEMPORARY tablespace is gelijk aan SYSTEM en hier kan weer beter een andere tablespace voor gebruikt worden.

In Oracle 10g worden een aantal scripts gegenereerd om de database en aanverwanten te maken:
**CreateDB.sql**

```
   | connect SYS/&&sysPassword as SYSDBA
 2 | set echo on
 3 | spool logs/CreateDB.log
 4 | startup nomount pfile="/u01/app/oracle/admin/erp/scripts/init.ora";
 5 | CREATE DATABASE "erp"
 6 | MAXINSTANCES 1
 7 | MAXLOGHISTORY 1
 8 | MAXLOGFILES 16
 9 | MAXLOGMEMBERS 3
10 | MAXDATAFILES 100
11 | DATAFILE '/u02/oradata/erp/system01.dbf' SIZE 300M REUSE AUTOEXTEND
12 |               ON NEXT  10240K MAXSIZE UNLIMITED EXTENT MANAGEMENT LOCAL
13 | SYSAUX DATAFILE '/u02/oradata/erp/sysaux01.dbf' SIZE 120M REUSE
                  AUTOEXTEND ON NEXT  10240K MAXSIZE UNLIMITED
14 | DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE '/u02/oradata/erp/temp01.dbf'
             SIZE 20M REUSE AUTOEXTEND ON NEXT  640K MAXSIZE UNLIMITED
16 | UNDO TABLESPACE "UNDOTBS1" DATAFILE '/u02/oradata/erp/undotbs01.dbf'
                  SIZE 200M REUSE AUTOEXTEND ON NEXT  5120K MAXSIZE
17 | UNLIMITED
```

```
18 | CHARACTER SET WE8ISO8859P1
19 | NATIONAL CHARACTER SET AL16UTF16
20 | LOGFILE GROUP 1 ('/u02/oradata/erp/redo01.log') SIZE 10240K,
21 | GROUP 2 ('/u02/oradata/erp/redo02.log') SIZE 10240K,
22 | GROUP 3 ('/u02/oradata/erp/redo03.log') SIZE 10240K
23 | USER SYS IDENTIFIED BY "&&sysPassword"
     USER SYSTEM IDENTIFIED BY "&&systemPassword";
24 | spool off
```

**CreateDBFiles.sql**

```
   | connect SYS/&&sysPassword as SYSDBA
 2 | set echo on
 3 | spool logs/CreateDBFiles.log
 4 | ALTER TABLESPACE SYSAUX NOLOGGING;
 5 | ALTER TABLESPACE SYSTEM NOLOGGING;
 6 | CREATE TABLESPACE "USERS" NOLOGGING DATAFILE '/u02/oradata/erp/users01.dbf'
             SIZE 5M REUSE AUTOEXTEND ON NEXT  1280K MAX SIZE UNLIMITED
             EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT  AUTO ;
 8 | ALTER DATABASE DEFAULT TABLESPACE "USERS";
 9 | spool off
```

**CreateDBCatalog.sql**

```
   | connect SYS/&&sysPassword as SYSDBA
 2 | set echo on
 3 | spool logs/CreateDBCatalog.log
 4 | @/u01/app/oracle/product/10.1.0/db_1/rdbms/admin/catalog.sql;
 5 | @/u01/app/oracle/product/10.1.0/db_1/rdbms/admin/catblock.sql;
 6 | @/u01/app/oracle/product/10.1.0/db_1/rdbms/admin/catproc.sql;
 7 | @/u01/app/oracle/product/10.1.0/db_1/rdbms/admin/catoctk.sql;
 8 | @/u01/app/oracle/product/10.1.0/db_1/rdbms/admin/owminst.plb;
 9 | connect SYSTEM/&&systemPassword
10 | @/u01/app/oracle/product/10.1.0/db_1/sqlplus/admin/pupbld.sql;
11 | connect SYSTEM/&&systemPassword
12 | set echo on
13 | spool logs/sqlPlusHelp.log
14 | @/u01/app/oracle/product/10.1.0/db_1/sqlplus/admin/help/hlpbld.sql helpus.sql;
15 | spool off
16 | spool off
```

**postDBCreation.sql**

```
   | connect SYS/&&sysPassword as SYSDBA
 2 | set echo on
 3 | spool logs/postDBCreation.log
 4 | connect SYS/&&sysPassword as SYSDBA
 5 | set echo on
 6 | create spfile='/u01/app/oracle/product/10.1.0/db_1/dbs/spfileerp.ora'
                  FROM pfile='/u01/app/oracle/admin/erp/scripts/init.ora';
 8 | shutdown immediate;
 9 | connect SYS/&&sysPassword as SYSDBA
10 | startup ;
11 | select 'utl_recomp_begin: ' || to_char(sysdate, 'HH:MI:SS') from dual;
12 | execute utl_recomp.recomp_serial();
13 | select 'utl_recomp_end: ' || to_char(sysdate, 'HH:MI:SS') from dual;
14 | spool postDBCreation.log
15 | exit;
```

### 5.3.2 De gegevensbibliotheek

De *Oracle-gegevensbibliotheek* bestaat uit *read-only* tabellen waarin informatie over de databank is opgeslagen. Deze tabellen zijn opgeslagen in de `SYSTEM` tablespace. Deze informatie wordt *metadata* genoemd: gegevens over gegevens; voorbeelden zijn

- alle objectdefinities van een schema — de definities van deze tabellen, indexen, volgnummers, views en andere databaseobjecten;
- de toegewezen (en actueel gebruikte) hoeveelheid opslagcapaciteit voor elk object;
- de namen van de Oracle gebruikersaccounts, samen met de rechten en rollen die horen bij elk account;
- de informatie die nodig is om integriteitsvoorwaarden af te dwingen.

Wanneer een SQL-opdracht zoals `CREATE TABLE`, `CREATE INDEX` of `CREATE SEQUENCE` uitgevoerd wordt om een object te creëren, wordt alle informatie over kolomnamen, kolombreedtes, standaardwaarden, voorwaarden, indexnamen, startwaarden voor volgnummers, ... opgeslagen in de vorm van metadata in de tabellen van de gegevensbibliotheek.
Systeemontwikkelaars of gebruikers zullen zelden (of nooit) deze read-only tabellen gebruiken. Alleen de verschillende processen van Oracle zullen de informatie in deze tabellen nodig hebben. Om de toegang tot informatie en het beheren van een databank te vergemakkelijken, zijn deze tabellen met behulp van verschillende *views* te lezen. Deze views zijn ook onderdeel van de gegevensbibliotheek. De vieus zijn onderverdeeld in drie categorieën.

**USER** : dit voorvoegsel wordt gebruikt bij alle views die informatie tonen over objecten die behoren bij een individuele gebruiker; men spreekt ook van het *gebruikersschema* van de databank. **USER**-views geven informatie over objecten waarvan een specifieke gebruiker zelf eigenaar is: informatie over tabellen, indexen, views, ..., welke rechten de gebruiker voor zijn objecten aan andere gebruikers gegeven heeft, ... Een voorbeeld is `user_objects`. Deze view bevat kolommen die de naam van het object, het type, de datum van creatie, ... weergeven. Met behulp van deze view kan een gebruiker een bepaald object terugvinden.

**ALL** : voor alle views die informatie over alle objecten in de databank tonen; hiermee wordt het 'blikveld' van de individuele gebruiker van de databank **ALL**-views geven informatie over objecten die niet het eigendom zijn van een gebruiker. De weergegeven objecten zijn objecten de de gebruiker kan benaderen, omdat een andere gebruiker hem daartoe voldoende rechten gegeven heeft. De view `all_objects` bevat naast kolommen van de `user_objects` ook een kolom `owner`, die de eigenaar van het desbetreffende object aangeeft.

**DBA** : voor alle views in het schema van de database beheerder van de database. **DBA**-views geven een allesomvattend beeld van de databank. Alleen databasebeheerders kunnen queries op deze views uitvoeren: men heeft hiervoor het systeemrecht `SELECT ANY TABLE` nodig. Omdat Oracle geen public synoniemen voor de DBA-views maakt, kunnen deze views alleen geraadpleegd worden door zowel de eigenaar van de view (sys) als de naam van de view te specificeren, bijvoorbeeld `sys.dba_objects`.

Een aantal gegevensbibliotheekviews zijn erg belangrijk bij het beheren van een databank.

**Dynamic performance views** zijn views waarmee je informatie en statistische informatie over de performantie van de databank kan verzamelen.

- `v$fixed_table`: toont alle `x$`-tabellen waarin de dynamische performatie-informatie wordt opgeslagen die kan worden weergegeven in `v$-views`.
- `v$session`: informatie over de huidige sessie. Deze informatie kan gebruikt worden om een sessie zo nodig te 'killen' als deze hangt of wanneer een gebruiker niet langer de databank mag benaderen.
- `v$sysstat`: informatie over de systeemperformantie; kan door de databasebeheerder gebruikt worden om de prestaties van een databank te verbeteren.

- `v$sga`: samenvattende informatie over de System Global Area.

**Views voor de toegekende rechten**

- `dba_tab_privs`: alle objectrechten van een gebruiker;
- `dba_col_privs`: alle rechten voor bepaalde kolommen van een tabel;
- `session_privs`: alle rechten van een gebruiker voor de huidige inlogsessie;
- `table_privs`: informatie over objectrechten die je hebt gegeven, hebt gekregen, waarvan je eigenaar bent of die aan PUBLIC toegekend zijn.

**Views voor rollen en rechten**

- `dba_roles`: alle rollen in de databank;
- `dba_role_privs`: de rollen die aan gebruikers en aan andere rollen toegekend zijn;
- `dba_sys_privs`: de systeemrechten die aan gebruikers en rollen toegekend zijn;
- `role_role_privs`: rollen die aan andere rollen toegekend zijn;
- `role_sys_privs`: alle systeemrechten die aan rollen toegekend zijn;
- `role_tab_privs`: tabelrechten die aan rollen toegekend zijn.

**Views voor tabellen, indexen, beperkingen, ...**

- `all_all_tables`: informatie over alle objectabellen en relationele tabellen die je als gebruiker kan benaderen;
- `all_users`: alle gebruikers die voor jou als gebruiker zichtbaar zijn, maar geeft geen beschrijving van de gebruikers;
- `dba_tablespaces`: informatie over de tablespaces in het systeem;
- `dba_data_files`: informatie over gegevensbestanden en welke tablespaces gekoppeld zijn aan welke gegevensbestanden;
- `user_constraints`: de definities van voorwaarden voor tabellen in jouw schema;
- `user_indexes`: informatie over indexen voor tabellen;
- `user_sequences`: informatie over door jouw gecreëerde volgnummers;
- `user_tables`: informatie over jouw tabellen.

### 5.3.3 Gebruikersaccounts

Toegang tot de databank wordt verleend aan een database account of *user*. Een gebruiker zonder eigenaar te zijn van één object kan bestaan. Maar wanneer een gebruiker objecten creëert in de databank, wordt hij eigenaar van deze objecten. Deze objecten zijn onderdeel van een *schema* dat dezelfde naam heeft als de database user. Een schema kan alle mogelijke objecten van een database bevatten: tabellen, indexen, sequenties, views, ...
Twee gebruikers (met elk hun eigen schema) kunnen twee verschillende objecten creëren met dezelfde naam omdat elk object opgeslagen wordt in het eigen schema. De volledige naam van elk object heeft als vorm `schema_naam.object_naam`.
De schema eigenaar of de DBA kan aan andere database users toegang verlenen tot deze objecten.
Wanneer een user gecreëerd wordt, kunnen een aantal karakteristieken aan de gebruiker toegewezen worden, bijvoorbeeld welke tablespaces beschikbaar zijn voor de gebruiker om objecten te creëren.
Creatie van een aantal tablespaces en de gebruiker die deze tablespaces gaat gebruiken:

```
SET CONCAT OFF
create tablespace &&1
datafile '/u03/oradata/brp/&&1.dbf' size 1M
nologging
```

```
minimum extent 8K
default storage ( initial 32K next 8K maxextents 50 pctincrease 0 );

create tablespace temp&&1
datafile '/u03/oradata/brp/temp&&1.dbf' size 80K
nologging
minimum extent 8K
default storage ( initial 32K next 8K maxextents 50 pctincrease 0)
temporary;

create user &&1
identified externally
default tablespace &&1
temporary tablespace temp&&1
quota unlimited on &&1
quota unlimited on temp&&1
profile default;

SET CONCAT ON
exit
```

Er zijn drie methodes om gebruikers te authenticeren in de databank:

**database authentication** : het geëncrypteerde paswoord is in de databank gestockeerd;

**operating system authentication** : een gebruiker die reeds in het operating system ingelogd is, heeft dezelfde privileges als een gebruiker met dezelfde of gelijkaardige naam (afhankelijk van de waarde van `OS_AUTHENT_PREFIX` in de databank);

**network authentication** : hierbij wordt gebruik gemaakt van *Public Key Infrastructure* (PKI) en mogelijk vanaf Oracle 10g Enterprise Edition met de Oracle Advanced Security optie.

Als een Oracle-database gecreëerd wordt, worden ook twee speciale gebruikersaccountnamen, SYS en SYSTEM, gecreëerd. SYS is eigenaar van alle basistabellen en views in de gegevensbibliotheek. Dit account moet goed beveiligd zijn en kan alleen door databasebeheerders (DBA) gebruikt worden. Nieuwe tabellen (of views) die door Oracle-software of software van derden aan de gegevensbibliotheek toegevoegd worden, zijn eigendom van de gebruiker SYSTEM, ook alleen toegankelijk door DBAs.
Nadat een gebruiker gecreëerd is, moet hij nog een aantal *rechten* krijgen. Er zijn verschillende rechten, en deze zijn onderverdeeld in twee soorten, systeemrechten en objectrechten. Om bijvoorbeeld te kunnen inloggen op SQL*Plus, moet een gebruiker het systeemrecht `CREATE SESSION` hebben.

```
grant create session to &&1;
```

**Systeemrechten** stellen een gebruiker in staat om bepaalde soorten bewerkingen, zoals het creëren, wissen of wijzigen van objecten, uit te voeren. Hieronder vallen de rechten waarmee je objecten die van jezelf zijn, kunt beheren en rechten waarmee je objecten van andere gebruiker kunt beheren.

```
CREATE SESSION          ALTER SESSION
CREATE TABLE            CREATE ANY TABLE        ALTER ANY TABLE
CREATE ANY INDEX        UNLIMITED TABLESPACE
```

Het principe is dat wanneer je een object kunt creëren, je het ook kunt wissen. Als je het recht `CREATE TABLE` hebt, kun je ook indexen voor een tabel creëren en deze vervolgens later ook wissen. Om de toegekende systeemrechten te bekijken, kan je een query uitvoeren op de view `dba_sys_privs`:

```
select * from dba_sys_privs;
```

**Objectrechten** geven een gebruiker toelating om een bepaalde bewerking op een specifiek object, zoals een tabel, view of index, uit te voeren. Mogelijke bewerkingen zijn `SELECT`, `INSERT`, `UPDATE`, `DELETE`.

Een **rol** is een verzameling rechten voor een bepaalde soort gebruiker. Telkens er een nieuwe gebruiker van die bepaalde soort gecreëerd wordt, kunnen in één keer alle nodige rechten toegekend worden door de rol toe te kennen aan de nieuwe gebruiker.
Creatie van een rol met een aantal privileges en toekennen aan een gebruiker:

```
create role student not identified;

grant create session, create table, create view,
      create procedure, create trigger to student;

grant student to &&1;
```

De naam van de rol moet binnen de database uniek zijn. Met een rol kan je zowel systeem- als objectrechten toekennen. Rollen zijn van niemand, ze worden in geen enkel gebruikersaccountschema weergegeven.

Omdat database resources niet onbeperkt zijn, moet een DBA deze resources beheren en toewijzen aan de verschillende gebruikers. Voorbeelden zijn CPU-tijd, concurrente sessies, connectietijden, ... Een database *profile* is een verzameling van resourcebeperkingen die aan een gebruiker kunnen toegewezen worden. Bij installatie van Oracle wordt een `DEFAULT` profile gemaakt en deze wordt toegewezen aan elke nieuwe gebruiker wanneer er geen expliciete toewijzing van een profile gebeurt. De initiële waarden in deze `DEFAULT` profile laten onbeperkt gebruik van alle database resources toe. De DBA kan nieuwe profiles creëren of wijzigingen aanbrengen aan de `DEFAULT` profile. Een voorbeeld van een profile (met naam conlog) waarin de connectietijd beperkt is tot 120 minuten en het aantal opeenvolgende mislukte login pogingen maximaal 8 is:

```
create profile conlog limit
     connect_time 120
     failed_login_attempts 8;
```

## 5.4 Verwerking van een SQL statement

1. Veronderstel dat een gebruiker in SQL*Plus een update statement ingeeft op de tabel `TAB` waarbij meerdere rijen aangepast zullen worden. Het statement wordt door het `USER` proces doorgegeven naar de Oracle server. De query processor van de server gaat na of het statement reeds in de library cache aanwezig is. In dat geval kan de corresponderende informatie (parse tree, uitvoeringsplan) herbruikt worden. Indien het statement niet gevonden wordt, wordt het ge-parse-d. Nadat het statement geverifieerd is (gebruikersprivilegies, betrokken tabellen en kolommen) op basis van data uit de dictionary cache, wordt een uitvoeringsplan gegenereerd door de query optimizer. Dit plan samen met de parse tree, wordt in de library cache gestockeerd.

2. Voor de objecten die betrokken zijn bij het statement (in dit geval de `TAB` tabel), wordt nagegaan of de corresponderende datablokken reeds in de database buffer aanwezig zijn. Indien niet, leest het `USER` proces de datablokken in de database buffer. Indien er niet voldoende ruimte is in de buffer, worden de minst recent gebruikte blokken van andere objecten naar disk geschreven door het `DBW`$n$ proces.

3. De aanpassingen van de rijen betrokken bij de update, worden in de database buffer gedaan. Voordat de datablokken worden aangepast, wordt de "before image" van de rijen naar de rollback segmenten geschreven door het `DBW`$n$ proces.

4. Terwijl de redo-log buffer opgevuld wordt tijdens de datablok aanpassingen, schrijft het `LGWR` proces elementen van de redo-log buffer naar de redo-log bestanden.

5. Wanneer alle rijen (of beter, alle corresponderende datablokken) aangepast zijn in de database buffer, kunnen deze aanpassingen door de gebruiker ge-commit worden met behulp van het **commit** command.

6. Zolang er geen **commit** door de gebruiker uitgevoerd is, kunnen de aanpassingen ongedaan gemaakt worden met behulp van het **rollback** statement. In dat geval worden de aangepaste datablokken in de database buffer overschreven met de originele blokken uit de rollback segmenten.

7. Indien de gebruiker een **commit** uitvoert, wordt de ruimte gealloceerd voor de blokken in de rollback segmenten vrijgegeven en kunnen deze door andere transacties gebruikt worden. De aangepaste blokken in de database buffer worden ook ge-unlock-ed, zodat andere gebruikers nu de aangepaste blokken lezen. Het einde van de transactie (d.i. de commit) wordt opgeslagen in de redo-log bestanden. De aangepaste blokken worden pas naar disk geschreven door het `DBW`*n* proces wanneer er ruimte moet vrij gemaakt worden in de database buffer voor andere blokken.

## 5.5 Enkele queries op de data dictionary views

```
-------------------------------------------------------------------------------
 510.sql | select column_name,comments from dict_columns
         | where table_name='DBA_TABLES'
-------------------------------------------------------------------------------
 607.sql | select name from v$controlfile
-------------------------------------------------------------------------------
 608.sql | select type,record_size,records_total,records_used
         | from v$controlfile_record_section
         | where type='DATAFILE'
-------------------------------------------------------------------------------
 716.sql | select * from v$logfile
-------------------------------------------------------------------------------
 832.sql | select tablespace_name,initial_extent,next_extent,min_extents,
         | max_extents,pct_increase,contents
         | from dba_tablespaces
-------------------------------------------------------------------------------
832u.sql | select tablespace_name,initial_extent,next_extent,min_extents,
         | max_extents,pct_increase,contents
         | from user_tablespaces
-------------------------------------------------------------------------------
 834.sql | select file_name, tablespace_name,bytes,
         | autoextensible,maxbytes,increment_by
         | from dba_data_files
-------------------------------------------------------------------------------
 915.sql | select *   from dba_free_space_coalesced
                      where tablespace_name='EM101'
-------------------------------------------------------------------------------
 922.sql | select segment_name,tablespace_name,extents,blocks
         | from user_segments
-------------------------------------------------------------------------------
 923.sql | select * from user_extents
-------------------------------------------------------------------------------
 924.sql | select * from user_free_space
```

```
1231.sql| declare
     2  |    v_owner VARCHAR(30) := 'nee';
     3  |    v_segment_name VARCHAR(30) := 'artikel';
     4  |    v_segment_type VARCHAR(30) := 'TABLE';
     5  |    v_total_blocks NUMBER;
     6  |    v_total_bytes NUMBER;
     7  |    v_unused_blocks NUMBER;
     8  |    v_unused_bytes NUMBER;
     9  |    v_last_used_extent_file_id NUMBER;
    10  |    v_last_used_extent_block_id NUMBER;
    11  |    v_last_used_block NUMBER;
    12  | begin
    13  |   dbms_space.unused_space( v_owner,        v_segment_name,
    14  |           v_segment_type,  v_total_blocks, v_total_bytes,
    15  |           v_unused_blocks, v_unused_bytes, v_last_used_extent_file_id,
    16  |           v_last_used_extent_block_id,     v_last_used_block ) ;
    17  |   dbms_output.put_line(v_segment_type || ' : ' || v_owner );
    18  |   dbms_output.put_line('.' || v_segment_name);
    19  |   dbms_output.put_line('total_blocks  : ' || v_total_blocks);
    20  |   dbms_output.put_line('unused_blocks : ' || v_unused_blocks);
    21  | end;
```

# 6 Het fysisch niveau: data

## 6.1 Bestandsorganisatie.

Een *bestand* is logisch georganiseerd als een sequentie van records. Deze records worden samengebracht op diskblokken. Diskblokken hebben een vaste grootte, bepaald door de fysische eigenschappen van de disk en het besturingssysteem, terwijl de lengte van een record kan variëren. In een relationele database hebben de tuples van verschillende relaties meestal een verschillende lengte.
Een eenvoudige oplossing bestaat erin de database in verschillende bestanden onder te brengen en enkel records met een bepaalde lengte in een bepaalde bestand te stockeren. Een alternatief is de bestanden zo te structureren dat er records met veranderlijke lengte in kunnen gestockeerd worden.

### 6.1.1 Records met vaste lengte.

Beschouw het voorbeeld van het bestand *klas* in school database. Elke record van het bestand is gedefinieerd als:

```
structure klas
{
       char    klas_naam[8];
       char    stud_naam[20];
       int     groep;
};
```

Indien 1 character 1 byte lang is en een integer 4 bytes, dan is deze record 32 bytes lang. In een eenvoudige benadering worden de eerste 32 bytes gebruikt voor de eerste record, de volgende 32 voor de tweede record, enz. (Figuur 7.)

| | | | |
|---|---|---|---|
| record 0 | 1ema | Huysmans | 4 |
| record 1 | 1eo | Tiri | 2 |
| record 2 | 1cc | Sanders | 1 |
| record 3 | 1bl | Jacobs | 3 |
| record 4 | 1bb | Lameire | 1 |
| record 5 | 1ema | Wuyts | 4 |
| record 6 | 1cb | Goossens | 1 |
| record 7 | 1bl | Peeters | 1 |
| record 8 | 1ema | Lemmens | 2 |

Figuur 7: Bestand met *klas* records

Er zijn echter twee problemen:

- Het is moeilijk een record te verwijderen: de ruimte die vrijkomt, moet terug opgevuld worden met een ander record van het bestand; of er moet een middel zijn om de verwijderde records aan te duiden.
- Tenzij de blokgrootte een veelvoud is van 32, zullen sommige records over blokgrenzen heen gestockeerd worden: dit vraagt dan twee blok accessen om deze record te lezen of te schrijven.

Bij het verwijderen van een record, kan men het volgende record één plaats naar voor opschuiven en ook alle volgende records. Dit kan vrij veel opschuif operaties vragen en het is misschien eenvoudiger om het laatste record op de plaats van de verwijderde record te schrijven.
Meestal worden er echter meer toevoegingen dan verwijderingen gedaan, en het is daarom beter om de verwijderde record open te laten en terug in gebruik te nemen bij een volgende toevoeging.

Een eenvoudig markeerder in de verwijderde record is echter niet voldoende omdat het dan moeilijk is deze vrije ruimte te vinden bij een volgende toevoeging. Er moet bijkomende informatie gestockeerd worden.
In het begin van het bestand worden een bepaald aantal bytes geallokeerd als *file hoofding*. Deze hoofding zal allerlei informatie omtrent het bestand bevatten. Nu is alleen de opslag nodig van het adres van de eerste record waarvan de inhoud verwijderd is. In dit eerste vrije record wordt dan het adres gestockeerd van de tweede vrije record, enz. Deze gestockeerde adressen kunnen als *pointers* geïnterpreteerd worden. Figuur 8 toont het bestand van figuur 7 waarbij records 1, 4 en 6 verwijderd zijn.

| | | | | | |
|---|---|---|---|---|---|
| hoofding | (1) | | | | (eerste vrije op 1) |
| record 0 | | 1ema | Huysmans | 4 | |
| record 1 | (4) | | | | (volgende vrije op 4) |
| record 2 | | 1cc | Sanders | 1 | |
| record 3 | | 1bl | Jacobs | 3 | |
| record 4 | (6) | | | | (volgende vrije op 6) |
| record 5 | | 1ema | Wuyts | 4 | |
| record 6 | $\perp$ | | | | (geen volgende vrije meer) |
| record 7 | | 1bl | Peeters | 1 | |
| record 8 | | 1ema | Lemmens | 2 | |

Figuur 8: Verwijdering van enkele records

Wanneer nu een record moet toegevoegd worden, wordt die record gebruikt waarnaar de hoofding wijst. In de hoofding wordt de pointer herzet zodat hij naar de tweede vrije record wijst. Indien geen ruimte vrij is, wordt de record toegevoegd aan het einde van het bestand.
Toevoegen en verwijderen van records met vaste lengte is op deze manier vrij eenvoudig te implementeren. Alleen moet opgepast worden voor het probleem van *dangling pointers*.

### 6.1.2 Records met veranderlijke lengte.

Records met veranderlijke lengte komen op verschillende plaatsen in een database voor: stockage van verschillende record types in één bestand, record types waarvan één of meer velden een veranderlijke lengte heeft, en record types met repeating velden.
Als voorbeeld wordt het *klas* bestand iets anders georganiseerd:

```
structure klas_lijst
{
        char   klas_naam[8];
        structure
        {
                char    stud_naam[20];
                int     groep;
        } stud_info[.];
};
```

*stud_info* is een array met een willekeurig aantal elementen, dus is er eigenlijk geen beperking op de lengte van de record (tenzij de capaciteit van de disk).

**Byte String voorstelling.**
Een eenvoudige manier om veranderlijke lengte records te implementeren is met behulp van een speciaal *end-of-record* ($\perp$) symbool op het einde van elke record. Elke record wordt dan als een string van opeenvolgende bytes gestockeerd. (Figuur 9.) Twee belangrijke nadelen zijn:

- Het is niet eenvoudig om ruimte die door een nu verwijderd record bezet was, te herbruiken. Er zijn technieken, maar die leiden snel tot fragmentatie.

- Normaal is er geen ruimte voorzien om records te laten groeien. De record zal dus eerst moeten verplaatst worden, wat complex kan zijn wanneer er pointers aanwezig zijn.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1ema | Huysmans | 4 | Wuyts | 4 | Lemmens | 2 | ⊥ |
| 1 | 1eo | Tiri | 2 | ⊥ | | | | |
| 2 | 1cc | Sanders | 1 | ⊥ | | | | |
| 3 | 1bl | Jacobs | 3 | Peeters | 1 | ⊥ | | |
| 4 | 1bb | Lameire | 1 | ⊥ | | | | |
| 5 | 1cb | Goossens | 1 | ⊥ | | | | |

Figuur 9: Byte String voorstelling

**Vaste-Lengte voorstelling.**
Eén of meerdere vaste-lengte records worden gebruikt om een record met veranderlijke lengte voor te stellen. Hiervoor bestaan twee technieken:

- Gereserveerde ruimte: wanneer er een maximum record lengte gekend is, kunnen vaste-lengte records met die lengte gebruikt worden. Dit leidt eventueel tot veel niet gebruikte ruimte op de disk. Alleen wanneer de meeste records een lengte hebben in de buurt van de maximum waarde, is deze techniek bruikbaar.
- Pointers: een lijst van vaste-lengte records die aan elkaar geketend zijn.

| | | | | |
|---|---|---|---|---|
| 0 | (5) | 1ema | Huysmans | 4 |
| 1 | | 1eo | Tiri | 2 |
| 2 | | 1cc | Sanders | 1 |
| 3 | (7) | 1bl | Jacobs | 3 |
| 4 | | 1bb | Lameire | 1 |
| 5 | (8) | | Wuyts | 4 |
| 6 | | 1cb | Goossens | 1 |
| 7 | | | Peeters | 1 |
| 8 | | | Lemmens | 2 |

Figuur 10: *klas* bestand met pointers

Om een bestand met de pointer methode te structureren, wordt een pointer veld toegevoegd. Met behulp van dit veld worden de records behorende bij eenzelfde klas aan elkaar geketend. (Figuur 10.) Een nadeel is de verkwisting van ruimte in alle records behalve de eerste in de keten. Dit eerste record bevat de waarde voor *klas_naam*, terwijl in de volgende records deze waarde niet gestockeerd is. Toch moet dit veld in elke record voorzien worden, omdat met vaste-lengte records gewerkt wordt. De verkwiste ruimte kan vrij groot zijn, omdat normaal in één klas verscheidene studenten zitten.
Om dit probleem op te lossen worden in het bestand met twee types van blokken gewerkt. Het **anchor blok** bevat het eerste record van de keten en in **overflow** blokken worden de overige records gestockeerd. Hierdoor hebben alle records *in een blok* dezelfde lengte, terwijl niet alle records in het bestand dezelfde lengte hebben. (Figuur 11.)

### 6.1.3 Organisatie van records in blokken.

Een bestand bestaat uit een verzameling records. Data tussen disk en primair geheugen wordt echter getransfereerd in *blok eenheden*. Het is daarom misschien nuttig om records zodanig aan blokken toe te wijzen zodat blokken gerelateerde records bevatten.
Wanneer records willekeurig aan blokken toegewezen worden, zal het gewoonlijk zo zijn dat een verschillend blok moet gelezen worden om toegang te krijgen tot elk record. Wanneer echter

| | | | | |
|---|---|---|---|---|
| 0 | (0) | 1ema | Huysmans | 4 |
| 1 | | 1eo | Tiri | 2 |
| 2 | | 1cc | Sanders | 1 |
| 3 | (1) | 1bl | Jacobs | 3 |
| 4 | | 1bb | Lameire | 1 |
| 5 | | 1cb | Goossens | 1 |

| | | | |
|---|---|---|---|
| 0 | (2) | Wuyts | 4 |
| 1 | | Peeters | 1 |
| 2 | | Lemmens | 2 |

Figuur 11: Anchor blok en overflow blok organisatie

toegang tot verscheidene van de gewenste records kan verkregen worden door middel van één blok access, worden disk accessen uitgespaard. Omdat disk accessen meestal de bottleneck zijn in de performantie van een database systeem, kan zorgvuldige toewijzing van records aan blokken de efficiëntie verhogen.
In plaats van met *anchor* en *overflow* blokken te werken, kan ook volgende structuur gebruikt worden. Deze heeft wel wat meer ruimte nodig, maar zorgt voor een snellere toegang tot de informatie. Aan elke *klas_naam* waarde wordt een *bucket* toegewezen. Deze bucket bevat de volledige veranderlijke-lengte record voor de corresponderende *klas_naam* waarde. Een bucket bestaat uit zoveel blokken als nodig is om de informatie voor te te stellen, maar een bepaald blok wordt niet gelijktijdig door twee buckets gebruikt. De structuur van een bucket is als volgt:

- een eerste record waarin de *klas_naam* gestockeerd wordt;
- volgende records voor de repeating velden; in deze records wordt de *klas_naam* niet herhaald, omdat deze toch dezelfde is voor alle records in de bucket; er is ook geen reden om de records aan elkaar te ketenen omdat ze dezelfde veranderlijke-lengte record voorstellen (ketens kunnen nog wel gebruikt worden voor het hergebruik van plaats van verwijderde records).

Merk op dat er twee verschillende record lengtes aanwezig zijn in een bucket. De eerste record bevat één veld, *klas_naam*. De volgende records bevatten twee velden, *stud_naam* en *groep*. Het toevoegen en verwijderen van records in een bucket kan gebeuren met technieken beschreven bij vaste-lengte records.
Wanneer er veel informatie in de bucket moet gestockeerd worden, is één blok waarschijnlijk niet voldoende. De verschillende blokken van de bucket kunnen dan elkaar geketend worden met dezelfde technieken als voor het ketenen van records. Een vaste hoeveelheid ruimte wordt in het begin van elk blok gealloceerd als *blok hoofding*. Hierin worden de bucket-keten pointers gestockeerd.
Wanneer de bucket groeit, moeten nieuwe blokken toegevoegd worden; bij verwijdering van records, kunnen bepaalde blokken leeg geraken. Zoals bij verwijderde records gebeurde, kan ook hier een keten van beschikbare blokken bijgehouden worden om ze te herbruiken voor buckets die nieuwe blokken nodig hebben.

blok 0

| | 1ema | |
|---|---|---|
| | Huysmans | 4 |
| | Wuyts | 4 |

blok 1

| | 1eo | |
|---|---|---|
| | Tiri | 2 |
| | | |

blok 2

| | 1ema | |
|---|---|---|
| | Lemmens | 2 |
| | | |

blok 3

| | 1bl | |
|---|---|---|
| | Jacobs | 3 |
| | Peeters | 1 |

blok 4

| | 1bb | |
|---|---|---|
| | Lameire | 1 |
| | | |

blok 5

| | 1cb | |
|---|---|---|
| | Goossens | 1 |
| | Sanders | 1 |

Figuur 12: Bucket bestandsorganisatie

Dit hergebruik is een goed idee vanuit het standpunt van ruimte-efficiëntie. Het is echter niet altijd even goed vanuit het standpunt van tijd-efficiëntie. Wanneer een bucket doorzocht wordt, moet ieder bijhorend blok gelezen worden. Om de tijd voor deze zoekoperatie te minimaliseren, moet de transfertijd van disk naar primair geheugen geminimaliseerd worden. Dit kan gerealiseerd worden door blokken behorende bij een bucket op dezelfde cylinder of naburige cylinders te plaatsen. Dus wanneer alle records uit een blok verwijderd zijn, is het te verkiezen dat dit blok voor dezelfde bucket hergebruikt wordt, in plaats van voor een andere bucket. Deze strategie kan in een groot aantal lege blokken resulteren, maar dit komt alleen voor wanneer record-verwijdering frekwenter is dan toevoeging.
In praktijk is het onmogelijk om een perfekte toewijzing van de blokken op de disk te bewaren zonder uitermate veel ruimte leeg te laten. Op bepaalde momenten heeft een bucket meer dan één cylinder nodig en het kan zijn dat geen van de naburige cylinders vrij is. In zo'n geval zal om het even welke vrije ruimte gebruikt worden, zodat de blokken van een bucket toch verspreid geraken over de disk. Wanneer de verspreiding zodanig uitgebreid is dat de performantie er onder begint te lijden, kan de database gereorganiseerd worden. De database wordt op tape gecopiëerd en dan herladen waarbij de blokken zodanig geplaatst worden op disk dat de buckets niet meer gefragmenteerd zijn en er voldoende ruimte voorzien is voor bucket groei. Tijdens zo'n reorganisatie is het gewoonlijk nodig om de toegang tot de database te ontzeggen aan de gebruikers.

## 6.2 Oracle database

### 6.2.1 Fysische database structuur

De database bestaat uit vier types bestanden.

**Initialisatie parameter files** : (gewoonlijk INIT.ORA genoemd) specificatie van de configuratie van de instance; wordt gelezen bij het STARTen van een instance. In dit bestand worden de locaties gespecificeerd van *control files*, *redo log files*, en nog enkele andere bestanden. Ook bevat dit bestand de grenzen van de groottes van de verschillende structuren in de SGA en ook het aantal gebruikers dat tegelijk kan connecteren naar de databank.

**Control files** : bevatten de namen van alle datafiles en de online en gearchiveerde log files, dus metadata of data in verband met de fysische structuur van de databank; worden gelezen bij het MOUNTen.

**Datafiles** : bevatten de actuele informatie van de database; na het OPENen van de database, worden deze bestanden aangesproken. Elke Oracle datafile correspondeert met één fysisch OS bestand op disk. Elke datafile is een deel van één en slechts één tablespace; een tablespace kan wel bestaan uit meerdere datafiles. Wanneer een datafile gecreëerd wordt met de `AUTOEXTEND` parameter, zal de datafile automatisch groeien wanneer meer ruimte nodig is (met als ultieme limiet de grootte van de disk). Met behulp van de `MAXSIZE` parameter kan de grootte van de expansie beperkt worden.

**Redo log files** : bevatten records voor elke verandering die op de database doorgevoerd wordt; telkens een update in de database gebeurt, wordt de verandering bewaard in de online log files; op deze manier wordt een recovery mechanisme voorzien in geval van faling. Elke database moet minstens twee redo log files hebben, en deze worden circulair gebruikt. In het ideale geval wordt de informatie in een redo log file nooit gebruikt. Alleen bij een faling van de server (bijvoorbeeld bij een spanningsuitval) kan het zijn dat de nieuwe en aangepaste datablokken in de database buffer cache nog niet weggeschreven zijn naar de datafiles. Bij het herstarten van de Oracle instance zullen de entries in de redo log files gebruikt worden in een *roll forward* operatie om de toestand van de databank te herstellen tot op het punt van de faling.

Een **block** bepaalt de fijnste eenheid van ruimte waarin data kan gestockeerd worden. Eén datablock komt overeen met een specifiek aantal bytes van de fysische database ruimte op disk.

De grootte van een datablock wordt bij de creatie van de database gespecificeerd. Een database alloceert en verbruikt ruimte in termen van Oracle-data-blocks. De data dictionary views USER_SEGMENTS en USER_EXTENTS geven aan hoeveel blokken er gealloceerd zijn voor een database object en hoeveel blokken er vrij zijn in een segment of extent.

### 6.2.2 Logische database structuren

Naast de fysische bestanden die samen de database uitmaken, maakt Oracle ook gebruik van een aantal **logische database structuren**.

**Database** : bestaat uit één of meerdere tablespaces.

**Tablespace** : de basis storage allocatie in Oracle. Elke tablespace is samengesteld uit één of meerdere fysische (operating system) bestanden. Elke database wordt gecreëerd met de SYSTEM tablespace. Andere tablespaces worden door de DBA gecreëerd.

**Schema** : essentieel hetzelfde als een gebruikersnaam. Elk object in de database is eigendom van een schema. Elke database wordt gecreëerd met twee initiële schema's: SYS, gebruikt om de data dictionary te stockeren en SYSTEM dat dikwijls een aantal data dictionary uitbreidingen en kritisch tabellen voor andere tools stockeert. Andere schema's worden door de DBA gecreëerd. Elk schema kan quota's toegewezen worden in om het even welke tablespace. Er is geen noodzakelijk verband tussen een schema en een tablespace.

**Segment** : Elk object dat ruimte nodig heeft, wordt gecreëerd als één of meerdere segmenten. Elk segment kan slechts in één tablespace zitten.

**Extent** : een aaneensluitende allocatie van ruimte in een datafile van een tablespace. Elk segment is samengesteld uit één of meerdere extents. Bij de creatie van een segment wordt de grootte van het initiële extent en de volgende extents gespecificeerd, alsook het minimum en maximum aantal extents.

**Rollback segment** : Telkens een update in een tabel gebeurt, wordt de oude waarde weggeschreven in een rollback segment, wat de andere gebruikers toelaat om een consistente read op de tabel te blijven doen. Het geeft Oracle ook de mogelijkheid om de inhoud van de tabel te herstellen wanneer de verandering niet gecommit wordt.

**Temporary segment** : wordt door Oracle gebruikt bij tabel en index creatie en bij sortering; en ook wanneer tijdelijke ruimte nodig is bij andere operaties, zoals bijvoorbeeld hash joins.

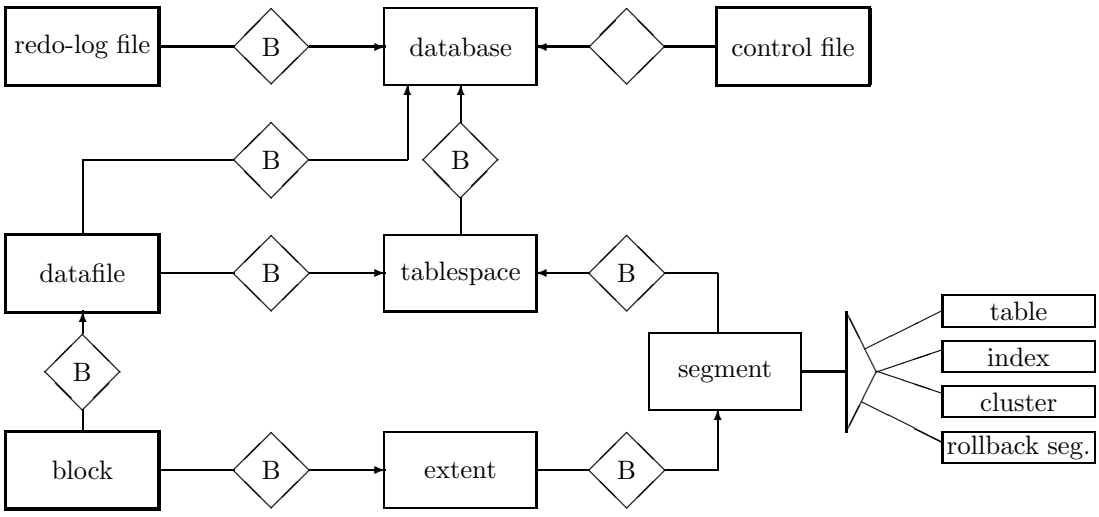
**Table** : Alle data in een database is gestockeerd in een tabel. Onder data wordt niet alleen de gebruikersdata verstaan, maar ook de inhoud van de data dictionary.

**Index** : Een index wordt gebruikt om het snel opvragen van gegevens uit een tabel te vergemakkelijken en om de uniqueness van kolomwaarden op te leggen. Indexen worden normaal in aparte segmenten naast de table data gestockeerd.

Figuur 13 geeft het verband tussen de logische en de fysische database structuren.

### 6.2.3 Creatie van een databaseobject

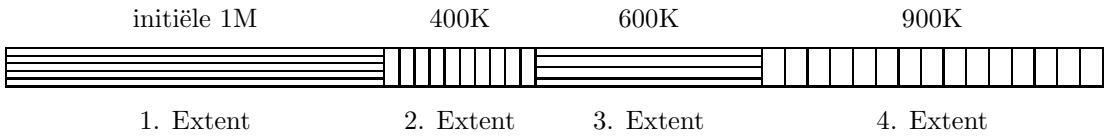
Voor elk databaseobject dat eigen geheugenruimte nodig heeft, zoals een tabel of een index, wordt een segment in een tablespace gealloceerd. Omdat het systeem gewoonlijk niet weet wat de grootte van het object zal worden, worden enkele default storage parameters gebruikt. De gebruiker heeft echter de mogelijk om deze storage parameters expliciet te specificeren door de `storage` clause in bijvoorbeeld het `create table` statement te gebruiken. Deze specificatie vervangt dan de systeem parameters en laat de gebruiker toe om de (verwachte) grootte van het object in termen van *extents* aan te geven.
Een voorbeeld:




Figuur 13: ER-diagram

```
create table STOCKS ( item   char(30),
                      aantal integer )
       storage ( initial 1M next 400k
                 minextents 1 maxextents 20 pctincrease 50 );
```

`initial` en `extents` specificeren respectievelijk de lengte van de eerste en de volgende extents: de initiële extent is 1MB lang en de volgende extent heeft een lengte van 400KB. `minextents` geeft het totaal aantal extents dat gealloceerd wordt wanneer een object gecreëerd wordt (default en minimum waarde is 1). `maxextents` geeft het maximum toegelaten extents aan. De parameter `pctincrease` specificeert het percentage waarmee elke extent, na de tweede extent, groeit ten opzichte van de vorige extent. De default waarde is 50: elke volgende extent is 50% groter dan de vorige extent.
Wanneer 4 extents in gebruik genomen zijn, is de totale lengte van de tabel 2900KB (zie figuur 14).




Figuur 14: Logische storage structuur van de tabel STOCKS

Wanneer de ruimte nodig voor het database object op voorhand gekend is, moet de initiële extent voldoende groot zijn om het database object te kunnen stockeren. De Oracle server zal dan aaneensluitende datablokken op de disk proberen te alloceren voor dit database object, zodat fragmentatie kan voorkomen worden.

**ROWID.** Dit data type is een unieke identificator van een rij en wordt gebruikt om een rij te localiseren. ROWID is een pseudo-kolom dat naast alle andere kolommen van een tabel kan opgevraagd worden. Het heeft de volgende karakteristieken:

- unieke identificator voor elke rij in de database;
- wordt niet expliciet als een kolom waarde gestockeerd;

- geeft niet direct het fysisch adres van een rij; maar kan toch gebruikt worden om een rij te lokaliseren;
- de snelste manier om een rij in een tabel aan te spreken;
- ROWIDs worden in indexen gestockeerd om rijen met een gegeven verzameling van sleutelwaarden te specificeren.

Het formaat:

| OOOOOO | FFF | BBBBBB | RRR |
|---|---|---|---|
| Data object number | Relative file number | Block number | Row number |

- Het *data object number* wordt toegekend aan elk data object en is uniek binnen de database.
- Het *relative file number* is uniek voor elk bestand in een tablespace.
- Het *block number* specificeert de positie van het blok dat de rij bevat, in het bestand.
- Het *row number* identificeert de positie van het rij-directory-slot in de blokhoofding.

Intern heeft het data object number 32 bits nodig, het relative file number 10 bits, het block number 22 bits en het row number 16 bits, dus in het totaal 80 bits, of 10 bytes.
ROWID wordt getoond met behulp van een basis-64 encoding schema: 6 posities + 3 posities + 6 posities + 3 posities. Dit basis-64 encoding schema gebruikt de tekens "A-Z", "a-z", "0-9", "+" en "/" (in het totaal 64 tekens).

```
select ROWID, knr, knaam from klant;

ROWID              KNR KNAAM
------------------ --- ----------
AAADLiAADAAAABHAAA k01 bart
AAADLiAADAAAABHAAB k02 sofie
AAADLiAADAAAABHAAC k03 jantje
AAADLiAADAAAABHAAD k04 bruno
AAADLiAADAAAABHAAE k05 sara
AAADLiAADAAAABHAAF k06 timmeke
```

Blocknumber `AAAABH` volgens base-64:

| | | | |
|---|---|---|---|
| B | (1) | 000001 | 64 |
| H | (7) | 000111 | 7 |
| | | | 71 |

# 7 Het fysisch niveau: indexen

## 7.1 Index-Sequentiële bestanden.

Een index-sequentieel bestand bestaat uit een sequentieel bestand en een index. Het wordt gebruikt in applikaties die zowel de gehele file sequentieel moeten verwerken als random access moeten hebben tot individuele records.

**Sequentieel bestand.**

Een sequentieel bestand is ontworpen voor het efficiënt verwerken van records in gesorteerde volgorde gebaseerd op een bepaalde zoeksleutel.

| | | | | |
|---|---|---|---|---|
| 0 | (1) | Brik | | 45 |
| 1 | (2) | Donk | | 25 |
| 2 | (3) | Donk | | 82 |
| 3 | (4) | Mei | | 64 |
| 4 | (5) | Peo | | 30 |
| 5 | (6) | Peo | | 90 |
| 6 | (7) | Peo | | 70 |
| 7 | (8) | Ree | | 17 |
| 8 | | Rots | | 58 |

| | | | | |
|---|---|---|---|---|
| 0 | (1) | Brik | | 45 |
| 1 | (2) | Donk | | 25 |
| 2 | (3) | Donk | | 82 |
| 3 | (71) | Mei | | 64 |
| 4 | (5) | Peo | | 30 |
| 5 | (6) | Peo | | 90 |
| 6 | (7) | Peo | | 70 |
| 7 | (8) | Ree | | 17 |
| 8 | | Rots | | 58 |
| 71 | (4) | Nu | | 36 |

Figuur 15: Voor en na een toevoeging van **Nu**

Om een snel opvragen van de records in zoeksleutel volgorde te realiseren, worden de records aan elkaar geketend m.b.v. pointers. De pointer in elke record verwijst naar de volgende record in zoeksleutel volgorde. En daarbij worden de records fysisch ook zoveel mogelijk in zoeksleutel volgorde gestockeerd om de blok accessen gedurende sequentiële verwerking zoveel mogelijk te minimaliseren.
Deze fysische sequentiële orde is moeilijk te behouden wanneer records toegevoegd en verwijderd worden omdat het niet opweegt om veel records te verschuiven wanneer één record toegevoegd of verwijderd wordt. Record-verwijdering kan beheerd worden m.b.v. pointer kettingen zoals in vorige sectie beschreven. Bij record-toevoeging worden de volgende regels toegepast:

1. Lokaliseer de record in het bestand die net voor de nieuw toe te voegen record komt in zoeksleutel volgorde.
2. Indien er een vrije record is (tengevolge van een vorige verwijdering) in hetzelfde blok als dit record, voeg het nieuw record daar toe. In het andere geval wordt het record toegevoegd in een *overflow blok*.
3. Pas de pointers aan zodat de ketting in zoeksleutel volgorde staat.

Een voorbeeld van toevoeging kan teruggevonden worden in de figuur 15. Deze manier resulteert in een snelle toevoeging van records, maar sequentieel verwerkende applikaties missen dan de juiste fysische ordening. Wanneer slechts weinig records in overflow blokken moeten geplaatst worden, is het probleem beperkt. Op een bepaald ogenblik kan echter de correspondentie tussen zoeksleutel volgorde en fysische volgorde totaal verloren zijn, zodat de sequentiële verwerking minder efficiënt gebeurt.
Op dat moment moet het bestand *gereorganiseerd* worden, zodat de sequentiële volgorde ook fysisch hersteld wordt. Zo'n reorganisatie kost tijd en geld en wordt daarom uitgevoerd wanneer de systeembelasting klein is. De frekwentie waarmee reorganisaties moeten gebeuren, hangt af van de frekwentie van het toevoegen van nieuwe records.

**Index bestand.**

Om ook een snelle random toegang toe te laten, wordt een index structuur gebruikt. Twee types van index kunnen gebruikt worden:

- **Dense index**. Er is een index record voor elke zoeksleutel waarde in het bestand. Deze record bevat de zoeksleutel waarde en een wijzer naar het data record.
- **Sparse index**. Index records worden slechts gecreëerd voor sommige data records. Om een data record te lokaliseren, wordt het index record gezocht met de grootste zoeksleutel waarde die kleiner of gelijk is aan de zoeksleutel waarde van de te zoeken record. Vanaf de record waarnaar de index record wijst, wordt de zoekoperatie gestart, waarbij de pointers in het bestand gevolgd worden totdat het gewenste record gevonden wordt.

| | | | | |
|---|---|---|---|---|
| Brik | (0) | 0 | (1) | Brik | | 45 |
| Donk | (1) | 1 | (2) | Donk | | 25 |
| Mei | (3) | 2 | (3) | Donk | | 82 |
| Peo | (4) | 3 | (4) | Mei | | 64 |
| Ree | (7) | 4 | (5) | Peo | | 30 |
| Rots | (8) | 5 | (6) | Peo | | 90 |
| | | 6 | (7) | Peo | | 70 |
| | | 7 | (8) | Ree | | 17 |
| | | 8 | | Rots | | 58 |

Figuur 16: Dense index

Met behulp van een *dense* index kan men sneller een record lokaliseren. Het voordeel van *sparse* indexen is dat ze minder ruimte nodig hebben en ook minder onderhoud vragen bij het toevoegen en verwijderen van records.

| | | | | | | |
|---|---|---|---|---|---|---|
| Brik | (0) | 0 | (1) | Brik | | 45 |
| Mei | (3) | 1 | (2) | Donk | | 25 |
| Ree | (7) | 2 | (3) | Donk | | 82 |
| | | 3 | (4) | Mei | | 64 |
| | | 4 | (5) | Peo | | 30 |
| | | 5 | (6) | Peo | | 90 |
| | | 6 | (7) | Peo | | 70 |
| | | 7 | (8) | Ree | | 17 |
| | | 8 | | Rots | | 58 |

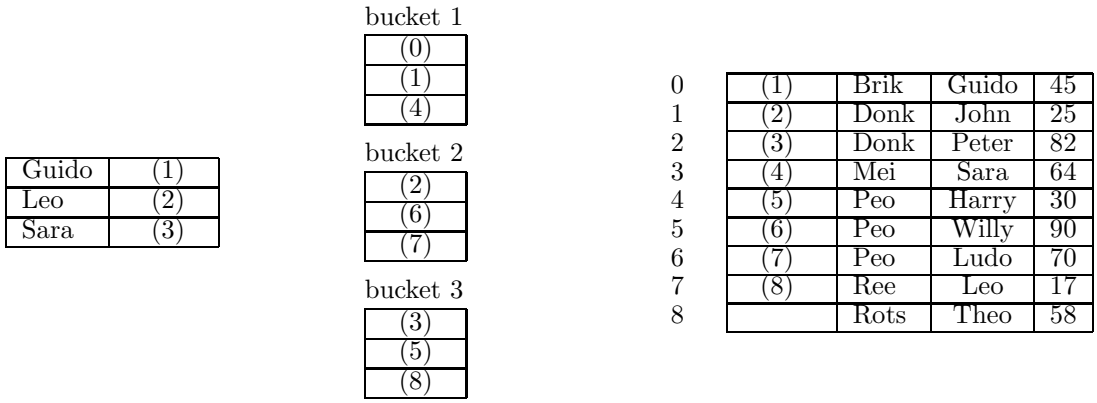
Figuur 17: Sparse index

Zelfs met een *sparse* index kan deze te groot worden voor efficiënte verwerking. Een bestand met bijvoorbeeld 100000 records (10 per blok) is in praktijk niet onredelijk. Indien er één index record per blok is, heeft men dus 10000 records in de index. Met bijvoorbeeld 100 index records per blok vraagt dit index bestand 100 blokken.
Wanneer de index voldoende klein is, kan deze in primair geheugen bewaard worden, zodat de zoektijd klein is. Bij een te grote index moet deze op disk bewaard worden en resulteert een zoekoperatie in verscheidene disk blok leesoperatie. Wanneer de index $b$ blokken bevat, moeten met een *binary search* tot $1+\log_2(b)$ blokken gelezen worden. Voor een 100 blok-index betekent dit 7 blokken. Wanneer *overflow* blokken gebruikt zijn, kan een *binary search* niet gebruikt worden. In dat geval moet sequentieel gezocht worden waarbij tot $b$ blokken gelezen moeten worden.
Om hieraan een oplossing te geven, wordt de index file beschouwd als een gewone sequentiële file en wordt hierop een extra *sparse* index geconstrueerd. Om een record te lokaliseren wordt eerst een binary search toegepast op de buitenste index waar een record gevonden wordt met de grootst mogelijke zoeksleutel waarde net kleiner dan of gelijk aan gewenste zoeksleutel. De pointer hierin wijst naar de binnenste index. Dit blok wordt afgezocht naar een record met de grootst mogelijke

zoeksleutel waarde net kleiner dan of gelijk aan de gewenste zoeksleutel. De pointer in dit record wijst dan naar het blok van het bestand waarin de gezochte record zich bevindt.
Door twee levels van indexen te gebruiken, moet slechts één index blok gelezen worden in plaats van 7, indien verondersteld wordt dat de buitenste index klein is en in primair geheugen aanwezig is. Indien het bestand erg omvangrijk is en de buitenste index te groot wordt om nog in primair geheugen te stockeren, kan een derde level toegevoegd worden. Dit is echter zeer uitzonderlijk.
Onafhankelijk van de vorm van de index, moet elke index aangepast worden telkens een record toegevoegd aan of verwijderd van wordt het bestand. Voor een enkelvoudige index kunnen volgende algoritmes gebruikt worden.

- **Verwijdering** De record die moet verwijderd worden, wordt eerst opgezocht. Indien het de laatste is met deze specifieke zoeksleutel waarde, dan wordt de zoeksleutel waarde uit de index verwijderd. Voor een *dense* index is dit gelijkaardig aan de verwijdering van een record in een bestand. Voor een *sparse* index wordt de sleutelwaarde vervangen door de entry in de index (indien er één was) met de volgende zoeksleutel waarde (in zoek sleutel volgorde). Indien de volgende zoeksleutel waarde reeds een index entry heeft, wordt de entry verwijderd.
- **Toevoeging** Er wordt een zoekoperatie uitgevoerd met de zoeksleutel waarde van de toe te voegen record. Bij een *dense* index en de waarde is nog niet aanwezig, dan wordt ze toegevoegd. Indien de index *sparse* is, moet er geen verandering gebeuren tenzij een nieuw blok gecreëerd wordt. In dat geval wordt de eerste zoeksleutel waarde (in zoeksleutel volgorde) van het nieuwe blok, in de index toegevoegd.

**Secundaire indexen.** In een standaard index-sequentieel bestand wordt slechts één index onderhouden. Wanneer er gekozen wordt om verschillende indexen te hebben op verschillende zoeksleutels, wordt de index welke de sequentiële orde van het bestand specificeert, *primaire index* genoemd. De andere indexen worden *secundaire indexen* genoemd.



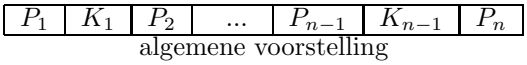

Figuur 18: Secundaire index

Secundaire indexen worden meestal anders gestructureerd dan de primaire index door het gebruik van een extra level van indirectie. De pointers wijzen niet direct naar het bestand, maar naar een bucket (een keten van één of meerdere blokken) die pointers naar het bestand bevat. Deze benadering laat toe dat alle pointers voor een bepaalde secundaire zoeksleutel waarde tesamen gestockeerd worden. Dit is interessant voor een sequentiële verwerking in secundaire sleutel volgorde.
Het is wenselijk om een *dense* secundaire index te gebruiken. Hierdoor wordt de noodzaak geëlimineerd om records met secundaire zoeksleutel waarde te lezen behalve dan diegene waarop de opzoeking gebeurt.

Bij elke aanpassing aan het bestand, moeten alle indexen aangepast worden. Dit kan een enorme overhead geven wanneer er veel secundaire indexen zijn.

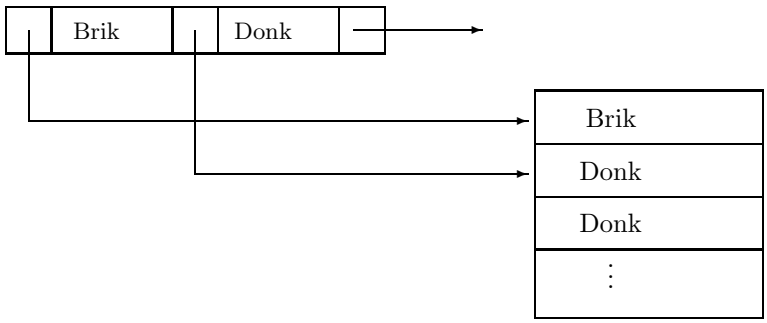
## 7.2 B$^+$-Boom bestanden.

De **B$^+$-boom** file structuur wordt zeer veel gebruikt omdat hier de efficiëntie bewaard blijft bij het toevoegen en verwijderen van data. Een B$^+$-boom index heeft de vorm van een *gebalanceerde* boom waarbij elk pad van de *root* naar een *leaf* van de boom even lang is. Elke *node* in de boom bevat tussen $\lceil n/2 \rceil$ [†] en $n$ kinderen waarbij $n$ vastligt voor een bepaalde boom.
Een B$^+$-boom structuur resulteert in wat overhead bij het toevoegen en verwijderen en ook is er wat ruimte overhead. Deze overhead is echter aanvaardbaar voor bestanden met een hoge frekwentie van modificaties omdat de kost van file reorganisaties vermeden wordt.
Een B$^+$-boom index is een multi-level index bestaande uit een root, nodes en leaves. Een typisch leaf wordt voorgesteld in de figuur 19.



algemene voorstelling

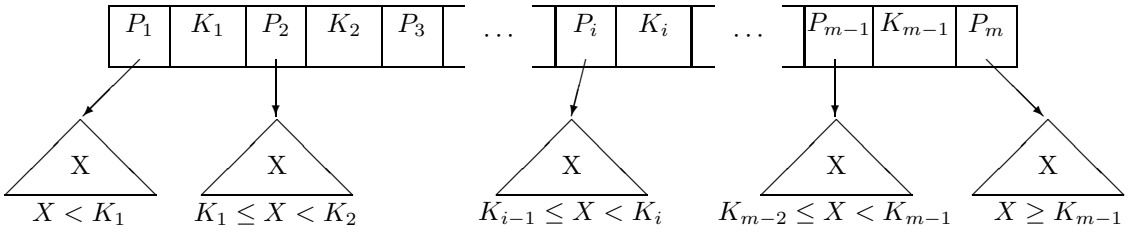
De leaf bevat tot $n-1$ zoeksleutel waarden $K_1, K_2, ..., K_{n-1}$ en $n$ pointers $P_1, P_2, ..., P_n$, waarbij $P_n$ een speciale betekenis heeft.



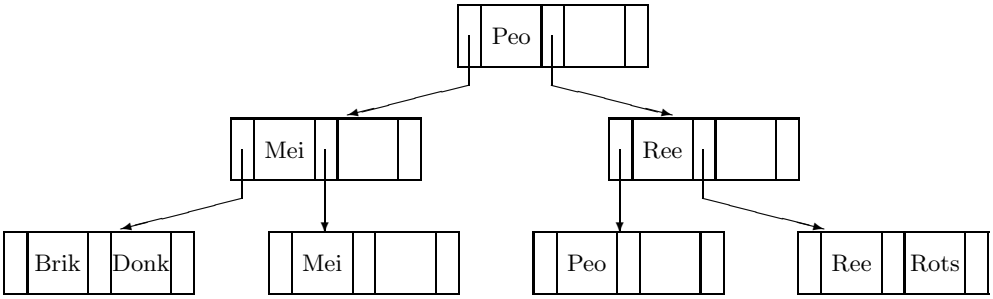

Figuur 19: Een leaf node met pointers ($n = 3$)

Voor $1 \leq i < n$ geldt dat $P_i$ wijst naar een bucket met records met zoeksleutel waarde $K_i$. De zoeksleutel waarden in een leaf worden in gesorteerde volgorde gestockeerd: indien $i < j$ dan $K_i < K_j$. Het is ook mogelijk dat $P_i$ wijst naar de eerste record met zoeksleutel waarde $K_i$; de volgende records met dezelfde zoeksleutel waarde worden aan elkaar geketend met behulp van pointers. Nog een alternatief is dat $P_i$ wijst naar een bucket waar pointers naar elke record met zoeksleutel waarde $K_i$ gestockeerd zitten (interessant wanneer er verschillende indexen op dezelfde bestand gedefinieerd zijn).
Elke leaf kan tot $n-1$ waarden bevatten en er moeten er minstens $\lceil (n-1)/2 \rceil$ aanwezig zijn. Het bereik van de waarden in elke leaf mogen niet overlappen. Indien $L_i$ en $L_j$ leaves zijn en $i < j$, dan zal elke zoeksleutel waarde in $L_i$ kleiner zijn dan elke zoeksleutel waarde in $L_j$.
Omdat er dus een lineaire orde op de leaves aanwezig is, kunnen ze dus samen geketend worden in deze volgorde. Hiervoor wordt de pointer $P_n$ gebruikt. Hierdoor wordt een efficiënte sequentiële verwerking van het bestand mogelijk.
De non-leaf nodes van de B$^+$-boom vormen een multi-level index op de leaves. De structuur is dezelfde, alleen wijzen de pointers nu naar knooppunten in de boom. Elke node kan tot $n$ pointers bevatten maar er moeten er tenminste $\lceil n/2 \rceil$ inzitten.
Beschouw een node met $m$ pointers (zie figuur 20). Voor $1 < i < m$, wijst pointer $P_i$ naar de subtree die zoeksleutel waarden kleiner dan $K_i$ en groter dan of gelijk aan $K_{i-1}$ bevat. Pointer $P_m$ wijst naar dat deel van de subtree met zoeksleutel waarden groter dan of gelijk aan $K_{m-1}$ . Pointer $P_1$ wijst naar dat deel van de subtree met zoeksleutel waarden kleiner dan $K_1$ .
Een voorbeeld van een B$^+$-boom met $n = 3$ is gegeven in figuur 21. Elke node in de boom behalve de root moet minstens $\lceil n/2 \rceil$ pointers bevatten. Afhankelijk van de waarde van $n$ zal het aantal

[†] eerstvolgende geheel getal, bijv. $\lceil 3/2 \rceil = 2$

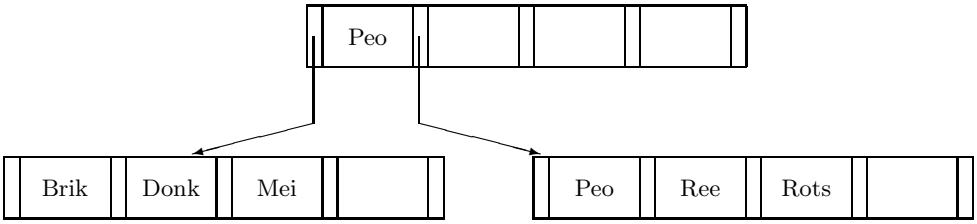



Figuur 20: Een non-leaf node (of internal node)




Figuur 21: B$^+$-boom met n=3

levels in de boom verschillen zodat aan de eis voldaan is. In figuur 22 wordt dezelfde informatie voorgesteld in een B$^+$-boom met $n = 5$.



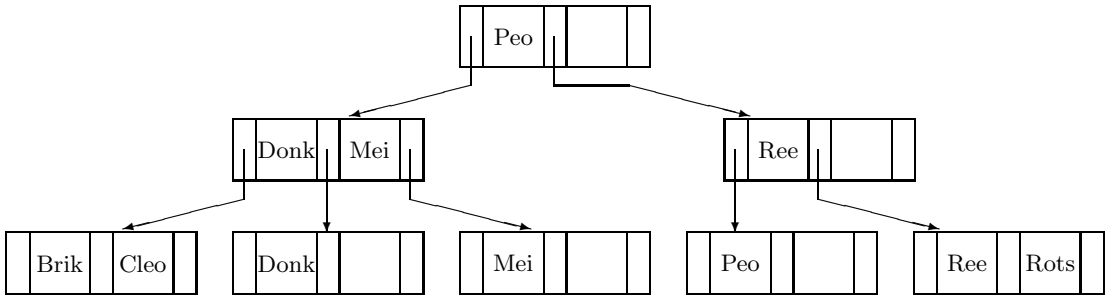

Figuur 22: B$^+$-boom met n=5

Afhandeling van een query in een B$^+$-boom. Onderstel dat alle records met zoeksleutel waarde $K$ moeten gezocht worden. Eerst wordt in de root gezocht naar de kleinste zoeksleutel waarde die groter is dan $K$, namelijk $K_i$. Pointer $P_i$ wordt gevolgd naar een volgende node. Indien $K < K_1$ wordt $P_1$ gebruikt; indien er $m$ pointers in de node zitten en $K \geq K_{m-1}$ dan wordt $P_m$ gebruikt. In de resulterende node wordt gezocht naar de kleinste zoeksleutel waarde die groter is dan $K$ en de corresponderende pointer wordt gevolgd. Op een bepaald moment wordt een leaf bereikt waar de pointer verwijst naar de gewenste records.
Bij de verwerking van een opzoeking wordt dus een pad gevolgd in de boom van de root naar een leaf. Wanneer er $K$ zoeksleutel waarden in het bestand aanwezig zijn, kan dit pad niet langer zijn $\log_{\lceil n/2 \rceil}(K)$. In praktijk betekent dit dat slechts een beperkt aantal nodes moet bezocht worden zelfs wanneer het bestand zeer groot is. Gewoonlijk heeft een node de grootte van een blok. Dit betekent dat $n$ een waarde heeft tussen 10 en 100 (of zelfs meer). Voor een bestand met 1 miljoen zoeksleutel waarden zal een opzoeking slechts tussen 3 en 6 nodes bezoeken.
Toevoegen en verwijderen zijn gecompliceerder dan een opzoeking. Het kan nodig zijn dat een node moet *gesplitst* worden omdat er teveel waarden in moeten ten gevolge van een toevoeging. Of

nodes moeten *gecombineerd* worden omdat er in een node te weinig (minder dan $\lceil n/2 \rceil$) pointers aanwezig zijn. Bij zo'n splitsing of combinatie moet er op gelet worden dat de boom gebalanceerd blijft.
Om het idee achter toevoeging en verwijdering in een $B^+$-boom te introduceren, wordt even verondersteld dat de nodes nooit te groot of te klein worden. Volgende procedures kunnen dan gebruikt worden:

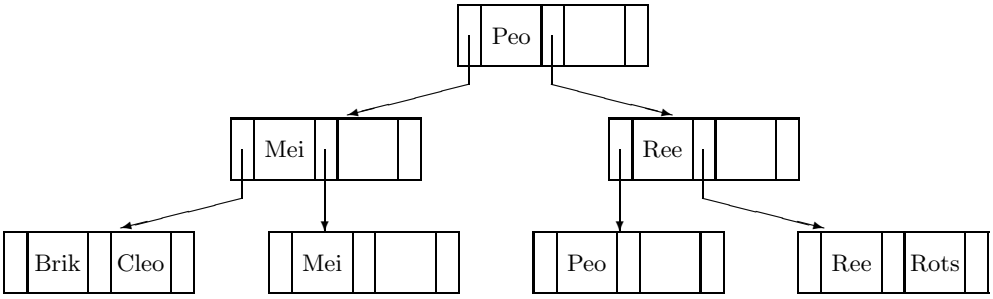
- **Toevoeging**. Met de reeds besproken opzoekingstechniek wordt de leaf gevonden waarin de zoeksleutel waarde zou moeten zitten. Indien de zoeksleutel waarde reeds aanwezig is, wordt de nieuwe record toegevoegd aan de bucket. Indien dit niet zo is, wordt de zoeksleutel waarde in de leaf toegevoegd in die positie zodat de zoeksleutels nog steeds in volgorde zitten. Er wordt een nieuwe bucket gecreëerd waarin het nieuwe record toegevoegd wordt.
- **Verwijdering**. Met de opzoekingstechniek wordt de te verwijderen record gevonden. De record wordt uit de bucket verwijderd. Indien de bucket hierdoor leeg wordt, wordt de zoeksleutel waarde uit de leaf verwijderd.

Het kan echter zijn dat een node moet gesplitst worden. Dit is het geval wanneer er $n-1$ zoeksleutel waarden in de leaf aanwezig zijn en er één moet toegevoegd worden. Van deze $n$ waarden worden de eerste $\lceil (n-1)/2 \rceil$ waarden in de bestaande node gelaten en de overige waarden worden in een nieuwe node geplaatst.
Deze nieuwe leaf moet toegevoegd worden aan de $B^+$-boom door in de parent node de kleinste zoeksleutel waarde van deze nieuwe leaf toe te voegen. Het is mogelijk dat in deze parent geen plaats meer is zodat deze ook moet gesplitst worden. In het slechtste geval moeten alle nodes langs het pad naar de root gesplitst worden. Wanneer de root zelf moet gesplitst worden, wordt de boom één niveau dieper.



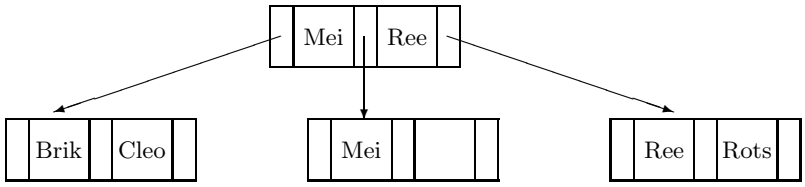

Figuur 23: Toevoeging van **Cleo**

Voorbeeld: toevoeging van **Cleo** tussen "Brik" en "Donk". De zoeksleutel waarde "Cleo" kan niet in de eerste leaf. De leaf wordt gesplitst zoals voorgesteld in figuur 23. De zoeksleutelwaarde "Donk" wordt toegevoegd aan de parent node.
De algemene techniek voor toevoeging in een $B^+$-boom: Bepaal de leaf $l$ waarin de toevoeging moet gebeuren. Indien hieruit een splitsing resulteert, voeg de nieuwe node in de parent van $l$ toe. Indien deze toevoeging een splitsing veroorzaakt, doe dan recursief verder totdat een toevoeging geen nieuwe splitsing veroorzaakt of totdat een nieuwe root gecreëerd wordt.
Bij een verwijdering is het mogelijk dat bepaalde nodes te weinig waarden bevatten. Wanneer **Donk** verwijderd wordt uit de boom van figuur 23, wordt de leaf vrijgegeven en in de parent verdwijnt de zoeksleutelwaarde. Er is geen enkele node die te weinig pointers bevat; het resultaat is voorgesteld in figuur 24.
Om een leaf te verwijderen moet de pointer in de parent verwijderd worden. Hierdoor kan deze parent node zelf te klein worden. Hierin kan echter nog bruikbare informatie aanwezig zijn, zodat de node niet zomaar kan verwijderd worden. Indien in de zuster node nog plaats is om deze informatie op te nemen, worden de records van de twee nodes samengenomen in één node; de andere node wordt verwijderd. Hiervoor moet in de parent een pointer verwijderd worden,



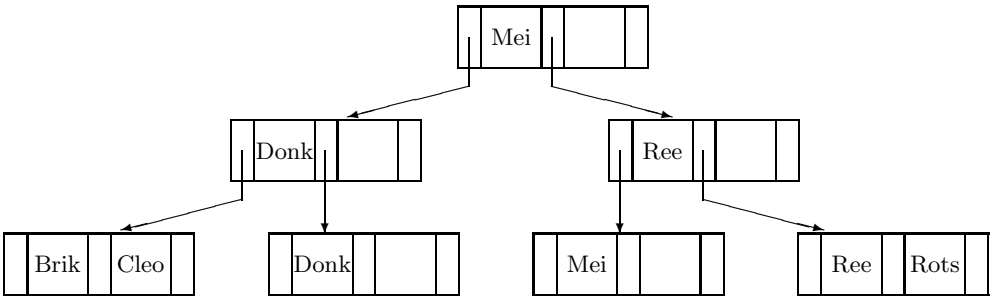

Figuur 24: Verwijdering van **Donk**

... Eventueel wordt op die manier de root bereikt, waar eventueel de laatste pointer verwijderd wordt. In dat geval vermindert de diepte van de $B^+$-boom met 1.
Dit is wat er gebeurt wanneer **Peo** uit de boom van figuur 24 verwijderd wordt. Het resultaat is weergegeven in figuur 25. De leaf met "Peo" wordt verwijderd. In de parent node met "Ree" blijft slechts 1 pointer over. Dit is te weinig. De informatie wordt overgedragen naar de zuster node met "Mei". In de parent node (dit is de root) verdwijnt hierdoor een pointer. Er blijft slechts één pointer in de rootnode over zodat hier geen nuttige informatie meer te vinden is.



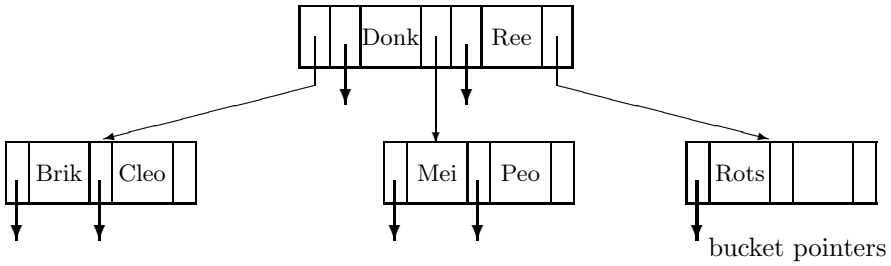

Figuur 25: Verwijdering van **Peo** (uit figuur 24)

Het is echter niet altijd mogelijk om nodes samen te nemen omdat er niet voldoende plaats in de zuster node aanwezig is. In dat geval moeten de pointers *geredistribueerd* worden. Dit resulteert in een verandering van de zoeksleutel waarde in de parent.
Wanneer **Peo** uit de boom van figuur 23 verwijderd wordt, zou de node met "Ree" bij de zuster node ("Donk","Mei") gevoegd moeten worden. Hier is echter geen plaats. De pointers worden herverdeeld zodat elke zuster node er twee heeft. Hierdoor wordt ook de zoeksleutel waarde in de parent node veranderd. Het resultaat is weergegeven in figuur 26.
De algemene techniek voor verwijdering in een $B^+$-boom: Zoek de waarde en verwijder ze. Indien




Figuur 26: Verwijdering van **Peo** (uit figuur 23)

de node te klein wordt, verwijder deze uit de parent. Herhaal dit recursief totdat de root bereikt wordt, een parent voldoende vol blijft na een verwijdering of een samennemen toegepast kan worden.
Alhoewel toevoegen en verwijderen in een $B^+$-boom vrij complex is, zijn toch relatief weinig operaties nodig. Men kan aantonen dat het aantal nodige operaties in het slechtste geval evenredig is met het logaritme van het aantal zoeksleutels. Omwille van de snelheid van verwerken in een $B^+$-boom, wordt deze structuur frekwent gebruikt als index in database implementaties.




Figuur 27: B-Boom equivalent

Een *B-boom* index is gelijkaardig aan een $B^+$-boom index. Het belangrijkste verschil is de eliminatie van redundante stockering van zoeksleutel waarden. In een $B^+$-boom index zijn sommige waarden verschillende malen opgenomen omdat elke zoeksleutel in één of andere leaf moet aanwezig zijn. Een B-boom heeft dus minder ruimte nodig en ook het zoeken kan soms sneller gebeuren. In een non-leaf node moeten echter pointers bijgevoegd worden zodat met twee verschillende structuren moet gewerkt worden. Ook de verwijdering is een complexere operatie.
In figuur 27 is de B-Boom getekend die equivalent is met de $B^+$-boom van figuur 23. De extra pointers in de non-leaf node wijzen naar de buckets van de corresponderende zoeksleutels.

## 7.3 Hashing.

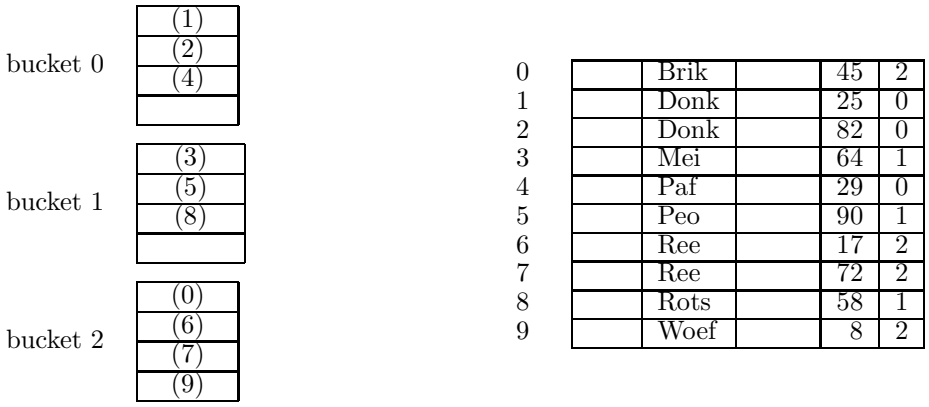
### 7.3.1 Statische Hash functies.

Een nadeel van index schema's is het doorlopen van de index structuur om gegevens te lokaliseren. Bij een *hashing* techniek is dit niet nodig. Het adres van het data element wordt berekend door het toepassen van een functie op de zoeksleutel waarde van het gewenste record.
Een *hash functie* $h$ is een functie van $K$ naar $B$ waarbij $K$ de verzameling van alle zoeksleutel waarden is en $B$ de verzameling van alle adressen.
Principe: Alhoewel de verzameling $K$ van alle mogelijke zoeksleutels groot is (eventueel oneindig), is de verzameling $\{K_1, K_2, ..., K_n\}$ van zoeksleutels die reëel in de database gestockeerd zijn, veel kleiner dan $K$. Op het moment van het ontwerp is niet geweten welke zoeksleutel waarden in de database gestockeerd zullen worden. Maar het is wel zeker dat er te veel mogelijke waarden zijn om de allokatie van een bucket voor ieder mogelijke waarde te verrechtvaardigen. Wat *wel* geweten is op het ontwerp moment, is een benadering voor hoeveel zoeksleutel waarden in de database zullen gestockeerd worden. Het aantal buckets wordt gekozen in overeenstemming met het te verwachten aantal zoeksleutel waarden die in de database zullen gestockeerd worden. De hash functie zal de toekenning van zoeksleutel waarden aan specifieke buckets definiëren.
Hash functies moeten zorgvuldig ontworpen worden. Een slechte hash functie kan in een opzoek tijd resulteren die evenredig is met het aantal zoeksleutel waarden in het bestand. Een goed ontworpen functie geeft een gemiddelde opzoek tijd die constant is, onafhankelijk van het aantal zoeksleutels in het bestand. Dit wordt bekomen door de records vrij uniform over de buckets te verdelen.
Met de hash functie $h$ wordt de opzoeking naar de zoeksleutel $K_i$ uitgevoerd door $h(K_i)$ te berekenen en de bucket te doorzoeken met dat adres. Het is mogelijk dat twee zoeksleutels,

$K_5$ en $K_7$, dezelfde hashwaarde hebben, $h(K_5) = h(K_7)$. Door een opzoeking naar $K_5$ te doen wordt een bucket gevonden waarin records aanwezig zijn met sleutelwaarde $K_5$ maar ook met sleutelwaarde $K_7$. Dus de zoeksleutel waarde van elke record in de bucket moet gecontroleerd worden om er zeker van te zijn dat de record diegene is die gewenst is.
De slechtst mogelijke hash functie beeldt alle zoeksleutel waarden af op dezelfde bucket. Dit is een slechte hashfunctie omdat alle records van het bestand in dezelfde bucket zitten en dus vraagt een opzoeking het aflopen van elke record in het bestand. Een ideale hash functie beeldt elke zoeksleutel waarde af op een verschillende bucket. Deze functie is ideaal omdat elke doorzochte record van de bucket er één is met de gewenste zoeksleutel waarde.
Omdat bij ontwerp niet precies geweten is, welke zoeksleutel waarden in het bestand zullen gestockeerd worden, is een goede hash functie diegene die zoeksleutel waarden aan buckets toekent zodanig dat:

- de distributie uniform is: aan elke bucket wordt hetzelfde aantal zoeksleutel waarden toegewezen uit de verzameling van alle mogelijk zoeksleutel waarden;
- de distributie random is: gemiddeld zijn aan elke bucket ongeveer evenveel waarden toegekend.

Veronderstel dat er beslist is 26 buckets te nemen. De hash functie kan dan gedefinieerd worden als een functie die namen beginnend met de $i$-de letter van het alfabet afbeeldt op de $i$-de bucket. Deze hashfunctie is zeer eenvoudig maar zal niet resulteren in een uniforme distributie, omdat er meer namen zijn beginnend met letters zoals 'B' of 'R' dan er zijn beginnend met 'Q' of 'X'.
Typische hash functies voeren een berekening uit op de interne binaire voorstelling van de characters van de zoeksleutel: bijvoorbeeld de som van de binaire voorstelling van de characters van de sleutelwaarde, modulo het aantal geallokeerde buckets.




Figuur 28: Hash tabel

In figuur 28 wordt een voorbeeld gegeven met drie buckets. De som van de ascii-waarden van de zoeksleutel "Brik" is gelijk aan $66 + 114 + 105 + 107 = 392$. Dit getal modulo 3 geeft het nummer van de bucket, in dit geval is dit bucket 2.
Toevoegen is bijna zo eenvoudig als opzoeken. Indien de zoeksleutel waarde van de toe te voegen record gelijk is aan $K_i$, wordt $h(K_i)$ berekend om de bucket te lokaliseren waarin het nieuwe record toegevoegd wordt.
Verwijdering gebeurt evenzeer zonder complicaties. Indien de zoeksleutel waarde van de te verwijderen record gelijk is aan $K_i$, wordt $h(K_i)$ berekend en in de corresponderende bucket wordt de record opgezocht en dan verwijderd.
Een belangrijk nadeel bij deze hashing techniek is dat de keuze van de hash functie reeds moet gebeuren bij implementatie van het systeem. Omdat de functie $h$ de zoeksleutel waarden naar een vaste verzameling $B$ van bucket adressen afbeeldt, wordt ruimte verkwist wanneer $B$ veel te groot genomen wordt. Wanneer $B$ te klein genomen wordt, zullen de buckets veel verschillende

zoeksleutel waarden bevatten, wat slecht is voor de performantie. Gewoonlijk wordt voor de grootte van $B$ tweemaal het aantal zoeksleutel waarden in het bestand genomen.

### 7.3.2 Dynamische Hash functies.

De noodzaak om de verzameling $B$ van bucket adressen op voorhand vast te leggen, resulteert in een aantal problemen bij de statische hashing techniek. Om het hoofd te bieden aan steeds groeiende databases, zijn er bij statische hashing drie mogelijke opties:

- hash functie kiezen op basis van de huidige file grootte: wanneer de database groeit resulteert dit in performantie degradatie;
- hash functie kiezen op basis van een vermoedelijke file grootte op een bepaald ogenblik in de toekomst: dit leidt initieel tot verkwisting van vrij veel ruimte;
- periodieke reorganisatie van de hash structuur in antwoord op de file groei: dit vraagt de keuze van een nieuwe hash functie, de herberekening van de hash-waarde van elk record in het bestand en de generatie van nieuwe bucket toekenningen: een vrij tijd-kostelijke operatie tijdens dewelke de toegang voor gewone gebruikers verboden is.

Er bestaan verscheidene hashing technieken die toelaten dat de hash functie dynamisch aangepast worden om tegemoet te komen aan de veranderende lengte van de database.

## 7.4 Indexen voor speciale doeleinden

$B^+$-bomen en hashindexen zijn indexstructuren die in verschillende situaties nuttig kunnen gebruikt worden. Daarnaast zijn er een aantal indexstructuren die slechts in heel speciale gevallen kunnen toegepast worden maar daar een grote besparing in geheugenruimte en verwerkingstijd kunnen opleveren.

### 7.4.1 Bitmap index

Een *bitmap index* wordt geïmplementeerd met behulp van één of meerdere bitvectoren. Deze index is geschikt voor attributen die slechts een beperkt aantal verschillende waarden kunnen hebben. Een klassiek voorbeeld is het `geslacht` attribuut in een `Persoon` relatie met slechts twee mogelijke waarden: *vrouw* en *man*. Veronderstel dat de `Persoon` relatie in het totaal 40000 rijen heeft. Een bitmap index op `Persoon` bevat twee bitvectoren, één voor elke mogelijke waarde van het attribuut `geslacht`. Elke bitvector bevat 40000 bits, één voor elke rij in `Persoon`. Indien de $i$-e bit in de *vrouw* bitvector gelijk is aan 1, dan is in de $i$-e rij van `Persoon` het `geslacht` attribuut gelijk aan *vrouw*. De rij-nummers van *vrouwen* kunnen dus geïdentificeerd worden door de *vrouw* bitvector te scannen. Met deze rij-nummer kan dan de offset vanaf het begin van de data in het gegevensbestand berekend worden en kan de gewenste rij onmiddellijk op disk gelokaliseerd worden.
De ruimte die nodig is om deze index te stockeren is 80000 bits groot, of dus 10 Kbytes. Deze ruimte kan gemakkelijk in het primair geheugen voorzien worden. Omdat het doorzoeken van zo'n index dan volledig in primair geheugen kan gebeuren, is een sequentiële scan voldoende efficiënt.
Eigenlijk is in het voorbeeld meer ruimte voorzien dan strict nodig is. Voor het `geslacht` attribuut is één bit per rij voldoende: indien gezet, dan gaat het bijvoorbeeld over een *vrouw*, indien niet, dan bevat de cooresponderende rij een *man*. De reden waarom toch meer ruimte voorzien wordt, heeft te maken met efficiëntie, voornamelijk wanneer twee of meer attributen van een bitmap index voorzien worden. Onderstel bijvoorbeeld dat in onze `Persoon` relatie nog twee bijkomende attributen voorzien zijn, `roker` met waarden *ja* en *nee* en `hartafwijking` met waarden *ja* en *nee*. Wanneer een lijst gevraagd wordt van alle mannen in een bepaalde leeftijdscategorie die roker zijn maar geen hartafwijking hebben, dan ziet een gedeelte van de voorwaarde in de `WHERE` er als volgt uit:

```
geslacht = 'man' AND roker = 'ja' AND hartafwijking = 'nee'
```

Wanneer op deze drie attributen bitmap indexen gemaakt zijn, dan kunnen de rij-nummers van alle tuples die aan deze voorwaarde voldoen gemakkelijk berekend worden: bereken de logische AND van de bitstrings overeenkomstig de *man* waarde van het `geslacht` attribuut, de *ja* waarde van het `roker` attribuut en de *nee* waarde van het `hartafwijking` attribuut. De rij-nummers van de tuples die aan deze voorwaarde voldoen corresponderen met die posities die een 1 hebben in de resulterende bitstring.
Naast de logische AND kunnen ook de OR en NOT operatoren toegepast worden. Wanneer bijvoorbeeld het `leeftijd` attribuut een bitmap index heeft met 120 bitstrings, geeft dit een benodigde ruimte van 120 bits per `Persoon` record. Op zich is dit aanvaardbaar omdat een gewone index tenminste 8 bytes (64 bits) nodig heeft voor elke `Persoon` record. Met behulp van deze bitmap index kan nu efficiënt een lijst met alle rokende mannen tussen de leeftijd van 50 en 80 zonder hartafwijking gevonden worden:

1. bereken de logische AND van de drie bitstrings zoals hoger;
2. bereken de logisch OR van alle bitstrings corresponderend met de leeftijd van 50 tot en met 80 uit de `leeftijd` bitmap index;
3. bereken de logische AND van de twee resultaten: de rij-nummers worden gegeven door die posities die een 1 hebben in deze resulterend bitstring.

Bitmap indexen worden zeer veel toegepast in OLAP toepassingen. In deze toepassingen worden dikwijls queries uitgevoerd die selecteren op attributen met een beperkt aantal verschillende mogelijke waarden. Daarnaast is in deze toepassingen de data vrij statisch: er worden niet frequent tuples toegevoegd of verwijderd. Bitmap indexen zijn bijzonder kostelijk om te onderhouden wanneer de onderlingende data frequent wijzigt.

### 7.4.2 Join index

Om de equi-join op twee relaties $R$ en $S$ met join conditie `R.a = S.b` efficiënter uit te kunnen voeren kan een *join index* gecreëerd worden. Een join index is een collectie met alle paren $(p, q)$ waarbij $p$ de rij-nummer van een tuple $t$ in relatie `R` en $q$ de rij-nummer van een tuple $v$ in relatie `S` is, zodat `t.a = v.b` (de twee tuples voldoen aan de join conditie).
Een join index is normaal lexicografisch gesorteerd op stijgende volgorde van rijnummers: het paar **(3,3)** komt dus voor het paar **(4,2)**. Zo'n index kan ook als een $B^+$ boom of als een hashtabel georganiseerd worden. De waarden van de zoeksleutel in de knopen van de $B^+$ boom zijn gelijk aan de rij-nummers van de rijen in `R`. Om de rij-nummers van alle tuples in `S` te vinden die in de join overeenkomen met een tuple in `R` met rij-nummer $p$, moet de boom doorzocht worden met behulp van zoekwaarde $p$. De pointers in de bladen met zoeksleutel-waarde $p$ zijn de gevraagde rij-nummers. Op een gelijkaardige manier zal bij een hashindex de hashfunctie op waarde $p$ uitgevoerd worden om de bucket te vinden met de gevraagde rij-nummers van tuples in $S$ die in de join overeenkomen.
Een join index kan gezien worden als een vooraf berekende join maar dan gestockeerd op een compacte manier. Alhoewel voor de berekening ervan de volledige join eerst misschien moet uitgevoerd worden, is het voordeel dat de join index incrementeel kan onderhouden worden. Indien nieuwe tuples aan relaties `R` of `S` toegevoegd worden, kunnen nieuwe index entries aan de join index toegevoegd worden zonder dat deze helemaal vanaf nul moet herberekend worden.
Op basis van een join index $\mathcal{J}$ kan het systeem de join gemakkelijk uitvoeren door $\mathcal{J}$ te scannen om alle rij-nummers van `R` en `S` te vinden die overeenkomen.
Een variatie op de join index is de *bitmapped join index*. Een join index wordt normaal georganiseerd als een gesorteerd bestand, een $B^+$ boom of een hashtabel zodat het vinden van de rij-nummers van alle rijen in `S` die in de join overeenkomen met de rij in `R` met rij-nummer $p$ gemakkelijk is. Deze rij-nummers van de tuples in `S` kunnen op de volgende manier gecombineerd worden: voor elke rij-nummer $p$ in `R` worden alle paren van de vorm $(p, q)$ vervangen door één tuple van de vorm

$$(p, \text{bitmap voor de overeenkomende tuples in } S)$$

De bitmap heeft een 1 in de $i$-e positie indien de $i$-e rij in `S` in de join overeenkomt met het tuple in `R` met rij-nummer $p$. Op het eerste zicht lijkt dit een enorme verspilling van ruimte omdat elke bitmap zoveel bits moet bevatten als er tuples zijn in relatie `S`. Dit valt nochtans allemaal wel mee omdat bitmaps gemakkelijk kunnen gecompresseerd worden.

## 7.5 Gereedschap voor de oefeningen

Om na te gaan of indexen gebruikt worden, kan het uitvoeringspad van een query opgevraagd worden. In Oracle kan dit op twee manieren:

- met het `set autotrace on` command; hiervoor moet er een `PLUSTRACE` gecreëerd worden door de DBA en deze moet toegewezen zijn aan de gebruiker die na de uitvoering van de query het uitvoeringspad en bijkomende *trace* informatie wil zien;
- met het `explain plan` command:

  ```
  explain plan for
  select * from artikel;
  ```

  worden records toegevoegd aan de `PLAN_TABLE`; de inhoud van deze tabel kan rechtstreeks opgevraagd worden of het `DBMS_XPLAN` package kan gebruikt worden:

  ```
  select * from table(DBMS_XPLAN.Display);
  ```

Methodes die door Oracle gebruikt worden om rijen op te vragen en te verwerken:

- table access full;
- table access by index rowid;
- index unique scan, index range scan, index full scan, index fast full scan
- merge join, hash join, nested loops, merge join cartesian
- sort join, sort order by, sort group by

# 8 Transaction management

## 8.1 ACID

Een DBMS moet ervoor zorgen dat het effect van een operatie nooit verloren gaat, zelfs niet door een systeem crash. Eén of meer queries en/of aanpassingen kunnen in een *transaction* gegroepeerd worden. Informeel is dit een groep operaties die sequentieel, samen als een eenheid moeten uitgevoerd worden.
Een databanksysteem laat frequent toe dat veel transacties concurrent uitgevoerd worden. De *transaction manager* verzekert dat al deze transacties juist uitgevoerd worden. "Juiste" uitvoering van transacties vereist wat meestal de ACID eigenschappen genoemd worden.

- *Atomair.* Een transactie moet volledig uitgevoerd worden of helemaal niet. Bijvoorbeeld in een geldtransfersysteem moet het overschrijven van een bedrag van één rekening naar een andere een enkelvoudige, atomaire transactie zijn. Het is onaanvaardbaar dat de ene rekening zou gecrediteerd zijn, zonder dat de andere gedebiteerd is, of omgekeerd.

- *Consistentie.* Een databank heeft in het algemeen een notie van een "consistente toestand", een toestand waarin de data overeenkomt met de verwachtingen die de gebruikers ervan hebben. Bijvoorbeeld in een vliegtuigreservatiesysteem is een normale consistentie voorwaarde dat geen enkele plaats toegewezen is aan twee verschillende klanten. Alhoewel het kan zijn dat aan deze voorwaarde voor een korte tijd tijdens de transactie niet voldaan is, moet de transaction manager ervoor zorgen dat de databank aan de veronderstelde consistentie voorwaarden voldoet na het voltooien van de transacties.

  (Soms ook *correctheid*. Transacties vormen een correcte toestand van de databank om in een andere correcte toestand, zonder daarbij noodzakelijkerwijs correctheid te hebben op alle intermediaire punten.)

- *Geïsoleerdheid.* Wanneer twee of meer transacties concurrent uitgevoerd worden, moeten hun effecten los staan van elkaar. Er mogen geen effecten te zien zijn veroorzaakt door het feit dat de twee transacties tegelijk uitgevoerd worden, welke er niet zouden zijn als de twee transacties de één na de ander zou uitgevoerd worden. Bijvoorbeeld in een vliegtuigreservatiesysteem waarbij twee agenten plaatsen verkopen voor dezelfde vlucht en er nog één plaats is, moet de ene aanvraag toegewezen worden en de andere verworpen. Het is onaanvaardbaar dezelfde plaats tweemaal of helemaal niet zou verkocht worden tengevolge van concurrente operaties.

  Bij twee transacties $T_1$ en $T_2$ mag $T_1$ de aanpassingen van $T_2$ zien (nadat $T_2$ commit gedaan heeft), of mag $T_2$ de updates van $T_1$ zien (nadat $T_1$ commit gedaan heeft), maar niet beide.

- *Duurzaamheid.* Wanneer een transactie haar werk voltooid heeft, mag het effect niet verloren gaan na een systeemcrash, zelfs als de faling komt onmiddellijk na de voltooiing van de transactie.

ACID is de afkorting van *Atomicity*, *Consistency*, *Isolation* en *Durability*.

## 8.2 Serializeerbaarheid

Veronderstel dat $A$ en $B$ twee "shared" database elementen zijn, bijvoorbeeld $A$ is de naam van de werknemer met `wnr` gelijk aan 123; en $B$ is de naam van de werknemer met `wnr` gelijk aan 456;

read $A \rightarrow x$

```
SELECT wnaam INTO :x
FROM werknemers
WHERE wnr = "123"
```

write $y \rightarrow A$

```
UPDATE werknemers
SET wnaam = :y
WHERE wnr = "123"
```

**werknemers**

| wnr | wnaam | ... |
|---|---|---|
| 123 | Jef | ... |
| ... | ... | ... |
| 456 | Louis | ... |
| ... | ... | ... |

Om de inhoud van A en B te wisselen, kan volgend programma gebruikt worden:

*Switch*
var $x, y$
begin
read $A \to x$
read $B \to y$
write $y \to A$
write $x \to B$
end

Een *seriële* schedule van twee *Switch* transacties:

| $T_1$ | $T_2$ | $A = \text{Jef}, B = \text{Louis}$ |
|---|---|---|
| read $A \to x$ | | $x = \text{Jef}$ |
| read $B \to y$ | | $y = \text{Louis}$ |
| write $y \to A$ | | $A = \text{Louis}$ |
| write $x \to B$ | | $B = \text{Jef}$ |
| | read $A \to x'$ | $x' = \text{Louis}$ |
| | read $B \to y'$ | $y' = \text{Jef}$ |
| | write $y' \to A$ | $A = \text{Jef}$ |
| | write $x' \to B$ | $B = \text{Louis}$ |
| | | $A = \text{Jef}, B = \text{Louis}$ |

Een *correcte* schedule van twee *Switch* transacties:

| $T_1$ | $T_2$ | $A = \text{Jef}, B = \text{Louis}$ |
|---|---|---|
| read $A \to x$ | | $x = \text{Jef}$ |
| read $B \to y$ | | $y = \text{Louis}$ |
| write $y \to A$ | | $A = \text{Louis}$ |
| | read $A \to x'$ | $x' = \text{Louis}$ |
| write $x \to B$ | | $B = \text{Jef}$ |
| | read $B \to y'$ | $y' = \text{Jef}$ |
| | write $y' \to A$ | $A = \text{Jef}$ |
| | write $x' \to B$ | $B = \text{Louis}$ |
| | | $A = \text{Jef}, B = \text{Louis}$ |

Een *incorrecte* schedule van twee *Switch* transacties:

| $T_1$ | $T_2$ | $A = \text{Jef}, B = \text{Louis}$ |
|---|---|---|
| read $A \to x$ | | $x = \text{Jef}$ |
| read $B \to y$ | | $y = \text{Louis}$ |
| write $y \to A$ | | $A = \text{Louis}$ |
| | read $A \to x'$ | $x' = \text{Louis}$ |
| | read $B \to y'$ | $y' = \text{Louis}$ |
| | write $y' \to A$ | $A = \text{Louis}$ |
| write $x \to B$ | | $B = \text{Jef}$ |
| | write $x' \to B$ | $B = \text{Louis}$ |
| | | $A = \text{Louis}, B = \text{Louis}$ |

**Transaction:** de uitvoering van een programma, bestaande uit een aantal operaties (of acties).

**Schedule van transacties:** rij van acties door transacties met elkaar te *vermengen*.

**Seriële schedule:** een schedule waarin de transacties ononderbroken voorkomen. Wanneer de schedule bestaat uit twee transacties $T_1$ en $T_2$, dan zijn er twee mogelijke seriële schedules: $T_1, T_2$ en $T_2, T_1$. Wanneer er $n$ transacties zijn, zijn er $n!$ mogelijke seriële schedules.

**Correcte schedule:** een schedule die equivalent is met een seriële schedule.

**Equivalente schedules:** twee schedules $S$ en $S'$ bestaande uit dezelfde transacties zijn equivalent wanneer $S(DB) = S'(DB)$ voor elke databank $DB$.

Dit is echter geen praktische karakterisatie van een correcte schedule omdat de gelijkheid moet nagegaan worden voor elke mogelijke databank.

Een verkorte notatie:

| $\underline{T_1}$ | $\underline{T_2}$ | |
|---|---|---|
| read $A \to x$ | | $R_1(A)$ |
| read $B \to y$ | | $R_1(B)$ |
| write $y \to A$ | | $W_1(A)$ |
| | read $A \to x'$ | $R_2(A)$ |
| write $x \to B$ | | $W_1(B)$ |
| | read $B \to y'$ | $R_2(B)$ |
| | write $y' \to A$ | $W_2(A)$ |
| | write $x' \to B$ | $W_2(B)$ |

of

$$R_1(A)R_1(B)W_1(A)R_2(A)W_1(B)R_2(B)W_2(A)W_2(B)$$

**Permuteerbare acties:** twee opeenvolgende acties in een schedule die tot verschillende transacties behoren, en het wisselen van deze twee acties levert een equivalente schedule.

$$R_1(A)R_1(B)W_1(A)\underline{R_2(A)W_1(B)}R_2(B)W_2(A)W_2(B)$$

is equivalent met

$$R_1(A)R_1(B)W_1(A)\underline{W_1(B)R_2(A)}R_2(B)W_2(A)W_2(B)$$

Rekenregels:

$R_1(X)$en $R_2(Y)$ zijn steeds permuteerbaar
$R_1(X)$en $W_2(Y)$ zijn permuteerbaar indien $X \neq Y$
$W_1(X)$en $W_2(Y)$ zijn permuteerbaar indien $X \neq Y$

**Serializeerbare schedule:** een schedule die in een seriële schedule kan getransformeerd worden door herhaaldelijk permuteerbare acties te wisselen. Dus, *elke serializeerbare schedule is correct.* Dit is een eerste realisatie om tot een praktische karakterisatie te komen van correcte schedules: het gaat niet meer over alle mogelijke databanken. Maar is er een algoritme of een strategie die aangeeft welke wisselingen moeten gebeuren om tot een seriële schedule te komen?
Het testen van serializeerbaarheid:

**Observatie:** de schedule $\ldots R_1(A) \ldots W_2(A) \ldots W_1(A) \ldots$ is niet serializeerbaar. Door herhaaldelijk wisselen van permuteerbare acties kan geen enkel van de volgende twee seriële schedules verkregen worden: $\ldots R_1(A) \ldots W_1(A) \ldots W_2(A) \ldots$ of $\ldots W_2(A) \ldots R_1(A) \ldots W_1(A) \ldots$.

**Generalisatie:** voeg de expressie $T_i < T_j (i \neq j)$ toe asa (als en slechts als) de schedule één van de volgende vormen heeft:

$$\ldots R_i(X) \ldots W_j(X) \ldots$$
$$\ldots W_i(X) \ldots R_j(X) \ldots$$
$$\ldots W_i(X) \ldots W_j(X) \ldots$$

Dit is, de schedule bevat een actie van $T_i$ die niet permuteerbaar is met een volgende actie van $T_j$. Men kan dan volgende elegante stelling bewijzen.

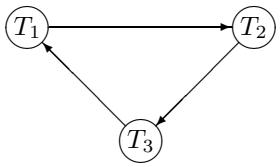
*De schedule is serializeerbaar als en slechts als "$<$" acyclisch is.*

Acyclisch betekent dat de *precedence graph* geen cyclus bevat.
Voorbeelden:

- $R_1(A)W_2(B)W_2(A)R_3(B)W_3(C)W_1(C)$

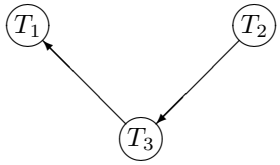
  Wanneer $T_1 = R_1(A)W_1(C)$, $T_2 = W_2(B)W_2(A)$ en $T_3 = R_3(B)W_3(C)$ is, heeft men de volgende expressies: $T_1 < T_2, T_2 < T_3, T_3 < T_1$.
  De schedule is dus niet serializeerbaar.

Precedence Graph




- $R_1(A)W_2(B)R_2(A)R_3(B)W_3(C)W_1(C)$

  Wanneer $T_1 = R_1(A)W_1(C)$, $T_2 = W_2(B)R_2(A)$ en $T_3 = R_3(B)W_3(C)$ is, heeft men de volgende expressies: $T_2 < T_3, T_3 < T_1$.
  $R_1(A)$ en $R_2(A)$ zijn nu permuteerbaar (twee leesacties) en dus is de schedule serializeerbaar en equivalent met $W_2(B)R_2(A)R_3(B)W_3(C)R_1(A)W_1(C)$




In praktijk blijkt het vrij moeilijk te zijn om de serializeerbaarheid van een schedule te testen. De meeste methodes voor concurrentie-controle doen *in feite* geen test op serializeerbaarheid. In plaats daarvan worden protocols gebruikt die garanderen dat de schedule serializeerbaar zal zijn. Dus, dat er "op het einde van de dag" geen cyclus zal zijn. Als op het einde van de dag, na zeer veel transacties uitgevoerd te hebben, gemerkt wordt dat er een cyclus is, is er toch niets meer aan te doen.

## 8.3 Two-Phase Locking

Uit het voorgaande blijkt dat er geen problemen zijn als er alleen `read` transacties zijn; wel als iemand iets leest dat iemand anders aan het schrijven is of als er twee aan het schrijven zijn.
Een *lock* is het verkrijgen van een soort privilege om een item te lezen of een item te schrijven. Met een item wordt een veld of een rij uit een tabel uit een relationele databank bedoeld.
Bij *two-phase locking* (2PL) moeten twee regels gevolgd worden:

1. Voordat een item geschreven wordt, moet de transactie een *exclusieve* lock (X-lock) verwerven op dat item.
   Voordat een item gelezen wordt, moet de transactie een *gedeelde* lock (S-lock) of een exclusieve lock verwerven op dat item.
2. Nadat een lock vrijgegeven is, kan de transactie geen nieuwe locks meer verwerven, zelfs niet op een ander item.

Dit leidt tot volgende herziene versie voor het `Switch` programma:

| | | |
|---|---|---|
| S_lock($A$) | | $S_1(A)$ |
| read $A \to x$ | | $R_1(A)$ |
| S_lock($B$) | | $S_1(B)$ |
| read $B \to y$ | | $R_1(B)$ |
| X_lock($A$) | | $X_1(A)$ |
| write $y \to A$ | | $W_1(A)$ |
| X_lock($B$) | de volgorde van dit en het volgende statement | $X_1(B)$ |
| Unlock($A$) | is belangrijk: ze kunnen NIET gewisseld worden | $U_1(A)$ |
| write $x \to B$ | | $W_1(B)$ |
| Unlock($B$) | | $U_1(B)$ |

**Lock management.**

1. Een **S_lock($A$)** aanvraag door transactie $T_i$ wordt genoteerd als $S_i(A)$, een **X_lock($A$)** aanvraag door $T_i$ als $X_i(A)$ en een **Unlock($A$)** aanvraag door $T_i$ als $U_i(A)$.
2. Lock aanvragen kunnen toegewezen of geweigerd worden door de lock manager. Indien een lock aanvraag geweigerd wordt, wordt deze in een wachtrij geplaatst en de transactie die de lock aanvraagt, zal blijven wachten totdat de lock aanvraag kan toegewezen worden.

3. Indien $S_i(A)$ toegewezen is, dan heeft $T_i$ een gedeelde lock op $A$ totdat de transactie een $U_i(A)$ doet. Indien $X_i(A)$ toegewezen is, dan heeft $T_i$ een exclusieve lock op $A$ totdat de transactie een $U_i(A)$ doet.
4. De lock manager zorgt ervoor dat, indien een transactie een exclusieve lock op een item heeft, er geen andere transactie een gedeelde of exclusieve lock op hetzelfde item kan hebben. De lock manager moet terzelfdertijd mogelijke *starvation* en *deadlock* problemen oplossen.

*Starvation*: je blokkeert dezelfde transactie, zodat deze zeer lang zou kunnen moeten wachten en eventueel nooit zal eindigen.
*Deadlock*: transacties zijn aan het wachten op andere transacties, die een lock moeten vrijgeven, maar deze transacties zijn zelf ook aan het wachten.
Een *locktabel element* voor een item $A$ bevat een triplet:

( $A$, `granted-set`, `wachtlijst` ).

De `granted-set` bevat alle privileges die aan de verschillende transacties gegeven zijn. Dit is een niet-geordende verzameling, bijvoorbeeld $\{S_1, S_2\}$. Indien de `granted-set` niet leeg is, bevat ze ofwel één enkele X-lock, ofwel één of meerdere S-locks. De `wachtlijst` is de lijst van wachtende transacties: een lijst met alle outstanding lock aanvragen die niet onmiddellijk voldaan kunnen worden. Dit is een geordende lijst, bijvoorbeeld $< X_1, X_2 >$.

- Een $S_i(A)$ aanvraag wordt toegewezen wanneer de `granted-set` van het locktabel element voor $A$ nul, één of meerdere S-locks bevat en wanneer de `wachtlijst` leeg is. In de andere gevallen wordt de lock aanvraag op de wachtlijst geplaatst en de transactie moet wachten. Op die manier wordt *starvation* vermeden.
- Een $X_i(A)$ aanvraag wordt toegewezen wanneer de `granted-set` van het locktabel element voor $A$ gelijk is aan de lege verzameling $\{\}$ of aan het singleton $\{S_i\}$ (lock upgrade). In de andere gevallen wordt de lock aanvraag op de wachtlijst geplaatst. Indien de `granted-set` onder meer $S_i$ bevat, wordt de lock aanvraag vooraan in de `wachtlijst` toegevoegd om *deadlock* te vermijden.

**Voorbeelden** :

| voor | aanvraag | | na |
|---|---|---|---|
| $(A, \{S_1, S_2\}, <>)$ | $S_3(A)$ | *ok* | $(A, \{S_1, S_2, S_3\}, <>)$ |
| $(A, \{S_1, S_2\}, < X_3 >)$ | $X_4(A)$ | | $(A, \{S_1, S_2\}, < X_3, X_4 >)$ |
| $(A, \{S_1, S_2\}, < X_3, X_4 >)$ | $S_5(A)$ | (1) | $(A, \{S_1, S_2\}, < X_3, X_4, S_5 >)$ |
| $(A, \{S_1, S_2\}, < X_3 >)$ | $X_2(A)$ | (2) | $(A, \{S_1, S_2\}, < X_2, X_3 >)$ |
| $(A, \{S_1\}, < X_3 >)$ | $X_1(A)$ | (3) | $(A, \{X_1\}, < X_3 >)$ |

(1): als $S_5(A)$ toegewezen wordt en daarna nog een paar $S_i(A)$, dan kan het zeer lang duren voor $X_3$ en $X_4$ toegewezen worden. Om starvation te vermijden, wordt $S_5(A)$ bij op de `wachtlijst` geplaatst. ($S_5(A)$ zou kunnen toegewezen worden, want de granted-set bevat alleen maar S-locks; de wachtlijst is echter niet leeg.)
(2): $X_2$ kan niet achteraan toegevoegd worden, omdat $T_3$ aan het wachten is op het vrijgeven van $S_2$; dit zal pas gebeuren als $T_2$ de lock niet meer nodig heeft, wat wil zeggen nooit omdat $T_2$ nog een $X_2$ achteraan op de wachtlijst zou hebben staan (eens een transactie een lock vrijgeeft, kan het geen nieuwe locks meer aanvragen); een voorbeeld van deadlock.
(3): lock upgrade om geen deadlock te hebben.

**Voorbeeld** : $S_1(A)S_2(A)X_3(A)X_1(A)S_4(A)U_2(A)$

| na uitvoering van | granted-set | wachtlijst voor $A$ | |
|---|---|---|---|
| $S_1(A)$ | $\{S_1\}$ | $<>$ | |
| $S_1(A)S_2(A)$ | $\{S_1, S_2\}$ | $<>$ | |
| $S_1(A)S_2(A)X_3(A)$ | $\{S_1, S_2\}$ | $< X_3 >$ | |
| $S_1(A)S_2(A)X_3(A)X_1(A)$ | $\{S_1, S_2\}$ | $< X_1, X_3 >)$ | vooraan<br>deadlock vermijden |
| $S_1(A)S_2(A)X_3(A)X_1(A)S_4(A)$ | $\{S_1, S_2\}$ | $< X_1, X_3, S_4 >)$ | geen starvation<br>niet-lege wachtlijst |
| $S_1(A)S_2(A)X_3(A)X_1(A)S_4(A)U_2(A)$ | $\{X_1\}$ | $< X_3, S_4 >)$ | lock upgrade<br>singleton$\{S_1\}$ |

**Voorbeeld** :

$$R_1(A)R_1(B)W_1(A)R_2(A)R_2(B)W_2(A)W_1(B)W_2(B)$$
$$\overbrace{\underline{X_1(B)U_1(A)}}$$

Na het toevoegen van lock aanvragen begint de schedule met:

$$\underline{S_1(A)}R_1(A)\underline{S_1(B)}R_1(B)\underline{X_1(A)}W_1(A)$$

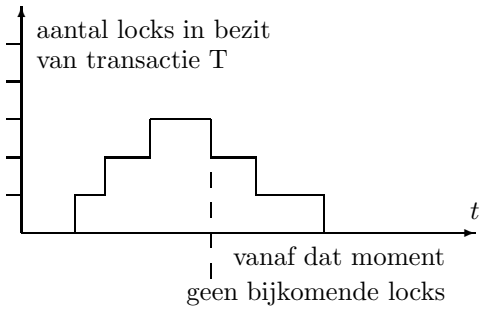
$T_1$ heeft nu een X-lock op $A$ (en een S-lock op $B$). Indien $T_2$ begint aan $\underline{S_2(A)}R_2(A)$, wordt de lock aanvraag afgewezen en $T_2$ moet wachten. $T_2$ kan niet verder doen totdat $T_1$ $\underline{X_1(B)U_1(A)}$ heeft uitgevoerd. Op dat moment kan $T_2$ verder doen en $\underline{S_2(A)}R_2(A)$ uitvoeren. Wanneer dan $T_2$ verder doet met $\underline{S_2(B)}R_2(B)$, zal $T_2$ weer moeten wachten. De uitgevoerde schedule kan er eventueel uitzien als volgt:

$$R_1(A)R_1(B)W_1(A)R_2(A)W_1(B)R_2(B)W_2(A)W_2(B)$$

Dit is een serializeerbare schedule.

Stelling: 2PL garandeert serializeerbeid en dus correctheid.

Verklaring van de naam:
**groeifase**: locks worden verworven, geen locks worden vrijgegeven;
**krimpfase**: locks worden vrijgegeven, geen locks worden verworven;
Het maximaal aantal locks wordt bereikt op het *lockpunt*.




Maar er zijn toch problemen met 2PL:

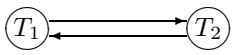
**Deadlock probleem.** Onderstel $X_1(A)X_2(B)S_1(B)S_2(A)$. De lock tabel zal er eventueel als volgt uitzien:

| item | granted-set | wachtlijst |
|---|---|---|
| A | $\{X_1\}$ | $< S_2 >$ |
| B | $\{X_2\}$ | $< S_1 >$ |

$T_2$ is aan het wachten totdat $T_1$ zijn X-lock op $A$ vrijgeeft, terwijl $T_1$ aan het wachten totdat $T_2$ zijn X-lock op $B$ vrijgeeft. Dit is niet te vermijden, kan niet met een andere schedule opgelost worden.

Veronderstel een aantal transacties, geordend op starttijd, $T_1, T_2, T_3, \ldots$. In deze rij is $T_1$ de oudere en $T_2$ de jongere transactie. De starttijd wordt gebruikt als "timestamp" voor elke transactie.

Teken een "wait-for" graaf: nodes corresponderen met transacties.
Er is een pijl van $T_i$ naar $T_j$ indien $T_i$ aan het wachten is totdat $T_j$ een lock vrijgeeft. Er is een deadlock asa de "wait-for" graaf een cyclus bevat. Dus een oudere is aan het wachten op een jongere en een jongere is aan het wachten op een oudere.




Dit moet voorkomen worden door een jongere transactie nooit te laten wachten op een oudere transactie. Als dit kan verzekerd worden, dan kan geen cyclus ontstaan.
Er zijn twee mogelijkheden voor **deadlock preventie**.

- *Wait-die*: een jongere transactie wacht niet totdat een oudere transactie een lock vrijgeeft; in plaatst daarvan *sterft* de transactie. In het voorbeeld sterft $T_2$ wanneer $S_2(A)$ uitgevoerd wordt.
- *Wound-wait*: een oudere transactie wacht niet totdat een jongere transactie een lock vrijgeeft; in plaatst daarvan *kwetst* de transactie de jongere transactie. In het voorbeeld wordt $T_2$ gekwetst wanneer $T_1$ $S_1(B)$ uitvoert.

Mislukte transacties worden herstart met dezelfde timestamp, zodat de transactie ooit de oudste zal zijn. Vermits in beide schema's de oudere transactie nooit mislukt, zal er geen transactie zijn die voortdurend blijft mislukken. Merk op dat in Wait-Die een transactie die alle benodigde locks heeft, nooit zal mislukken omwille van deadlock redenen. Wound-wait is meer aggressief: zelfs als de transactie bijna gedaan heeft, kan ze er nog uit gegooid worden omdat er een oudere transactie aankomt.
In de meeste commerciële systemen wordt **deadlock detectie** gebruikt. Het systeem inspecteert van tijd tot tijd de "wait-for" graaf. Indien een cyclus gevonden wordt, zal één of meer transacties mislukken om de cyclus te breken. De keuze welke transactie gestopt wordt, wordt gedaan door de lock manager. Deze transactie zal dan later herstart worden.
Zowel deadlock preventie als deadlock detectie doen transacties mislukken. Er zijn nog andere redenen waarom transacties mislukken: deling door nul, spanningsonderbreking, disk crash. De vraag is wel hoe er kan gezorgd worden dat de databank in een consistente toestand blijft.
Bijvoorbeeld in een geldtransfersysteem moet eerst rekening A gedebiteerd worden en dan rekening B gecrediteerd. Veronderstel dat de transactie mislukt na het debiteren van A maar voor het crediteren van B.
Het succesvol einde van een transactie: een *commit*. Bij een niet-succesvol einde zullen alle uitgevoerde veranderingen moeten ongedaan gemaakt worden.
Naast deadlock, is er een tweede ernstiger probleem: niet alle 2PL schedules zijn herstelbaar.

| $T_1$ | $T_2$ |
|---|---|
| $X_1(A)$ | |
| $W_1(A)$ | |
| $U_1(A)$ | |
| | $X_2(A)$ |
| | $R_2(A)$ |
| | $W_2(A)$ |
| | $U_2(A)$ |
| andere acties | |
| $T_1$mislukt | |

Wanneer $T_1$ mislukt, moeten alle acties van $T_1$ ongedaan gemaakt worden. Dus, item $A$ moet terug zijn originele waarde krijgen. Probleem is dat de aanpassingen door $T_1$ ook zichtbaar zijn voor de andere transacties. In dit geval heeft $T_2$ het aangepaste item $A$ gelezen. Dus, wanneer $T_1$ ongedaan moet gemaakt worden, zou dat ook met $T_2$ moeten gebeuren.

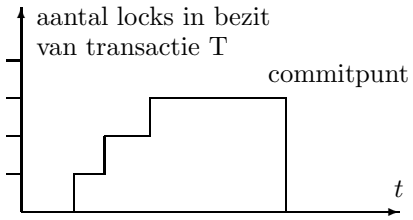
- Indien $T_2$ nog geen commit heeft gedaan, *cascading abort*.
- Indien $T_2$ wel een commit heeft gedaan, dan kunnen deze acties niet meer ongedaan gemaakt worden.

Om het probleem van herstelbaarheid op te lossen moet een aanpassing aan het protocol gebeuren: **strict 2PL**:

- $T_2$ mag de updates van $T_1$ niet zien, voordat $T_1$ een commit heeft gedaan.
- Strict 2PL = 2PL + *een transactie kan geen items unlocken voordat de commit gedaan is.*

- Strict 2PL garandeert serializeerbaarheid (het is een sterker protocol).
- Voordelen ten opzichte van 2PL:
  1. Strict 2PL garandeert herstelbaarheid.
  2. Strict 2PL is transparant voor de programmeur.
- Strict 2PL lost het deadlock probleem niet op, omdat dat in de groeifase zit, en die is niet verdwenen.

Transparantheid: in 2PL moet de programmeur zelf de unlock doen. In strict 2PL mogen slechts na de commit unlocks gebeuren: ze kunnen dus in de commit geïntegreerd worden. Bij strict 2PL moet ook geen expliciete lock gebeuren, deze kan impliciet in de read en de write gestopt worden.




Zo'n expliciete lock wordt in 2PL (soms) te vroeg gedaan, zodat op een ander item een unlock kan gedaan worden; maar in strict 2PL bestaan geen unlocks meer; ze zijn geïntegreerd in commit.

Strict 2PL is geïmplementeerd in de meeste commerciële systemen.

## 8.4 Timestamp gebaseerde concurrentie controle

Dit is een andere manier om te garanderen dat alle schedules serializeerbaar zijn.

- Transacties worden genummerd volgens hun starttijd: $T_1, T_2, \ldots$.
- Bij $n$ transacties zijn er $n!$ mogelijke seriële schedules. Het idee is het verzekeren dat elke schedule equivalent is met de seriële schedule $T_1 T_2 \ldots$.
- Read operaties $R_j(A)$ die moeten vermeden worden:

$$\ldots W_i(A) \ldots R_j(A) \ldots \quad \text{met } i > j$$

- Write operaties $W_j(A)$ die moeten vermeden worden:

$$\ldots R_i(A) \ldots W_j(A) \ldots \quad \text{met } i > j$$

$$\ldots W_i(A) \ldots W_j(A) \ldots \quad \text{met } i > j$$

**Algoritme.** $RTS(A)$ (read time stamp): geeft de laatste transactie aan die item A gelezen heeft. $WTS(A)$ (write time stamp): geeft de laatste transactie aan die item A geschreven heeft.

1. $T_j$ wenst item A te lezen ($R_j(A)$):
   - wordt uitgevoerd indien $WTS(A) \leq j$
   - $T_j$ mislukt in de andere gevallen
2. $T_j$ wenst item A te schrijven ($W_j(A)$):
   - wordt uitgevoerd indien $RTS(A) \leq j$ en $WTS(A) \leq j$
   - $T_j$ mislukt in de andere gevallen

**Optimisatie:** Write regel van Thomas. In de rij $\ldots W_i(A) \ldots W_j(A) \ldots$ met $i > j$, kan $W_j(A)$ genegeerd worden indien $RTS(A) \leq j$.

1. $T_j$ wenst item A te lezen ($R_j(A)$):
   - wordt uitgevoerd indien $WTS(A) \leq j$
   - $T_j$ mislukt in de andere gevallen
2. $T_j$ wenst item A te schrijven ($W_j(A)$):

- wordt uitgevoerd indien $RTS(A) \leq j$ en $WTS(A) \leq j$
- wordt genegeerd indien $RTS(A) \leq j < WTS(A)$
- $T_j$ mislukt in de andere gevallen

Voorbeelden:

- $W_2(A)R_3(A)W_1(A)$: de laatste actie staat niet in de juiste volgorde, had eerder moeten gebeurd zijn, dus de transctie mislukt.
- $R_1(A)W_3(A)W_2(A)$: de laatste actie staat niet in de juiste volgorde, zou eerder moeten gebeurd zijn; in plaats van $T_2$ te laten mislukken, kan $W_2(A)$ genegeerd worden en kan $T_2$ doorlopen. Indien $W_2(A)$ op de juiste plaats had gestaan, dan had het overschreven geweest door $W_3(A)$. Indien niemand het gezien heeft (er mag geen $R_i(A)$ tussenin gebeurd zijn), kan het dus evengoed weggelaten worden.

**Merk op.** Er is een verschil tussen timestamp ordening en two-phase locking:

- In 2PL is een schedule serializeerbaar als de schedule equivalent is met een seriële schedule toegelaten door het locking protocol.
- Bij timestamp ordening is de schedule equivalent met één specifieke seriële orde, namelijk die overeenkomt met de volgorde van de transactie timestamps.

## 8.5 Oefeningen

1. Beschouw de schedule: $R_1(A)W_2(B)W_3(A)W_1(B)Commit_1Commit_2Commit_3$
   en de volgende concurrentie controle protocols: 2PL, strict 2PL en Timestamp gebaseerd met "Thomas' Write" regel. Welke van deze protocols laten deze schedule toe, of laten de acties toe *in de getoonde volgorde*? Voor 2PL moeten lock en unlock aanvragen toegevoegd worden waarbij onderscheid moet gemaakt worden tussen gedeelde en exclusieve locks. Voor het Timestamp gebaseerde protocol moet de equivalente serializeerbare schedule geconstrueerd worden.
2. Zelfde vraag voor $R_1(A)W_2(A)W_1(A)W_3(A)Commit_1Commit_2Commit_3$
3. Zelfde vraag voor $R_1(A)W_1(B)R_2(B)R_3(B)W_2(B)W_2(A)Commit_1Commit_2Commit_3$
4. Beschouw de volgende rij van acties, getoond in de volgorde zoals ze aan het DBMS aangeboden worden: $R_1(A)R_2(B)W_2(A)W_1(B)Commit_1Commit_2$.
   Beschrijf hoe het concurrentie controle mechanisme de rij afhandelt voor de volgende mechanismes:
   - Strict 2PL met deadlock detectie; maak de "waits-for" graaf indien er een deadlock cyclus is.
   - Strict 2PL met wait-die deadlock preventie.
   - Strict 2PL met wound-wait deadlock preventie.

   Indien een transactie geblokkeerd wordt, veronderstel dan dat al haar acties moeten wachten totdat de transactie herneemt; het DBMS doet voort met de volgende actie (volgens de gegeven rij) van een niet-geblokkeerde transactie. De herstart van mislukte transacties moeten niet getoond worden, dus wanneer een transactie mislukt, kunnen al haar acties van de gegeven rij verwijderd worden.
5. Zelfde vraag voor $R_1(A)R_2(B)W_1(B)Commit_1Commit_2$.

## 8.6 Voorbeelden waarom concurrentie controle nodig is

In een zeer eenvoudig reservatiesysteem wordt een record gestockeerd voor elke vlucht. Elke record bevat het aantal gereserveerde stoelen naast een aantal andere informatieelementen. Transactie $T_1$ transfereert $N$ reservaties van een vlucht waarvan het aantal reserveerde stoelen gestockeerd is in databaseelement $X$ naar een andere vlucht, waarvan het aantal reserveerde stoelen gestockeerd is in databaseelement $Y$. Transactie $T_2$ reserveert $M$ stoelen op een vlucht waarvan het aantal reserveerde stoelen gestockeerd is in databaseelement $X$.

| $T_1$ (transfer) |
|---|
| read_item(X); |
| $X \leftarrow X - N$; |
| write_item(X); |
| read_item(Y); |
| $Y \leftarrow Y + N$; |
| write_item(Y); |

| $T_2$ (reservatie) |
|---|
| read_item(X) |
| $X \leftarrow X + M$; |
| write_item(X); |

**Lost Update** probleem. De operaties van twee transacties die dezelfde databaseelementen aanspreken, zijn zodanig gemengd dat de waarde van sommige databaseelementen onjuist is. Veronderstel dat transacties $T_1$ en $T_2$ ongeveer op hetzelfde moment starten. De operaties van deze transacties zijn in de tijd gespreid zoals in volgende tabel.

| $T_1$ | $T_2$ | | |
|---|---|---|---|
| read_item(X); | | | |
| $X \leftarrow X - N$; | | | |
| | read_item(X) | | |
| | $X \leftarrow X + M$; | | |
| write_item(X); | | | |
| read_item(Y); | | | Item $X$ heeft een onjuiste waarde |
| | write_item(X); | $\longleftarrow$ | omdat de aanpassing door $T_1$ gedaan, overschreven wordt |
| $Y \leftarrow Y + N$; | | | |
| write_item(Y); | | | |

De finale waarde van $X$ is verkeerd, omdat $T_2$ de waarde van $X$ leest *voordat* $T_1$ deze waarde wijzigt in de database. Daardoor zal de aangepaste waarde vanuit $T_1$ verloren zijn.

**Dirty Read** (of Temporary Update) probleem. Wanneer een transactie een databaseelement aanpast en dan afbreekt omwille van één of andere reden, dan kan dit aangepaste element reeds opgevraagd zijn voor dat de originele waarde terug hersteld is. $T_1$ wijzigt item $X$ en breekt dan voortijdig af, waardoor het systeem $X$ terug herstelt op de originele waarde. Maar voor dat dit gebeurt, leest $T_2$ de *tijdelijke* waarde van $X$, welke eigenlijk niet definitief in de database opgeslagen wordt omwille van het falen van $T_1$.

| | | $T_1$ | $T_2$ |
|---|---|---|---|
| | | read_item(X); | |
| | | $X \leftarrow X - N$; | |
| | | write_item(X); | |
| | | | read_item(X) |
| | | | $X \leftarrow X + M$; |
| | | | write_item(X); |
| $T_1$ faalt en moet de waarde van $X$ herstellen op de originele waarde; terwijl $T_2$ de *tijdelijke* onjuiste waarde reeds gelezen heeft | $\longrightarrow$ | read_item(Y); | |

**Incorrect Summary** probleem. Indien één transactie een aggregatie som functie berekent op een aantal records, terwijl een andere transactie sommige van deze records aan het aanpassen is, kan de aggregatie functie sommige waarden gebruiken voor ze aangepast zijn en andere nadat ze aangepast zijn.

| $T_1$ | $T_3$ | | |
|---|---|---|---|
| | $som \leftarrow 0$; | | |
| | read_item(A) | | |
| | $som \leftarrow som + A$; | | |
| | $\vdots$ | | |
| read_item(X); | | | |
| $X \leftarrow X - N$; | | | |
| write_item(X); | | | |
| | read_item(X) | | $T_3$ leest $X$ nadat $N$ er af getrokken is |
| | $som \leftarrow som + X$; | $\longleftarrow$ | en leest $Y$ voordat $N$ er bij geteld is; |
| | read_item(Y) | | het resultaat is een verkeerde som |
| | $som \leftarrow som + Y$; | | |
| read_item(Y); | | | |
| $Y \leftarrow Y + N$; | | | |
| write_item(Y); | | | |

**Unrepeatable Read** probleem. Wanneer een transactie $T$ een item tweemaal leest en dit item wordt door een andere transactie $T'$ gewijzigd tussen de twee leesoperaties, krijgt $T$ *verschillende* waarden te zien bij deze leesoperaties van hetzelfde item. Dit kan bijvoorbeeld voorvallen wanneer tijdens een reservatie transactie de klant de beschikbaarheid van stoelen op verschillende vluchten onderzoekt. Wanneer dan voor een specifieke vlucht gekozen wordt, zal de transactie het aantal stoelen op die vlucht voor de tweede maal gelezen worden voordat de reservatie volledig afgehandeld is.

## 8.7 Isolatieniveau van een transactie

Naast het *lost-update* probleem kunnen er, zoals hoger reeds aangegeven, nog andere soorten problemen optreden. Een *dirty read* treedt op als een bepaalde transactie een gewijzigde record inleest die nog niet in de database is uitgevoerd. *Nonrepeatable reads* vinden plaats als een transactie bepaalde gegevens opnieuw inleest die in de tussentijd aangepast of deels verwijderd zijn door een andere transactie. Een *phantom read* treedt op als een transactie bij het opnieuw inlezen van bepaalde gegevens ontdekt dat tussentijds nieuwe rijen ingevoegd zijn door een andere transactie. Deze nieuwe rijen zijn *phantoms*, omdat ze bij de eerste keer dat de gegevens gelezen werden, nog niet bestonden.
In de ANSI-SQL standaard uit 1992 worden vier *isolatieniveaus* gedefinieerd waarin wordt aangegeven welke van de problemen mogen optreden (zie tabel 5). Het is de taak van de applicatieprogrammeur het type isolatieniveau op te geven zodat het DBMS op overeenkomstige wijze locks kan plaatsen.

| | | isolatieniveau | | | |
|---|---|---|---|---|---|
| | | Read Uncommitted | Read Committed | Repeatable Read | Serializable |
| Probleem geval | Dirty read | mogelijk | niet mogelijk | niet mogelijk | niet mogelijk |
| | Nonrepeatable read | mogelijk | mogelijk | niet mogelijk | niet mogelijk |
| | Phantom read | mogelijk | mogelijk | mogelijk | niet mogelijk |

Tabel 5: Overzicht van isolatieniveaus

## 8.8 Concurrency controle in Oracle

Oracle ondersteunt drie verschillende transactie-isolatieniveaus en staat toe dat toepassingen expliciete locks plaatsen. Het gebruik van expliciete locks is echter af te raden, omdat dergelijke locks het standaardlockinggedrag van Oracle kunnen beïnvloeden en omdat ze de kans op transactiedeadlocks vergroten.
Het zogenaamde *System Change Number* (SCN) is een globale databasewaarde in Oracle die wordt verhoogd als er wijzigingen in de database plaatsvinden. Als een rij wordt gewijzigd, wordt de huidige waarde van het SCN samen met die rij opgeslagen. Tegelijkertijd wordt het before image van de rij in een *rollbacksegement* geplaatst. Dit is een buffer die door Oracle wordt onderhouden voor het terugdraaien en loggen van transacties. Het before image bevat het SCN dat vóór de wijziging in de rij stond. Nadat de wijziging is uitgevoerd, verhoogt Oracle het SCN.
Bijvoorbeeld, bij het SQL-statement

```
UPDATE tabel
SET kolom1 = 'nieuwe-waarde'
WHERE kolom2 = 'iets';
```

wordt de waarde van SCN vastgelegd die geldt op het moment dat het statement start; dit wordt weleens het *statement SCN* genoemd. Terwijl de query wordt verwerkt, d.i. het zoeken naar rijen met `kolom2='iets'`, selecteert Oracle alleen rijen die wijzigingen hebben vastgelegd met een SCN-waarde kleiner dan of gelijk aan het statement SCN. Als een rij wordt aangetroffen met een vastgelegde wijziging en een SCN-waarde groter dan het statement-SCN, wordt in het rollbacksegment gezocht naar een eerdere versie van de rij met een vastgelegde wijziging met een SCN lager dan het statement SCN.
Op deze manier lezen SQL-statements altijd een consistente reeks waarden — waarden die zijn vastgelegd op het moment zelf of net voordat het statement start. Oracle leest alleen vastgelegde wijzigingen; dirty reads zijn niet mogelijk.
Oracle ondersteunt drie transactie-isolatieniveaus.

1. *Read committed*: het standaardisolatieniveau in Oracle, omdat in Oracle nooit gegevenswijzigingen worden gelezen die nog niet zijn vastgelegd. Dirty reads zijn niet toegestaan, maar repeated reads kunnen verschillende gegevens opleveren en ook phantoms zijn mogelijk.
   Elk statement is consistent, maar twee verschillende statements in dezelfde transactie kunnen inconsistente gegevens lezen. Als consistentie op transactieniveau vereist is, moet men gebruik maken van Serializable isolatie.
   Oracle hoeft nooit read-locks te plaatsen vanwege de manier waarop het SCN wordt gebruikt. Als een rij echter moet gewijzigd of verwijderd worden, plaatst Oracle een exclusieve lock op die rij voordat die wijziging of verwijdering uitgevoerd wordt. Als een andere transactie een exclusieve lock op die rij heeft, moet met de uitvoering van het statement worden gewacht. Als de blokkerende transactie wordt opgeheven, kan de wijziging of verwijdering plaatsvinden.
   Als de blokkerende transactie wordt vastgelegd, wordt de nieuwe SCN-waarde aan het statement gegeven, waarna het statement (dus niet de transactie) wordt geannuleerd en opnieuw gestart. Als een statement wordt teruggedraaid, worden wijzingen die al door het statement gemaakt zijn, met behulp van rollbacksegmenten alsnog verwijderd.
   Omwille van de exclusieve locks kunnen deadlocks optreden; deze worden door Oracle met behulp van een *wait-for* graaf gededecteerd en één van de conflicterende statements wordt teruggedraaid.
2. *Serializable*: dirty reads zijn niet mogelijk, repeated reads leveren dezelfde waarde op en phantoms zijn niet mogelijk. Deze vorm van isolatie kan alleen werken als het applicatieprogramma er een rol in speelt. Met behulp van

   ```
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   ```

   binnen een transactie, wordt aangegeven dat gedurende de transactie sprake is van serialiseerbare isolatie. Het is ook mogelijk om het isolatieniveau voor alle transacties in een

sessie aan te passen:

```
ALTER SESSIONS SET ISOLATION_LEVEL SERIALIZABLE;
```

Bij dit niveau slaat Oracle de waarde van het SCN op die geldt op het moment dat de transactie begint; dit wordt weleens het *transactie SCN* genoemd. Tijdens de transactie leest Oracle alleen vastgelegde wijzigingen met een SCN-waarde die kleiner is dan of gelijk aan het transactie SCN. Vandaar dat leesacties altijd herhaalbaar zijn en phantoms niet mogelijk zijn.
Zolang de transactie niet probeert een rij te wijzigingen of te verwijderen die een vastgelegde wijziging heeft met een SCN groter dan de transactie SCN, verloopt de transactie verder normaal. Als de transactie dit wel probeert, genereert een Oracle een "Cannot serialize"-exceptie als de wijziging of verwijdering plaatsvindt. Het applicatieprogramma moet hiervoor exceptieafhandelingscode bevatten: het kan de tot dan toe voorgestelde wijzigingen vastleggen of de hele transactie annuleren of een andere bewerking uitvoeren.
Daarnaast moet een transactie die probeert een rij te wijzigingen of te verwijderen die exclusief gelockt is door een andere transactie of statement, wachten. De transactie kan pas verdergaan als de blokkerende transactie/statement wordt teruggedraaid. Indien de blokkerende transactie vastgelegd wordt, dan genereert Oracle de "Cannot serialize"-exceptie en moet de toepassing die exceptie gaan verwerken.
Ook als een serialiseerbare transactie vanwege een deadlock teruggedraaid wordt, wordt een "Cannot serialize"-exceptie gegenereerd.

3. *Read only*: een transactie leest alleen rijen met vastgelegde wijzigingen met een SCN-waarde die kleiner is dan of gelijk aan het transactie SCN. Als de transactie een rij tegenkomt met vastgelegde wijziging met een SCN-waarde die groter is dan het transactie SCN, gaat Oracle op zoek in de rollbacksegmenten om de rij terug te brengen in de staat waarin deze was vóór het transactie SCN. Bij dit niveau zijn invoegen, wijzigen of verwijderen verboden.

**System Change Number** SCN.
Dit getal kan opgevraagd worden met `GET_SYSTEM_CHANGE_NUMBER`, en functie uit de `DBMS_FLASHBACK` package. Hiervoor moet je wel EXECUTE privilege hebben op de `DBMS_FLASHBACK` package.

```
commit;
variable ascn number;
exec :ascn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
print ascn
```

De `commit` zorgt ervoor dat je de laatste SCN opvraagt.

# 9 Databaseherstel

Computersystemen kunnen falen. Hardware gaat wel eens stuk of programma's bevatten fouten. Al deze fouten kunnen optreden in databasetoepassingen. Omdat een database een belangrijke plaats inneemt binnen een organisatie en door veel mensen gebruikt wordt, is het belangrijk dat die fouten hersteld worden. Het systeem moet zo snel mogelijk weer in een bruikbare toestand gebracht worden, die zo dicht mogelijk in de buurt ligt bij toen het systeem vastliep. Wanneer het systeem weer operationeel is, moeten de gebruikers weten wat ze moeten doen. Sommige bewerkingen moeten opnieuw worden uitgevoerd en gebruikers moeten weten hoever ze terug moeten gaan.
Als er fouten in het systeem optreden, is het onmogelijk het probleem op te lossen en verder te gaan alsof er niets is gebeurd. Zelfs als tijdens de storing geen gegevens verloren zijn gegaan, is het onmogelijk om een multi-usersysteem in exact dezelfde toestand terug te brengen. Daarvoor zou een enorme hoeveelheid overheadgegevens moeten bijgehouden worden. Er bestaan wel twee procedures die de schade zo goed mogelijk proberen te herstellen.

## 9.1 Recovery via reprocessing

Het is niet mogelijk om het systeem in precies dezelfde toestand te brengen waarin het zich bevond toen de storing optrad. Wat men wel kan doen, is teruggaan naar een bekend punt en de verwerking vanaf daar weer op te pakken. De meest eenvoudige vorm van dit type recovery bestaat uit het regelmatig kopiëren van de database (een *database save*) en het bijhouden van een status van alle transacties die sindsdien zijn uitgevoerd. Als dan een fout optreedt, kan de database worden hersteld vanaf de laatste save, waarna alle transacties sinds die save opnieuw worden uitgevoerd. Helaas is deze eenvoudige strategie in de meeste gevallen niet toepasbaar.

1. Het opnieuw uitvoeren van die laatste transacties kost net zoveel tijd als het normaal zou gekost hebben. Als de computer druk bezet is, kan het zijn dat het systeem 'zichzelf nooit inhaalt', omdat er steeds nieuwe transacties bijkomen.
2. Taken die gelijktijdig worden verwerkt in een multi-usersysteem, vinden op asynchrone wijze plaats. De kleinste afwijking in menselijke handelingen (bijvoorbeeld het al of niet lezen van een e-mail voordat er op een prompt gereageerd wordt), kan de volgorde waarin de opeenvolgende transacties worden uitgevoerd, al veranderen.

Het kan dus voorkomen dat klant A oorspronkelijk de laatste stoel op een bepaalde vlucht toegewezen kreeg, maar dat bij het opnieuw uitvoeren van de bewerkingen nadat de storing opgelost is, klant B de gelukkige is. Daarom is recovery via reprocessing meestal geen geschikte oplossing.

## 9.2 Recovery via rollback/rollforward

Ook hier wordt regelmatig een kopie van de database gemaakt (een database save). Daarnaast wordt een logbestand bijgehouden met de wijzigingen van de gegevens sinds de save. Als dan een storing optreedt, zijn er twee mogelijkheden.

1. *Recovery via rollforward*: de database wordt hersteld met behulp van de opgeslagen gegevens en worden alle in de log verwerkte wijzigingen sinds de save opnieuw uitgevoerd. De transacties worden dus niet opnieuw verwerkt, zoals bij reprocessing het geval is, omdat de applicatieprogramma's niet bij de rollforward betrokken zijn.
2. *Recovery via rollback*: wijzigingen die door foutieve of slechts deels uitgevoerde transacties gemaakt zijn, worden ongedaan gemaakt door ze uit de database te verwijderen. Vervolgens worden de geldige transacties die in bewerking waren ten tijde van de storing, opnieuw gestart.

Bij beide methodes is een *log*bestand noodzakelijk, waarin de resultaten van de transacties worden bewaard. In een log worden de gegevenswijzigingen in chronologische volgorde opgeslagen. Transactieresultaten worden eerst naar de log geschreven voordat ze ook werkelijk in de database

worden geplaatst. Op die manier is er ook in het ergste geval, namelijk als het systeem vastloopt in de periode tussen het loggen van een transactie en het resultaat vastleggen in de database, een record van de transactie die nog niet in de database is geplaatst. Als het omgekeerde het geval is, de resultaten van een transactie zijn reeds in de database geschreven voordat ze worden gelogd, bestaat de kans dat de database is gewijzigd zonder dat daarvan een log is bijgehouden. In dat geval kan een gebruiker een transactie opnieuw gaat invoeren, niet wetende dat die al in de database is verwerkt.
Als een storing optreedt, wordt de log zowel gebruikt om transacties ongedaan te maken als opnieuw uit te voeren. Om een transactie terug te draaien, moet de log een kopie bevatten van de waarden van elke databaserecord van vóór de wijziging (*before image*). Een transactie wordt ongedaan gemaakt door de before images van alle wijzigingen opnieuw in de database te zetten.
Om een transactie opnieuw uit te voeren, moet de log een kopie hebben van de waarden van elke databaserecord van ná de wijziging (*after image*). Een transactie wordt herhaald door de after images opnieuw naar de database te kopiëren.

Voorbeeld van een transactielogbestand met een aantal mogelijke gegevenselementen in tabel 6. Elke transactie heeft een unieke naam waarmee de transactie geïdentificeerd wordt en alle images van een gegeven transactie zijn aan elkaar gekoppeld via pointers. De achterwaartse pointer wijst naar de vorige wijziging die door deze transactie is uitgevoerd en de voorwaartse pointer wijst naar de volgende wijziging. Een nul in dit pointerveld betekent het einde van de lijst. Het DBMS-recoverysysteem gebruikt deze pointers om alle records voor een bepaalde transactie op te sporen.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | OT1 | 0 | 2 | 11:42 | START | | | |
| 2 | OT1 | 1 | 4 | 11:43 | MODIFY | KLANT 100 | (oude waarde) | (nieuwe waarde) |
| 3 | OT2 | 0 | 8 | 11:46 | START | | | |
| 4 | OT1 | 2 | 5 | 11:47 | MODIFY | VK AA | (oude waarde) | (nieuwe waarde) |
| 5 | OT1 | 4 | 7 | 11:47 | INSERT | ORDER 11 | | (waarde) |
| 6 | CT1 | 0 | 9 | 11:48 | START | | | |
| 7 | OT1 | 5 | 0 | 11:49 | COMMIT | | | |
| 8 | OT2 | 3 | 0 | 11:50 | COMMIT | | | |
| 9 | CT1 | 6 | 10 | 11:51 | MODIFY | VK BB | (oude waarde) | (nieuwe waarde) |
| 10 | CT1 | 9 | 0 | 11:51 | COMMIT | | | |

Tabel 6: Voorbeeld van een transactielogbestand

Andere gegevens die in de log opgenomen zijn: het tijdstip van iedere actie, het soort bewerking (START is het begin van een transactie, COMMIT beëindigt een transactie en geeft alle locks vrij), het object dat bewerkt wordt (recordtype en identifier) en de before en after images.

Om een database terug te brengen naar haar recentste save-toestand en vervolgens alle transacties opnieuw toe te passen kan aardig wat tijd in beslag nemen. Om die tijd wat te verkleinen, maken sommige DBMS-producten gebruik van **checkpoints**. Een checkpoint (controlepunt) is een synchronisatiepunt tussen de database en de transactielog. Om een checkpoint uit te voeren, weigert het DBMS nieuwe verzoeken, rondt de verwerking van openstaande verzoeken af en schrijft de buffers naar de schijf. Vervolgens wacht het DBMS op een bevestiging van het besturingssysteem waaruit blijkt dat alle openstaande schrijfverzoeken naar de database en de log succesvol afgerond zijn. Op dat moment worden de log en de database met elkaar gesynchroniseerd en wordt er een checkpointrecord naar de log geschreven. Later kan de database vanaf dit checkpoint worden hersteld en hoeven alleen after images te worden aangebracht voor transacties die starten na het checkpoint.
Checkpoints nemen weinig tijd in beslag en kunnen gemakkelijk drie à vier keer per uur worden aangebacht. Op die manier hoeft er nooit meer dan vijftien of twintig minuten werk te worden hersteld. De meeste DBMS-producten plaatsen zelf automatisch checkpoints zonder menselijke tussenkomst.

# A Oracle SQL*Plus

Basis werking. Starten met `% sqlplus <lognaam>/<lognaam>` of (afgekort `sql`).

Na de prompt kunnen SQL statements ingetikt worden en ook *SQL*Plus* commands:

- `EXIT` of `QUIT`: stoppen.
- `DESCribe <tabelnaam>`: lijst van tabel met veldnamen en types.
- `DESCribe <functienaam>`: functie met returnwaarde en lijst van argumenten.
- `GET <filenaam>`: lees de query uit het genoemde bestand in de query buffer.
- `EDit`: editeer de actuele query buffer.
- `RUN` of `/`: stuur de actuele query buffer naar de backend, om uit te voeren.
- `SAVe <filenaam>`: schrijf de actuele query buffer weg naar het genoemde bestand.
- `List`: druk de inhoud van de query buffer af.
- `@<filenaam>` of `STArt <filenaam>`: lees en voer de queries uit het genoemde bestand uit.
- `SHOW USER`: toon de Oracle gebruikersnaam.
- `SHOW ALL`: toon de waarde van elke parameter in SQL*PLUS.
- `SHOW ERRORS`: toon de fouten na een "compilation with errors"van een PL/SQL blok.
- `HOST <unix_command>` (of `!<unix_command>`): uitvoeren van unix_command.
- `SET SERVEROUTPUT ON`: aangeven dat de output van een PL/SQL procedure door SQL*Plus op het scherm weergegeven wordt.

Datumfuncties:

```
SQL> select sysdate from dual;
SQL> select to_char(sysdate, 'MM') from dual;
SQL> select to_char(sysdate, 'IW') from dual;
SQL> select to_char(sysdate, 'YYYY') from dual;
```

Data dictionary:

```
SQL> select table_name from user_tables;
SQL> select view_name from user_views;
SQL> select table_name,column_name,date_type from user_tab_columns;
SQL> select table_name,search_condition from user_constraints;
```

Importeren van gegevens: `% sqlldr <lognaam>/<lognaam> control=<bestand>.ctl` (of `ladendb <bestand>.ctl`

## SQL*Plus commands voor het definiëren van een rapport.

**SPOOL** : de output van bijvoorbeeld een rapportgeneratieprogramma naar een bestand schrijven, zodat dit bestand afgedrukt kan worden.

```
SPOOL  filenaam | OFF | OUT
```

**COLUMN** : definiëren van het formaat van de kolommen in een rapport. Controle op kolombreedte, kolomtitel, voorstelling van getallen, splitsen van lange waarden in een kolom.

```
COLUMN kolomnaam [ALIAS aliasnaam] [CLEAR]
       FORMAT specificatie  HEADING hoofdingtekst
       NEW_VALUE variabele_naam       [NO]PRINT
       JUSTIFY { LEFT | CENTER | RIGHT }    NULL null_tekst
       TRUNCATED | WORD_WRAPPED | WRAPPED
```

Indien de specificaties te lang zijn, kan een lijn afgesloten worden met een minteken '-' en kan op een volgende lijn verder gegaan worden.

De *aliasnaam* kan in BREAK, COMPUTE en andere COLUMN commands gebruikt worden. Afhankelijk van de kolombreedte kunnen waarden afgekapt, gesplitst op woordgrenzen of om het even waar gesplitst worden.
Formaatspecificatie:

| | | |
|---|---|---|
| 9 | tonen van significante cijfers | |
| 0 | markeren van de plaats vanaf waar leidende nullen getoond worden | |
| A<getal> | voor characters, *getal* geeft de kolombreedte aan | |
| TO_CHAR(datum, formaat) | D | dag van de week |
| | DD | dag van de maand |
| | DY | afgekorte dagnaam |
| | IW | week van het jaar |
| | MM | maandnummer |
| | MON | afgekorte maandnaam |
| | YYYY | jaartal |
| | YY | laatste twee cijfers van het jaartal |

**TTITLE** : definiëren van bovenmarge van een pagina van een rapport.

```
TTITLE [ON|OFF] | SKIP aantal_lijnen | LEFT | CENTER | RIGHT
       FORMAT specificatie | tekst | variable
```

Voorbeelden van `variablen`: *SQL.PNO*, actuele paginanummer, *SQL.LNO*, actuele lijnnummer en *SQL.USER*, Oracle gebruikersnaam van de gebruiker die het rapportprogrmma uitvoert.

**BTITLE** : definiëren van ondermarge van een pagina van een rapport.

```
BTITLE [ON|OFF] | SKIP aantal_lijnen | LEFT | CENTER | RIGHT
       FORMAT specificatie | tekst | variable
```

**BREAK** : definiëren van het afbreken van pagina's en lijnen op basis van veranderingen in kolomwaarden in een rapport. Controle op het al of niet herhalen van waarden in een kolom en het afdrukken van berekende waarden zoals totalen en subtotalen.

```
BREAK [ON { kolomnaam | ROW | REPORT } ]
      [ SKIP {aantal_lijnen | PAGE } ] [ NODUPLICATES | DUPLICATES ]
```

**COMPUTE** : definiëren van berekeningen van subtotalen en totalen, op basis van BREAK commands.

```
COMPUTE { AVG | COUNT | MAX | MIN | NUMBER | STD | SUM | VAR }
        [LABEL tekst] [ OF kolomnaam ]   [ON { kolomnaam | ROW | REPORT } ]
```

**REPHEADER** : definiëren van hoofding van een rapport; wordt afgedrukt op de eerste pagina, voor de eerste detail lijn maar onder de bovenmarge van deze pagina.

```
REPHEADER [ON|OFF] | SKIP aantal_lijnen | LEFT | CENTER | RIGHT
          FORMAT specificatie | tekst | variable
```

**REPFOOTER** : definiëren van voeting van een rapport; wordt afgedrukt op de laatste pagina, na de laatste detail lijn maar voor de ondermarge van deze pagina.

```
REPFOOTER [ON|OFF] | SKIP aantal_lijnen | LEFT | CENTER | RIGHT
          FORMAT specificatie | tekst | variable
```

**CLEAR** : verwijderen van all COLUMN definities, BREAK settings, COMPUTE definities en andere dingen.

```
CLEAR { BREAKS | BUFFER | COLUMNS | COMPUTES | SCREEN | SQL }
```

**SET** : voor rapport en formaat settings:

| | |
|---|---|
| COLSEP | scheidingsteken tussen kolommen |
| LINESIZE | maximum lengte van een rij |
| NULL | tekst die getoond wordt voor een NULL waarde |
| NUMFORMAT | default formaat voor voorstelling van getallen |
| NUMWIDTH | default breedte van een kolom met getalwaarden |
| PAGESIZE | aantal lijnen die op een pagina kunnen gedrukt worden |
| SPACE | de ruimte tussen kolommen |
| TRUNCATE | afkappen van lange lijnen |
| WRAP | splitsen van lange lijnen |

**Voorbeelden.**

```
set pagesize 20
repheader  center 'Een overzicht ' skip 3
ttitle skip 2 right 'De Studenten '
column pro heading 'Roker' format a5
column pnm heading 'Werknnaam' format a15 truncated
column pnr format 999
column pvn heading 'Roepnaam' format a5 word_wrapped
select * from e_persoon order by pnr;
clear breaks columns
```

---

```
set pagesize 40
repheader  left 'Het Verslag ' skip 3
ttitle skip 2 center 'De Studenten '
column pnr heading 'Studnummer' format 999
column bafk heading 'Fase' format a7 truncated
column res heading 'Result' format 99.99
break on pnr noduplicates skip 1 -
  on report
compute avg label 'gemid ' of res on pnr
compute max label 'max ' of res on report
select * from e_deelname order by pnr ;
clear breaks columns computes
```

---

```
set pagesize 30
repheader  left 'Het Verslag ' skip 3
ttitle skip 2 center 'De Studenten ' -
'van het '  format a3 pgs_var ' geslacht' skip 2
column pgs new_value pgs_var noprint
column pnr format 99999
column pnm heading 'Werknnaam' format a15 truncated
column pvn heading 'Roepnaam' format a10 truncated
column pwp heading 'Woonplaats' format a10 truncated
column pro heading 'Roker' format a10
break on pgs noduplicates skip page -
  on pro noduplicates -
      on report
```

```
compute count of pnr on pgs
compute count of pro on pro
compute count of pnr on report
select * from e_persoon order by pgs,pro ;
clear breaks columns computes
```

# B DBMS: oefeningen

## B.1 Beschrijving van de gegevens

Voor het creëren van een uurrooster is informatie in verband met opleidingen, activiteiten, docenten en lokalen nodig. Een *opleiding* wordt gekenmerkt door de afkorting (**afk**), de volledige **naam** en de **fase**. Er zijn twee soorten opleidingen: *basis*opleidingen en *minor*gedeeltes. Zo'n minorgedeelte behoort steeds bij een basisopleiding. Bij elke opleiding wordt een schatting van het aantal studenten (**astd**) gegeven en het aantal groepen (**agrp**) dat moet voorzien worden. Bij een *basis*opleiding wordt een programmacoördinator (**bpc**) aangegeven en of deze basisopleiding al of niet minoren bevat (**bsoort**).
Binnen een opleiding worden *activiteit*en georganiseerd. Een activiteit kan in meerdere opleidingen opgenomen zijn. Per activiteit wordt de **anaam** en het aantal contacturen op semesterbasis (**ascu**) bijgehouden. Er zijn verschillende soorten activiteiten: hoorcolleges *theorie* met vermelding van de docent (**tdafk)**, oefeningen en practica (*oefpra*) waarbij het aantal groepen (**gaant**) wordt aangegeven dat een bepaalde docent toegewezen krijgt; voor *practica* is er nog bijkomende informatie omtrent het lokaal waarin het practicum georganiseerd wordt (**lid**) en het aantal begeleidende docenten (**padoc**).
Per *lokaal* wordt **lnaam**, **lcapa** (maximum aantal studenten) en het al of niet aanwezig zijn van een **lbeamer** bijgehouden.
Activiteiten worden verzorgd door docenten. Per *docent* wordt docentennummer (**dnr**), **dnaam**, **dgeslacht** en geboortedatum (**dgbd**) bijgehouden.
Als extra informatie wordt per activiteit nog een bijkomende *fiche* voorzien met daarin de beschrijving van de doelstellingen (**fdoel**), de inhoud (**finh**), de studiepunten (**fstp**) en de geschatte studielasturen (**fslu**).
Tip. Geef aan elke entiteit ook een ID-attribuut: opleiding (**bafk**, **mafk**), activiteit (**aid**), docent (**dafk**), lokaal (**lid**), ...

## B.2 Formatteringsaspecten van sql*plus

1. Gegeven `vb1.sql`
   Maak er een *mooi* rapport van met onderaan de gebruikersnaam en paginanummer. Laat nu ook de namen in alfabetische volgorde zien.
2. Gegeven `vb2.sql`
   Maak een gelijkaardig rapport maar nu per fase.
3. Gegeven `vb3.sql`
   - Maak een rapport per rokerssoort en daarbij nog eens sorteren per geslacht.
   - Geef een overzicht van de studentnamen per fase met de gemiddelde en maximale resultaten per geslacht. Geef daarbij ook het slechtste resultaat per fase en over alle fasen heen.

## B.3 Fysische organisatie

**Data dictionary views**

```
segment.sql | select segment_name,tablespace_name,extents,blocks,initial_extent,
            |        min_extents, max_extents
            | from user_segments
-------------------------------------------------------------------------------
extent.sql  | select segment_name, segment_type, tablespace_name, bytes,
            |        blocks, extent_id
            | from user_extents
-------------------------------------------------------------------------------
vrij.sql    | select * from user_free_space
```

```
--------------------------------------------------------------------------------
infseg.sql| declare
     2    |    v_owner VARCHAR(30) := 'E400';
     3    |    v_segment_name VARCHAR(30) := 'docent;
     4    |    v_segment_type VARCHAR(30) := 'TABLE';
     5    |    v_total_blocks NUMBER;
     6    |    v_total_bytes NUMBER;
     7    |    v_unused_blocks NUMBER;
     8    |    v_unused_bytes NUMBER;
     9    |    v_last_used_extent_file_id NUMBER;
    10    |    v_last_used_extent_block_id NUMBER;
    11    |    v_last_used_block NUMBER;
    12    | begin
    13    |   dbms_space.unused_space( v_owner,        v_segment_name,
    14    |        v_segment_type,  v_total_blocks, v_total_bytes,
    15    |        v_unused_blocks, v_unused_bytes, v_last_used_extent_file_id,
    16    |        v_last_used_extent_block_id,     v_last_used_block ) ;
    17    |   dbms_output.put_line(v_segment_type || ' : ' || v_owner );
    18    |   dbms_output.put_line('.' || v_segment_name);
    19    |   dbms_output.put_line('total_blocks  : ' || v_total_blocks);
    20    |   dbms_output.put_line('unused_blocks : ' || v_unused_blocks);
    21    | end;
```

Nadat alle tabellen en data verwijderd zijn zie je dat m.b.v. vrij.sql het eerste vrije bloknummer 128 is. Na het uitvoeren van crea1.sql en ladendb met geg1.ctl bepaal je zelf vanaf waar er nog vrije blokken zijn en m.b.v. extent.sql,segment.sql en infseg.sql waar elke tabel met zijn key zit.
Pas rowid.sql aan zodat je van de lokaaltabel het objectnummer kan berekenen. Met behulp van object.sql controleer je je resultaat.
Pas de analyse.sql file aan zodat je al uw tabellen kan analizeren. Noteer het resultaat van allalltables.sql en ga na in welke blokken de data van de lokaal tabel zit.
Voer nu crepersdeel.sql uit en vul deze twee extra tabellen met ladendb e_pers1.ctl en invoegdeelname.sql.
Bepaal terug waar uw eerste vrij blok zit en verklaar met behulp van extent.sql en segment.sql en infseg.sql waar de tabellen en de data zitten. Ga ook eens kijken naar /u03/oradata/erp . Als je de e_persoon tabel analizeert krijg je 20 blokken te zien. Verklaar dit resultaat. Na ladendb van e_pers2.ctl beantwoord je terug dezelfde vragen maar nu is het aantal blokken wel 35 geworden.
Hoe groot is uw .dbf file (/u03/oradata/erp/e4??.dbf) geworden? Hoeveel extra blokken zijn erbij gekomen?
Bespreek in welke volgorde oracle de databloks alloceert op basis van de e_persoon tabel.
Bepaal voor de e_persoon tabel hoeveel records er in een blok gemiddeld kunnen. Verwijder nu de activiteit tabel. Opgelet vergeet niet purge te vragen. Zoek de vrijgekomen blokken op. Voeg nu een nieuwe tabel met key toe en bepaal welke bloknummers deze gekregen heeft.

## B.4 Transaction management

**Bij aanvang.** Noteer voor de belangrijkste oracle processen (checkpointwriter,logfilewriter en de databasewriter) de reeds gebruikte cpu tijden. Kies er nog zelf een proces uit en verklaar waarvoor dit dient. Ga na dat je over de role scn beschikt en vraag het huidige scn nummer op.

**Default Oracle read committed** .

1. Stel een sequentie op die aantoont dat repeated reads (die verschillende waarden geven) mogelijk zijn, maar dirty reads onmogelijk.
2. Stel een sequentie op die aantoont dat de uitvoering van een statement wacht.

3. Stel een sequentie op die aantoont dat phantoms mogelijk zijn.

Als je een record toegevoegd hebt, graag dan nu verwijderen.

**Set transaction isolation level serializable** .

1. Stel een sequentie op die aantoont dat repeated reads nu wel dezelfde waarden geven.
2. Stel een sequentie op die aantoont dat er geen phantoms mogelijk zijn.
3. Stel een sequentie op die aantoont dat de uitvoering van een statement wacht, en zelfs niet kan doorgaan (error krijgen).

Als je een record toegevoegd hebt, graag dan nu verwijderen.

**Set transaction read only** .
In de cursus staat er bij read only:

> Een transactie leest alleen rijen met vastgelegde wijzigingen met een SCN-waarde die kleiner is dan of gelijk aan het transactie SCN. Als de transactie een rij tegenkomt met vastgelegde wijziging met een SCN-waarde die groter is dan het transactie SCN, gaat Oracle op zoek in de rollbacksegementen om de rij terug te brengen in de staat waarin deze was voor het transactie SCN. Bij dit niveau zijn invoegen, wijzigingen of verwijderen verboden.

Stel een sequentie op die dit aantoont.

**Tot slot.** Bekijk terug bij de belangrijkste oracle processen de reeds gebruikte cpu tijden. Welk besluit kan je hieruit nemen?

## B.5 Fysische dataorganisatie

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
| A | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | / |
| B | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
| B | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | / |
| C | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
| C | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | / |
| D | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
| D | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | / |
| E | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
| E | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | / |
| F | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
| F | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | + | / |

## B.6 DBA Views

Met behulp van dynamic performance views (dpv1.sql-dpv9.sql) en views voor de toegekende rechten (vtr1.sql-vtr3.sql) en views voor rollen en rechten (vrr1.sql-vrr7.sql) en views voor tabellen,indexen,... (vti1.sql- vti8.sql) kan je hopelijk vlot de volgende vragen beantwoorden.

- Geef een lijst met procesnaam en procesid van uw oracleprocessen.
- In welke tabel worden er statistieken bijgehouden over de processen.
- Wat is de juiste naam van de control en redofiles en waar staan deze op disk.
- Hoeveel bytes bevat een datablok.
- Hoeveel gebruikersprocessen kunnen tegelijkertijd een verbinding met de databank maken.
- Hoeveel rijen bevat de dba_tab_privs tabel en hoeveel rijen gaan over uzelf.
- Welke sessie mogelijkheden heb jij gekregen.
- Welke rollen heb jij gekregen.
- In welke tabel kan je de eigenschappen van de role "student"vinden.
- Hoeveel rijen bevat de role_tab_privs tabel en hoeveel rijen gaan over uzelf.
- Analyzeer uw tabellen. Geef van uw tabellen het aantal aanwezige rijen en de gemiddelde rijlengte weer. Kan je deze getallen verklaren voor de tabellen docent, lokaal, e_persoon.
- Geef informatie over uw tablespace.
- Waar worden al uw beperkingen die je tijdens uw creatie van tabellen opgeeft bijgehouden. Bij het bekijken van het resultaat zie je in de derde kolom de letters P, C of R staan. Verklaar deze letters.
- Geef van al uw primary_keys het blevel en het aantal leaf_blocks.
- In het begin van het labo zijn we eens gaan kijken in /u01/app/oracle/admin/erp/scripts. In welk bestand vind je info over waar de redofiles op het systeem komen en hoe groot deze gemaakt worden.
- In welk bestand vind je info waar de controlfiles op het systeem komen.

## B.7 Van databankgegevens naar een xml-bestand en omgekeerd

1. Van databank naar xml.
   Gegeven printxml.sql en maakxml.sql. Pas maakxml.sql aan zodat je een procedure (bvb. oracle_to_xml) maakt met als parameter de sql query die je wilt uitvoeren. Het uitvoeren van uw procedure zal er dan als volgt kunnen uitzien:

   ```
   exec oracle_to_xml('select * from basis')
   ```

   Het resultaat wordt naar standaard output gestuurd. Vergeet dus niet `set serveroutput on`. Het is ook mogelijk om rechtstreeks het resultaat naar een bestand te sturen. Hiervoor moet je de procedure die in het bestand printtofile.sql staat in uw nieuwe versie integreren. Gebruik als p_dir XML_STUD_DIR en verwerk in de filenaam uw eigen naam. Voeg deze filenaam toe als extra parameter van uw procedure. In Oracle is de XML_STUD_DIR gekend als /student/e4/hfd/xml. Ga gerust eens kijken of je er een bestand geschreven hebt en wat de inhoud ervan is.
2. Van xml-bestand naar databank.
   Gegeven een xml-bestand (kortbasis.xml). Maak op basis van dit bestand een tabel aan. Schrijf dus een create table instructie en voer uit. Nu gaan we met behulp van insertxml.sql ervoor zorgen dat de data in uw tabel komt.

## B.8 Information Retrieval

Uw gegevens zijn gewist en daarom maken we terug de tabellen met creatie.sql. De gegevens laden we in mbv. ladendb geg.ctl. We gaan in de fiche tabel een veld van type *clob* opnemen en verwijderen daarom de originele fiche tabel en maken een nieuwe met behulp van `fichecreatie.sql`. Het laden van nieuwe gegevens doe je dan met `ladendb tfiche.ctl`. Uw bestaande tabellen kan je opvragen met (usertable.sql):

```
select table_name from user_tables;
```

of met `userobj.sql`. Als we nu vragen naar alle ficheinhouden waar een bepaald woord in voorkomt mbv contains lukt dat niet. Bvb:

```
select * from fiche where contains (finh,'opbouw') > 0
```

We gaan dus een basisindex maken met indexb.sql.
Als je nu al uw tabellen bekijkt merk je dat er 4 nieuwe zijn. Als je al uw objecten bekijkt, merk je dat er ook 7 nieuwe indexen en 2 extra lobs bij zijn. De inhoud van de `$I` tabel kan je opvragen met toonindex.sql en als argument de indexnaam geven. Spool dit resultaat naar een bestand.Verklaar de cijfers in de kolom first, last en count. Nu lukt de bovenstaande select wel en krijg je 3 records te zijn.
Als je in uw vraag bvb een lidwoord zet, krijg je alle teksten als resultaat. Niet echt de bedoeling om stopwoorden te verzamelen. Je kan nu zelf een stopwoordenlijst maken. Voer maakstop.sql uit en maak de index op basis daarvan met indexstop.sql(Eerst basis index verwijderen).
Stel nu terug uw vraag met een lidwoord en bekijk het resultaat. Bekijk terug de `$I` tabel en probeer een verklaring te geven voor het aantal rijen. Om aan te geven dat de teksten in het nederlands staan maken we een nieuwe index.
Voer maakvlaams.sql en indexvlaams.sql uit. Hoeveel rijen telt het `$I` bestand nu en welke woorden kan je nu meer vinden. Probeer het ? eens.
Kan je woorden vinden als je 2 letters van plaats wisselt? Kan je het meervoud vinden als het enkelvoud in de tekst staat en omgekeerd?
Kan je vervoegingen (zoeken met $-teken) vinden, bvb "kennen"staat in de tekst,
kan je "`$ken`"vinden? Probeer ook eens naar "?finterfce"(interface) te vragen.
Als we nog de nederlandse vervoeging willen activeren gebruiken we maakvlstam.sql en indexvlstam.sql. Hoe zit het met het `$I` bestand. Geef een voorbeeld van een woord dat je nu wel vindt met behulp van het `$` symbool en daarjuist niet. Probeer nog eens naar "?wort"(wordt) of "?deploijmend"(deployment) of "?oplijdingsonerdel"(opleidingsonderdeel) te vragen.
Om het helemaal echt te maken laat je toe dat de gebruiker zich serieus vergist ( ei ipv ij of t ipv dt ). Dit kan je bekomen door maakvlstfuz.sql en indexvlstfuz.sql uit te voeren.
Gebruik nu een plsql procedure (f01.sql) die voor een gegeven woord, de inhoud van de teksten waar dat woord in voorkomt, laat zien.
Er is een thesaurus aanwezig met naam fiche_thes. Bekijk de structuur van het fiche_geg bestand waar de relaties tussen de verschillende woorden gegeven zijn. Wat zou het resultaat zijn van de volgende vragen.

- Geef de narrower terms van databank totaan het eerste niveau.
- Geef de narrower terms van databank totaan het tweede niveau.
- Geef de broader terms van databank totaan het eerste niveau.
- Geef de broader terms van databank totaan het tweede niveau.
- Geef de synoniemen van databank.
- Geef de synoniemen van fiche.

Deze thesaurus kan je ook gebruiken in uw contains queries. Bepaal de score in de fiche tabel van de broader terms van het woord java op het tweede niveau. Ga na dat uw resultaat inderdaad het totaal is van de vragen met al de extra termen.

## B.9 Indexen

Gelieve eerst al uw tabellen te analizeren.
Om te weten welke indexen er reeds op uw tabellen aanwezig zijn kan je deze opvragen met behulp van indexlijst.sql .
Maak een PLAN_TABLE met createplan_table.sql. Uitleg over hoe je een PLAN_TABLE moet lezen vind je in het leesplan bestand.
Schrijf voor al deze vragen een explain plan en becommentarieer dan de PLAN_TABLE.

1. Geef alle velden van de lokaaltabel.
2. Geef alle velden van de lokaaltabel met lid gelijk aan een bepaalde waarde.
3. Geef alle velden van de lokaaltabel met lid tussen twee waarden.
4. Geef lid van de lokaaltabel.
5. Geef lid van de lokaaltabel met lid kleiner dan een bepaalde waarde.
6. Geef lcapa van de lokaaltabel.
7. Geef lcapa van de lokaaltabel met lcapa tussen twee waarden (10 en 100).
8. Maak een index op de lcapa.
9. Bekijk nogmaals uw bestaande indexen met behulp van indexlijst.sql.
10. Geef lcapa van de lokaaltabel (allemaal).
11. Geef lcapa van de lokaaltabel met lcapa tussen bvb. 10 en 100.
12. Geef lcapa van de lokaaltabel met lcapa tussen bvb. 5 en 400 (allemaal).
13. Geef alle velden van de lokaaltabel met lcapa gelijk aan een bepaalde waarde.
14. Geef de verschillende lcapa's met hun aantal, van de lokaaltabel (group by).
15. Geef de verschillende lcapa's van de lokaaltabel (distinct).
16. Geef de kleinste lcapa van de lokaaltabel.
17. Geef de grootste lnaam van de lokaaltabel.
18. Geef de lid van de lokaaltabel gesorteerd op lid.
19. Geef de lcapa van de lokaaltabel gesorteerd op lcapa.
20. Geef alle velden van de lokaaltabel en alle velden van de bijhorende practicatabel.
21. Geef alle velden van de lokaaltabel en de practicatabel gesorteerd op lid.
22. Geef de lid en paid van de lokaaltabel en de practicatabel.
23. Geef de lid en paid van de lokaaltabel en de practicatabel gesorteerd op lid.
24. Verklaar het verschil in resultaat tussen t02expl.sql en to2zexpl.sql
25. Maak een view die voor elke docentafkorting aangeeft hoeveel groepen oefpra hij/zij mag geven. Maak nu gebruik van deze view om per docentnaam het aantal groepen te geven, geordend op het aantal. Maak voor deze vraag dan een explain plan. Herhaal het voorgaande maar maak nu een materialized view. Verklaar het verschil in resultaat. Voor de ex-schakels: Het maken van een view gebeurt met

```
create or replace view 'naam van de view' as select 'wat je nodig hebt';
```

Het maken van een materialized view:

```
drop materialized view 'naam van view';
create materialized view 'naam van de view' as select 'wat je nodig hebt';
```

## B.10 sqldev

Opstarten van sqldeveloper met commando sqldev.
Migrate user settings: no.
Er wordt een map .sqldeveloper gemaakt in uw homedir en dat duurt even.
Tip of the day mag je sluiten.
Op connections klikken en vraag een new connection.
Geef een connection name in: bvb test ( mag origineler).
Als user name en password geef je uw e-nummer in: E400 en E400 .
Connection type,hostname en port staan goed. Alleen de sid aanpassen naar erp. Je kan nu op save drukken om deze info te bewaren en test om eens te testen. Bedoeling is te connecteren. Komt nog eens naar password vragen en geef dat dan maar terug.
Als je op het + teken van connections klikt verschijnt uw db met uw gekozen naam. Op + van test verschijnt een lijst met tabellen,views,indexen... Op + van tables geeft al uw tabellen weer. Nog eens op + op een table geeft de verschillende velden van uw tabel.
Naast het tabblad Connections heb je Reports. Klik daar eens op Bij Data Dictionary Reports en dan Table, Columns, Columns. In venster met titel "Enter Bind Values" kies je voor Table Name en als Value vul je bvb. basis in. Je krijgt dan een overzicht van al uw tabellen waar het woord basis in voorkomt. Om een User Defined Report te maken, rechtermuisknop en kiezen voor New Report. Geef het rapport een naam "Docentinfo" en voeg eventueel een omschrijving toe. Selecteer dan bij Style voor Chart en vul dan uw sql query in (rap1.sql). Na apply krijg je uw resultaat. Je kan nu ook met de rechtermuisknop op rap1 klikken en dan edit en dan bij property het chart type aanpassen. Maak nu zelf een rapport waarbij je in een taartdiagramma het aantal lokalen laat zien gegroepeerd op het al dan niet aanwezig zijn van een beamer en de grootte van het lokaal. We gaan nu terug naar het tabblad Connections.
Nu gaan we zonder een letter sql te typen een tabel maken.
Met rechtermuisknop op table en kies voor new table. We maken een tabel personeel met voorlopig 3 velden.
De eerste kolom intypen met volgende gegevens:

```
pnum number not null primary key
```

daarna op de add column drukken en volgende 2 invullen

```
pnaam vchar2 20 - -
pdat date - - -
```

Op het tweede tabblad DDL kan je zien hoe je vroeger de juiste sql instructie moest geven om dit resultaat te krijgen.
Na ok save die tabel en je ziet ze nu aan de linkerkant staan tussen uw andere tabellen.
In de rechterkant kan je nu het volgende typen

```
insert into personeel values (1,'Marie','12 feb 2000')
```

en op het groene driehoekje klikken voert het deze insert uit en bevestigen met icoon voor commit (cilinder met groen v-tje). Maar je kan een insert ook doen zonder sql te typen. Dubbelklik linkerkant op personeel en dan bovenaan het tweede tabblad Data. Dit laat zien wat er in uw tabel zit. Het inserticoon (blad met groene +) indrukken en dan kan je uw gegevens gewoon intypen. Als je op commit drukt krijg je uw inserts te zien. Voeg een paar rijen toe in uw tabel.

Bovenaan op de sort knop drukken en dan kies je maar de kolom waar je wilt op sorteren en dan apply sort.
We zijn een kolom vergeten. Moeilijke sql met alter table...
Nu druk op personeel en het eerste tabblad gaat over columns. Je ziet uw 3 kolommen staan. Kies bij actions voor columns en dan add.
Als naam salaris en type number.Bekijk ook eens de voor u gemaakte sql mbv het tweede tabblad. De blauwe pijltjes verversen uw scherm.
Als je op tabblad constraints drukt zie je uw beperkingen van een primary key en niet null staan. We gaan er nog eentje bijvoegen. Kies voor actions en dan constraints en dan add check.De naam van de constraint: minsalaris en voorwaarde salaris $>$ 500 en apply.
Nu terug bij Data en daar insert doen met een rij met salaris $<$ 500 en dan commit zal geweigerd worden. Voeg een paar geldige rijen toe.
Om de inhoud van een veld te wijzigen moest je vroeger een update ... typen. Ga in het betreffende veld staan en overtyp de oude waarde met de nieuwe.
Als je commit drukt zie je ook uw sql instructie staan. Wil je uw pnum wijzigen naar een ander reeds bestaand nummer zal dat niet toegelaten worden.
Bij de Grants-tab zie je dat je nog niemand toelating hebt gegeven om iets met uw tabellen te doen. Bij actions kies je voor privileges en grant en dan kies je bvb voor select en bij users kies je de nummer van uw buur. Sluit af met apply.
Als je nu in de linkerbalk naar other users gaat en daar de nummer van de buur zoekt die u de toelating heeft gegeven kan je zijn tabellen zien. Bij de andere users zie je niets.
Het maken van triggers wordt ook kinderspel. We gaan bvb een trigger maken voor het invoeren en aanpassen van het veld aant in de groepoefpra tabel. Als de gebruiker een aant ingeeft zodat de docent in totaal meer dan 10 groepen krijgt moet er een foutmelding gegeven worden. Bij dubbelklik op groepoefpra kies je voor actions en dan trigger en dan create. Er wordt een nieuw venster geopend. Vul een naam in voor uw trigger, klik dan insert en update aan. Zorg ervoor dat het een rowlevel trigger is. Laat in de selected columns aant staan. Druk dan op OK en dan zie je het reeds gemaakte ddl statement. Nu hoef je alleen nog voor begin een variabele te declareren van het aant type. Na begin dan de som van de aant maken per docent. Als het gesommeerde aantal dan $>$ 10 is ga je een error geven in de vorm van `raise_application_error(-20222, 'uw aantal groepen is te groot')`. Druk dan op het groene pijltje om uw trigger te compileren. Test uw trigger uit door een insert te proberen. Mogelijke input: 2050, 'HDA', 3. Ga terug naar uw personeel tabel.
Bovenaan op tabblad columns kies je voor actions en daar dan export. Er wordt een wizard gestart. Export DDL afvinken, export data aan en bij format kies je voor html. Voor filenaam browse je naar uw submap en OK. Kies next en bij columns op potloodje en we gaan de datum afvinken. Bij object where gaan we als voorwaarde vragen salaris is not null, druk dan op de go toets en dan zie je de rijen die aan uw voorwaarde voldoen. Daarna op OK. Nog eens op next en finish. Er is nu een file bijgemaakt. Bekijk die even en start dan een browser op en aanschouw het resultaat. In het search veld vul je maar een getal of stuk naam in en dan zal je de betreffende rijen zien.
We gaan nog eens data exporteren maar nu naar een xml formaat. We kiezen nu alle kolommen en in de where clause vragen we alle mensen met datum na februari. Tip: to_char(pdat,'MM') $>$ 2 . Bekijk even het bekomen resultaat.
Als je op het tabblad statistics klikt zie je uw aantal rijen,aantal bloks in uw databank in gebruik ,de gemiddelde rijlengte (20) en wanneer last_analyzed.
In het tweede venster de gemiddelde kolomlengte 3+7+8+2 = 20
Om de tabel te laten analizeren druk op actions en dan statistics en dan tweede item gather statistics en dan apply en dan refresh van scherm en je ziet dat de datum en uur op 'nu' staat.
Als je op tabblad flashback drukt kan je het beruchte rowid zien.
Als je op tabblad indexes drukt kan je de primary key zien.
Verras me nog aangenaam met eigen inbreng.