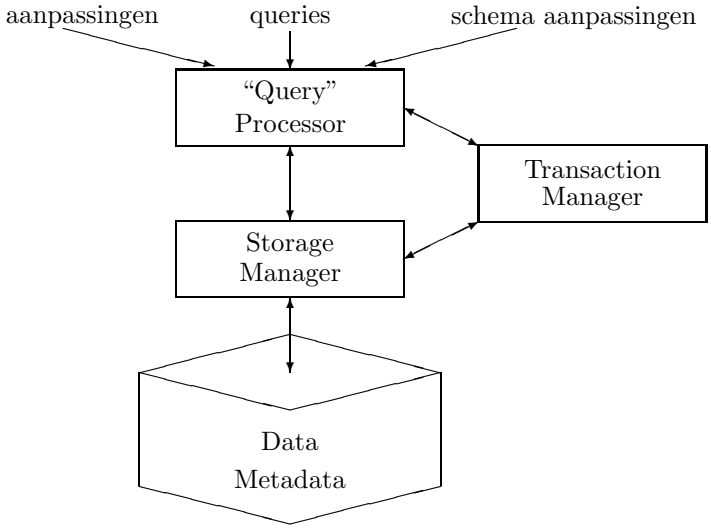
Campus De Nayer
Industrieel ingenieur
Opleiding Elektronica-ICT
Masterjaar

KULeuven
Academiejaar 2019-20

D A T A B A N K B E H E E R

1 Architectuur van een DBMS

In figuur 1.1 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 1.1: 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 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.

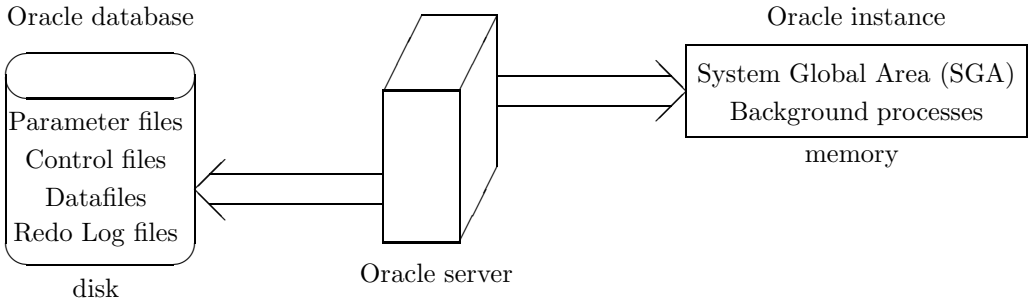
Aanpassingen : operaties om de gegevens te veranderen; eventueel zijn dit toevoegingen of worden er gegevens verwijderd. De manier waarop is zoals bij queries.

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.

2 Architectuur van Oracle.

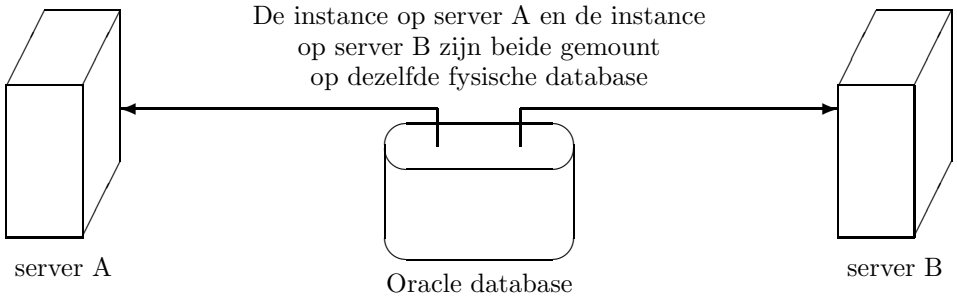
2.1 Database versus instance

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 2.1 toont de basiscomponenten van een Oracle database en instance.



Figuur 2.1: 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.

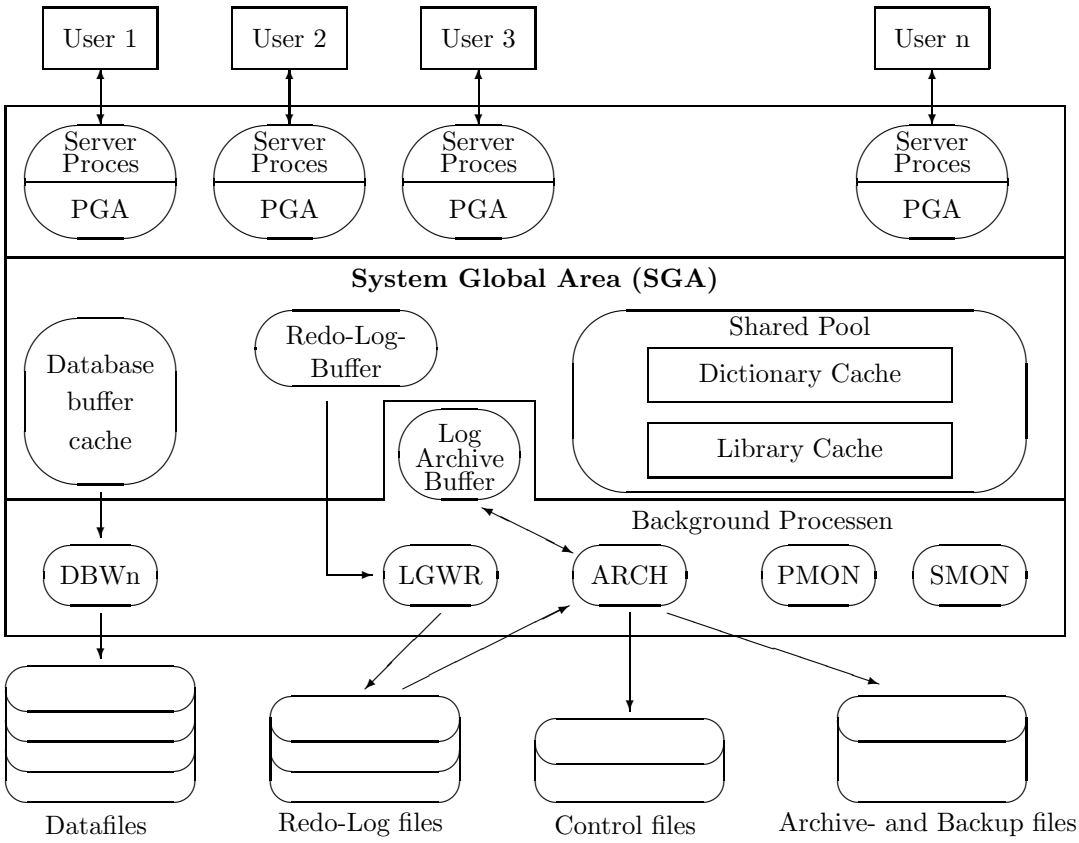


Figuur 2.2: Oracle Parallel Server

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 2.3 illustreert de architectuur van een Oracle DBMS bestaande uit geheugenstructuren, processen en bestanden.

2.2 Oracle instance

De Oracle instance bestaat uit de Oracle processen en de bijhorende geheugenstructuren (zie figuur 2.4). 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 2.3: Oracle systeem architectuur

2.2.1 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.

```
   UID   PID  PPID  C    STIME TTY       TIME COMMAND
oracle  2599     1  0  May 23  ?        1:38 ora_pmon_erp
oracle  2601     1  0  May 23  ?        9:34 ora_psp0_erp
oracle  2603     1  0  May 23  ?        4:42 ora_vktm_erp
oracle  2605     1  0  May 23  ?        0:57 ora_gen0_erp
oracle  2607     1  0  May 23  ?        0:38 ora_mman_erp
oracle  2611     1  0  May 23  ?        0:34 ora_diag_erp
oracle  2613     1  0  May 23  ?        3:59 ora_dbrm_erp
oracle  2615     1  0  May 23  ?        8:01 ora_vkrm_erp
oracle  2617     1  0  May 23  ?       27:06 ora_dia0_erp
oracle  2619     1  0  May 23  ?        2:13 ora_dbw0_erp
oracle  2621     1  0  May 23  ?        1:09 ora_lgwr_erp
oracle  2623     1  0  May 23  ?        6:57 ora_ckpt_erp
oracle  2625     1  0  May 23  ?        0:09 ora_lg00_erp
oracle  2627     1  0  May 23  ?        0:41 ora_smon_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 2.4: Een eenvoudige instance

```
oracle  2629     1  0  May 23  ?         0:07 ora_lg01_erp
oracle  2632     1  0  May 23  ?         0:07 ora_reco_erp
oracle  2634     1  0  May 23  ?         0:34 ora_lreg_erp
oracle  2636     1  0  May 23  ?         0:30 ora_pxmn_erp
oracle  2638     1  0  May 23  ?        14:03 ora_mmon_erp
oracle  2640     1  0  May 23  ?        20:54 ora_mmnl_erp
oracle  2642     1  0  May 23  ?         0:12 ora_d000_erp
oracle  2644     1  0  May 23  ?         0:13 ora_s000_erp
oracle  2656     1  0  May 23  ?         0:14 ora_tmon_erp
oracle  2658     1  0  May 23  ?         0:36 ora_tt00_erp
oracle  2660     1  0  May 23  ?         3:36 ora_smco_erp
oracle  2668     1  0  May 23  ?         0:09 ora_aqpc_erp
oracle  2672     1  0  May 23  ?         8:48 ora_cjq0_erp
oracle  2674     1  0  May 23  ?         0:16 ora_p000_erp
oracle  2775     1  0  May 23  ?         0:15 ora_qm02_erp
oracle  7325     1  0 10:03:27 ?         0:00 ora_w002_erp
oracle  3257     1  0  May 23  ?         0:32 /u01/app/oracle/product/12.1.0/bin/tnslsnr LISTEN
```

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.

2.2.2 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.

2.3 Oracle database

2.3.1 Parameters

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

```
# Copyright (c) 1991, 2013 by Oracle Corporation
# Cache and I/O
db_block_size=8192
# Cursors and Library Cache
open_cursors=300
# Database Identification
db_domain=local.thomasmore.be
db_name="erp"
# File Configuration
control_files=("/u03/oradata/erp/control01.ctl", "/u03/oradata/erp/control02.ctl")
# Miscellaneous
compatible=12.1.0.2.0
diagnostic_dest=/u01/app/oracle
# NLS
nls_language="ENGLISH"
nls_territory="BELGIUM"
# Processes and Sessions
processes=300
sessions=225
# SGA Memory
sga_target=9780m
# Security and Auditing
audit_file_dest="/u01/app/oracle/admin/erp/adump"
audit_trail=db
remote_login_passwordfile=NONE
# Shared Server
dispatchers="(PROTOCOL=TCP) (SERVICE=erpXDB)"
# Sort, Hash Joins, Bitmap Indexes
pga_aggregate_target=3260m
# System Managed Undo and Rollback Segments
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.

2.3.2 Script om een database te creëren

In Oracle 12c worden een aantal scripts gegenereerd om de database en aanverwanten te maken:
CloneRmanRestore.sql

```
SET VERIFY OFF
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool /u01/app/oracle/admin/erp/scripts/CloneRmanRestore.log append
startup mount pfile="/u01/app/oracle/admin/erp/scripts/initerpTempOMF.ora";
execute dbms_backup_restore.resetCfileSection(dbms_backup_restore.RTYP_DFILE_COPY);
execute dbms_backup_restore.resetCfileSection(13);
host /u01/app/oracle/product/12.1.0/bin/rman @/u01/app/oracle/admin/erp/scripts/rmanRestoreDataf
column file0 NEW_VALUE file0;
select NAME file0 FROM V$DATAFILE_COPY where file# = 3;
column file1 NEW_VALUE file1;
select NAME file1 FROM V$DATAFILE_COPY where file# = 1;
column file2 NEW_VALUE file2;
select NAME file2 FROM V$DATAFILE_COPY where file# = 4;
column file3 NEW_VALUE file3;
select NAME file3 FROM V$DATAFILE_COPY where file# = 6;
spool off
```

cloneDBCreation.sql

```
SET VERIFY OFF
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool /u01/app/oracle/admin/erp/scripts/cloneDBCreation.log append
shutdown abort;
startup nomount pfile="/u01/app/oracle/admin/erp/scripts/init.ora";
Create controlfile reuse set database "erp"
MAXINSTANCES 1
MAXLOGHISTORY 1
MAXLOGFILES 16
MAXLOGMEMBERS 3
MAXDATAFILES 100
Datafile
'&&file0',
'&&file1',
'&&file2',
'&&file3'
LOGFILE GROUP 1 ('/u03/oradata/erp/redo01.log') SIZE 50M,
GROUP 2 ('/u03/oradata/erp/redo02.log') SIZE 50M,
```

```
GROUP 3 ('/u03/oradata/erp/redo03.log') SIZE 50M RESETLOGS;
exec dbms_backup_restore.zerodbid(0);
shutdown immediate;
startup nomount pfile="/u01/app/oracle/admin/erp/scripts/initerpTemp.ora";
Create controlfile reuse set database "erp"
MAXINSTANCES 1
MAXLOGHISTORY 1
MAXLOGFILES 16
MAXLOGMEMBERS 3
MAXDATAFILES 100
Datafile
'&&file0',
'&&file1',
'&&file2',
'&&file3'
LOGFILE GROUP 1 ('/u03/oradata/erp/redo01.log') SIZE 50M,
GROUP 2 ('/u03/oradata/erp/redo02.log') SIZE 50M,
GROUP 3 ('/u03/oradata/erp/redo03.log') SIZE 50M RESETLOGS;
alter system enable restricted session;
alter database "erp" open resetlogs;
exec dbms_service.delete_service('seeddata');
exec dbms_service.delete_service('seeddataXDB');
alter database rename global_name to "erp.local.thomasmore.be";
ALTER TABLESPACE TEMP ADD TEMPFILE '/u03/oradata/erp/temp01.dbf' SIZE 61440K
         REUSE AUTOEXTEND ON NEXT 640K MAXSIZE UNLIMITED;
select tablespace_name from dba_tablespaces where tablespace_name='USERS';
alter user sys account unlock identified by "&&sysPassword";
connect "SYS"/"&&sysPassword" as SYSDBA
alter user system account unlock identified by "&&systemPassword";
select sid, program, serial#, username from v$session;
alter database character set INTERNAL_CONVERT WE8ISO8859P15;
alter database national character set INTERNAL_CONVERT AL16UTF16;
alter system disable restricted session;
```

postScripts.sql

```
SET VERIFY OFF
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool /u01/app/oracle/admin/erp/scripts/postScripts.log append
UPDATE sys.USER$ set SPARE6=NULL;
@/u01/app/oracle/product/12.1.0/rdbms/admin/dbmssml.sql;
execute dbms_datapump_utl.replace_default_dir;
commit;
connect "SYS"/"&&sysPassword" as SYSDBA
alter session set current_schema=ORDSYS;
@/u01/app/oracle/product/12.1.0/ord/im/admin/ordlib.sql;
alter session set current_schema=SYS;
connect "SYS"/"&&sysPassword" as SYSDBA
create or replace directory XMLDIR as '/u01/app/oracle/product/12.1.0/rdbms/xml';
create or replace directory XSDDIR as '/u01/app/oracle/product/12.1.0/rdbms/xml/schema';
connect "SYS"/"&&sysPassword" as SYSDBA
connect "SYS"/"&&sysPassword" as SYSDBA
execute ORACLE_OCM.MGMT_CONFIG_UTL.create_replace_dir_obj;
execute dbms_qopatch.replace_logscrpt_dirs;
```

```
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool /u01/app/oracle/admin/erp/scripts/postDBCreation.log append
grant sysdg to sysdg;
grant sysbackup to sysbackup;
grant syskm to syskm;
```

lockAccount.sql

```
SET VERIFY OFF
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
spool /u01/app/oracle/admin/erp/scripts/lockAccount.log append
BEGIN
 FOR item IN ( SELECT USERNAME FROM DBA_USERS WHERE ACCOUNT_STATUS IN
     ('OPEN', 'LOCKED', 'EXPIRED') AND USERNAME NOT IN ( 'SYS','SYSTEM') )
 LOOP
  dbms_output.put_line('Locking and Expiring: ' || item.USERNAME);
  execute immediate 'alter user ' ||
   sys.dbms_assert.enquote_name(
   sys.dbms_assert.schema_name(
   item.USERNAME),false) || ' password expire account lock' ;
 END LOOP;
END;
/
spool off
```

postDBCreation.sql

```
SET VERIFY OFF
spool /u01/app/oracle/admin/erp/scripts/postDBCreation.log append
@/u01/app/oracle/product/12.1.0/rdbms/admin/catbundleapply.sql;
connect "SYS"/"&&sysPassword" as SYSDBA
set echo on
create spfile='/u01/app/oracle/product/12.1.0/dbs/spfileerp.ora'
       FROM pfile='/u01/app/oracle/admin/erp/scripts/init.ora';
connect "SYS"/"&&sysPassword" as SYSDBA
select 'utlrp_begin: ' || to_char(sysdate, 'HH:MI:SS') from dual;
@/u01/app/oracle/product/12.1.0/rdbms/admin/utlrp.sql;
select 'utlrp_end: ' || to_char(sysdate, 'HH:MI:SS') from dual;
select comp_id, status from dba_registry;
execute dbms_swrf_internal.cleanup_database(cleanup_local => FALSE);
commit;
shutdown immediate;
connect "SYS"/"&&sysPassword" as SYSDBA
startup ;
spool off
exit;
```

2.4 Gebruikersaccounts

2.4.1 Gebruikerscreatie

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/erp/&&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/erp/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.

2.4.2 Beveiliging

Er zijn zes types van privileges. De eerste vier, `SELECT`, `INSERT`, `DELETE` en `UPDATE`, hebben te maken met een relatie (zowel een basistabel als een view). Degene die het privilege gekregen heeft, kan de desbetreffende actie uitvoeren op de tabel.

Het `REFERENCE` privilege is het recht om naar de tabel te refereren in een integriteitsconstraint, bijv. assertions, attribuut- en tuple-gebaseerde CHECKs en referentiële integriteitsbeperkingen. Het `USAGE` privilege op een domein of een ander schema-element (met uitzondering van relaties en assertions) geeft het recht dat element in zijn eigen declaraties te gebruiken.
Normaal worden privileges toegekend op een ganse tabel. In SQL2 is het mogelijk INSERT, DELETE en UPDATE privileges toe te kennen op één of meerdere attributen van een relatie. Dit kan ook gerealiseerd worden door een VIEW te creëren en dan privileges op het niveau van de VIEW te definiëren.
De gebruiker die een schema-element creëert, wordt eigenaar van dat element en heeft er alle privileges op. Met behulp van `GRANT` en `REVOKE` statements kunnen privileges toegekend worden aan of afgenomen worden van andere gebruikers.

1. GRANT gevolgd door één of meer privileges of ALL PRIVILEGES;
2. ON gevolgd door een tabel of een view;
3. TO gevolgd door een lijst van gebruikers of PUBLIC;
4. eventueel, `WITH GRANT OPTION`, wat aangeeft dat de gebruiker die het privilege krijgt, dit privilege zelf kan doorgeven aan andere gebruikers.

Voorbeeld:

```
GRANT SELECT                 ON L   TO bae, kvn, top;
GRANT SELECT, INSERT         ON L,A TO svd, lro   WITH GRANT OPTION;
GRANT SELECT, DELETE, UPDATE ON LA  TO PUBLIC;
```

1. REVOKE gevolgd door één of meer privileges of ALL PRIVILEGES;
2. ON gevolgd door een tabel of een view;
3. FROM gevolgd door een lijst van gebruikers of PUBLIC.

Voorbeeld:

```
REVOKE SELECT          ON L   FROM bae, kvn;
REVOKE SELECT, INSERT  ON A   FROM lro;
REVOKE DELETE, UPDATE  ON LA  FROM fdf;
```

2.4.3 Gebruikersrechten

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;
```

2.5 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 views 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 vastgelegd; **ALL**-views geven informatie over objecten die niet het eigendom zijn van een gebruiker. De weergegeven objecten zijn objecten die 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.

2.6 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.

3 Het fysisch niveau: data

3.1 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.

3.1.1 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 3.1.)

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 3.1: 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 3.2 toont het bestand van figuur 3.1 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 3.2: 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*.

3.1.2 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 3.3.) 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	$\perp$
1	1eo	Tiri	2	$\perp$				
2	1cc	Sanders	1	$\perp$				
3	1bl	Jacobs	3	Peeters	1	$\perp$		
4	1bb	Lameire	1	$\perp$				
5	1cb	Goossens	1	$\perp$				

Figuur 3.3: 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 3.4: *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 3.4.) 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 3.5.)

3.1.3 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

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

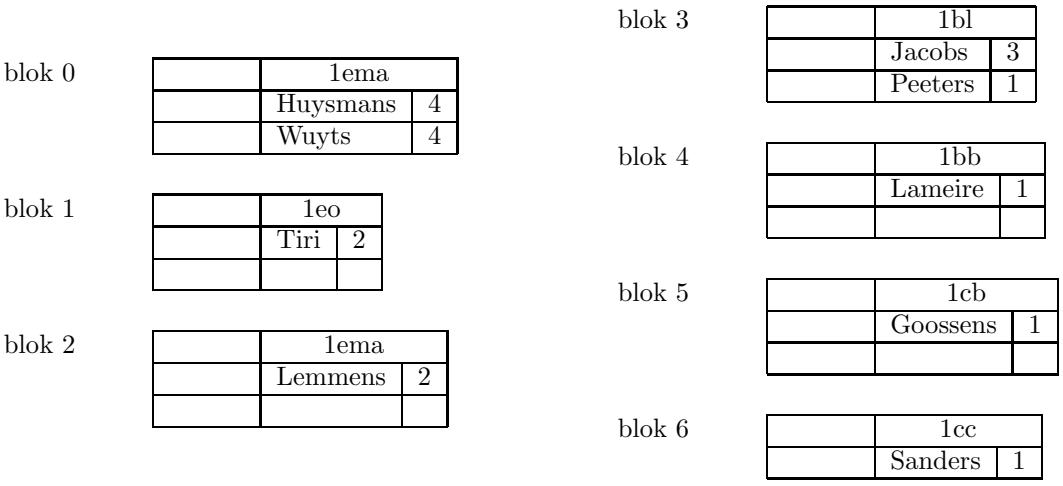
0	(2)	Wuyts	4
1		Peeters	1
2		Lemmens	2

Figuur 3.5: Anchor blok en overflow blok organisatie

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 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 aan 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.
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



Figuur 3.6: Bucket bestandsorganisatie

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.

3.2 Oracle database

3.2.1 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*, en nog enkele andere bestanden. Ook het aantal gebruikers dat tegelijk kan connecteren naar de databank is opgenomen.

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.

3.2.2 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.

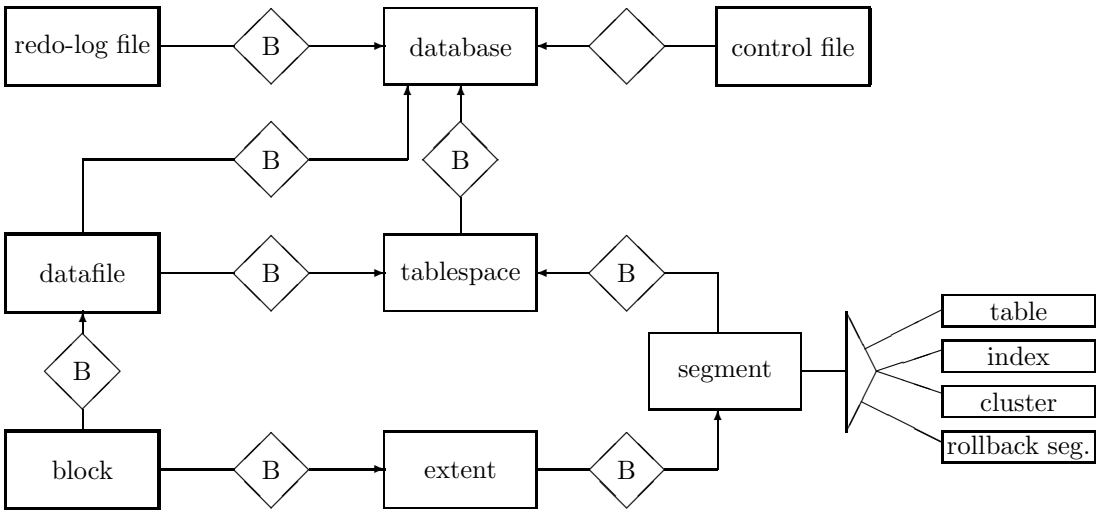
Rollback segment : Telkens een update in een tabel gebeurt, wordt de oude waarde weggeschreven in een rollback segment, wat de andere gebruikers toelaat om een consistente read op de tabel te blijven doen. Het geeft Oracle ook de mogelijkheid om de inhoud van de tabel te herstellen wanneer de verandering 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 3.7 geeft het verband tussen de logische en de fysische database structuren.



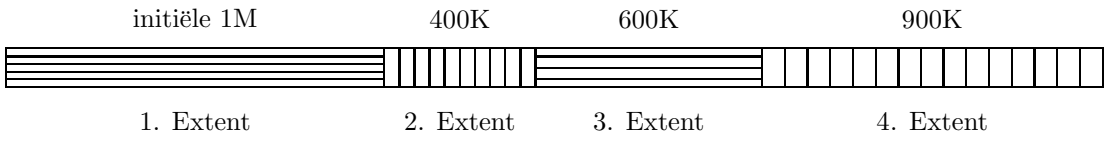
Figuur 3.7: ER-diagram

3.2.3 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:

```
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 3.8).



Figuur 3.8: 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).

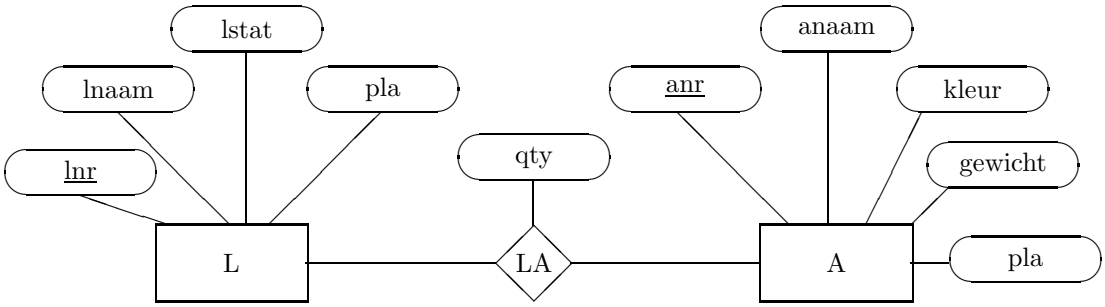
```
select rowid, bafk from basis;

ROWID              BAFK
------------------ --------
AAAW7dAAHAAAADbAAA ab1
AAAW7dAAHAAAADbAAB ab2
AAAW7dAAHAAAADbAAE abc2
AAAW7dAAHAAAADbAAF abc3
AAAW7dAAHAAAADbAAC abei3
AAAW7dAAHAAAADbAAD abem3
AAAW7dAAHAAAADbAAG mei4
```

Blocknumber `AAAADb` volgens base-64:				
	D	(3)	000011	192
	b	(27)	011011	27
				219

4 PL/SQL

4.1 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

4.2 Beschrijving

PL/SQL is een procedurele uitbreiding op SQL, omdat SQL op zich niet krachtig genoeg is om complexe databank applicaties te ontwikkelen. Server-side functies, of *stored procedures* worden op de database server uitgevoerd in plaats van in de client-applicatie. Er zijn verschillende redenen om server-side functies te gebruiken:

- verhogen van de uitdrukkingskracht van SQL;
- query resultaten tuple per tuple verwerken;
- optimisatie van gecombineerde SQL statements;
- modulaire programma's;
- hergebruik van programma code; een functie die door verschillende applicaties gebruikt wordt, moet niet in elk van deze applicaties gecopieerd worden; telkens een functie nodig is, zal de client de functie oproepen;
- kostreductie bij onderhoud en aanpasingen van applicaties. De functies zijn centraal op de databaseserver gestockeerd en bij aanpassingen zullen alle client-applicaties onmiddellijk de nieuwe versie gebruiken.

Triggers zijn speciale server-side functies, die automatisch opgeroepen worden telkens een tabel aangepast wordt.

4.3 Structuur

PL/SQL is een blok-georiënteerde taal:

```
[ <Blok hoofding> ]
[ DECLARE
        <Constanten>
        <Variabelen>
        <Cursors>
        <Eigen gedefinieerde exceptions> ]
  BEGIN
        <PL/SQL statements>
        [ EXCEPTION
              <afhandeling exception > ]
  END;
```

PL/SQL statement : { SQL statement (alleen DML) | controle structuur (lus of selectie) | afhandelen van excepties | oproepen van een PL/SQL blok

Een PL/SQL statement kan zelf een PL/SQL blok zijn. Indien geen < Blok hoofding> gebruikt wordt, spreekt men van een *anoniem blok*. Een <Blok hoofding> wordt gebruikt voor de definitie van een *function* of een *procedure*.

4.4 Declaraties van variabelen

<Variabele naam> [**constant**] <data **type**> [**not null**] [:= <expressie>]

constant : eens een waarde is toegekend aan de variabele, kan deze waarde niet meer gewijzigd worden.

not null : de gedeclareerde variabele moet steeds een waarde verschillend van `null` hebben.

- Klassieke types: **number**, **integer**, **char**(n), **date**, **boolean**.
- Verwijzingen:

  Tabelnaam.kolomnaam%TYPE : het type van de variabele is gelijk aan het data type van de gerefereerde kolom van de gespecificeerde tabel.

  Tabelnaam%ROWTYPE : dit data type specificeert dat de record-variabele alle attribuutwaarden van een volledige rij uit de gespecificeerde tabel kan bevatten.

  Een veld in deze record aanspreken gebeurt met de dot-notatie: `recordnaam.kolomnaam`.

- Cursors:

  CURSOR <cursornaam> [(<parameterlijst>)] **IS** <**select** statement>]

4.5 Toekenningen

Het resultaat van een rekenkundige expressie aan een variabele toewijzen:

```
DECLARE
      teller  integer;
BEGIN
      teller  := 0;
      teller  :=  teller  + 1;
```

```
        dbms_output.put_line( teller );
    END;
```

Om een PL/SQL blok uit te voeren, wordt het blok in SQL*Plus ingetikt en deze input wordt afgesloten met een slash (/) op een aparte lijn. Deze slash geeft aan SQL*Plus aan dat het blok volledig ingetikt is en dat het naar de database server moet verzonden worden om daar uitgevoerd te worden. (Deze slash wordt in de PL/SQL voorbeelden in deze tekst niet getoond.)

Package voor het sturen van output:

- DBMS_OUTPUT.PUT(<string>) : de string wordt aan de output buffer toegevoegd;
- DBMS_OUTPUT.PUT_LINE(<string>) : de string en een newline worden aan de output buffer toegevoegd;
- DBMS_OUTPUT.NEW_LINE : een newline wordt aan de output buffer toegevoegd.

Code in een PL/SQL blok kan output genereren, maar deze wordt by default niet getoond door SQL*Plus. De reden hiervoor is dat SQL*Plus niet zelf de PL/SQL code uitvoert. Wanneer de slash (/) ingetikt wordt, zendt SQL*Plus de code naar de database server, die de code uitvoert. Maar de Oracle database server heeft niet de mogelijkheid om de eventuele output rechtstreeks aan de gebruiker te tonen. In plaats daarvan wordt de eventuele output van een PL/SQL blok gebufferd door de server totdat de applicatie die de uitvoering van het blok aanvroeg, deze informatie opvraagt. By default vraagt SQL*Plus de PL/SQL output van de server niet op. Dit kan wel gewijzigd worden door de **serveroutput** optie te zetten:

```
SQL> SET SERVEROUTPUT ON
```

Het resultaat van een SELECT statement aan een variabele toewijzen:

```
DECLARE
    product a.anaam%TYPE;
    alles   a%ROWTYPE;
BEGIN
    SELECT anaam INTO product FROM a WHERE anr = 'A1' ;
    dbms_output.put_line(product);
    SELECT * INTO alles FROM a WHERE anr = 'A1' ;
    dbms_output.put_line('naam ' || alles .anaam
            || ' met kleur ' || alles .kleur );
END;
```

Het resultaat van de SELECT mag slechts één waarde of één rij zijn. Indien meerdere rijen uit de tabel aan de WHERE voldoen, wordt een *exception* gegenereerd.

4.6 Controle structuren

4.6.1 IF statement

```
IF <conditie> THEN
    <PL/SQL statement>
    <PL/SQL statement>
    ...
ELSIF <conditie> THEN
    <PL/SQL statement>
    <PL/SQL statement>
    ...
ELSE
    <PL/SQL statement>
    <PL/SQL statement>
```

```
    ...
END IF;
```

```
declare
    kleurtje   a.kleur%type;
    som        integer := 0;
begin
    select kleur into kleurtje from a where anr='A1';
    if kleurtje = 'rood' then
        som := som + 1;
    elsif kleurtje = 'geel' then
        som := som + 2;
    else
        som := som + 3;
    end if;
    dbms_output.put_line(kleurtje || ' : ' || som);
end;
```

4.6.2 WHILE lus

```
[ << <label naam> >> ]
WHILE <conditie> LOOP
    <PL/SQL statement>
    <PL/SQL statement>
    ...
END LOOP [ <label naam> ] ;
```

```
declare
    teller integer;
begin
    teller := 0;
    while teller < 10 loop
        teller := teller + 1;
        dbms_output.put_line(teller );
    end loop;
end;
```

4.6.3 Continue lus

```
[ << <label naam> >> ]
LOOP
    <PL/SQL statement>
    EXIT WHEN <conditie>
    <PL/SQL statement>
    ...
END LOOP [ <label naam> ] ;
```

```
declare
    teller integer;
    som    integer := 0;
begin
    teller := 0;
```

```
    loop
        teller := teller + 1;
        exit when teller >= 10;
        som := som + teller;
        dbms_output.put_line(teller || ' : ' || som );
    end loop;
    dbms_output.put_line('het totaal : ' || som
                || ' met telller : ' || teller );
end;
```

4.6.4 FOR lus

```
[ << <label naam> >> ]
  FOR <index> IN [ REVERSE ] <ondergrens> .. <bovengrens> LOOP
      <PL/SQL statement>
      <PL/SQL statement>
      ...
  END LOOP [ <label naam> ] ;
```

```
declare
    teller integer;
    som integer := 0;
begin
    for teller in 0..9 loop
        som := som + teller;
        dbms_output.put_line(teller || ' : ' || som );
    end loop;
end;
```

4.7 Triggers

Beperkingen in het SQL create statement worden gecontroleerd op het moment dat het betrokken element gewijzigd wordt. Het DBMS zorgt daarvoor.
In de voorgestelde SQL3 standaard worden ook *triggers* gedefinieerd. De keuze om al of niet te triggeren wordt hier overgelaten aan de database programmeur. Op die manier worden aan de gebruiker een aantal opties aangeboden om database operaties te trigggeren en dit niet alleen om schendingen van beperkingen te vermijden.
Verschillen tussen triggers (of *event-condition-action* regels) en beperkingen:

- Triggers worden alleen getest wanneer bepaalde *events*, die door de database programmeur bepaald zijn, zich voordoen. De toegelaten events zijn meestal toevoegen, verwijderen of aanpassen aan/uit een bepaalde relatie. Een andere event is het einde van een transactie (zie volgend hoofdstuk).
- Bij het optreden van een event wordt een *conditie* getest. Indien de voorwaarde niet geldt, wordt verder niets dat met de trigger te maken heeft, uitgevoerd bij deze event.
- Indien aan de voorwaarde van de trigger voldaan is, wordt de *actie* die bij de trigger hoort, door het DBMS uitgevoerd. Deze actie kan bijvoorbeeld voorkomen dat de event plaats vindt, of kan de event ongedaan maken (bijv. een rij verwijderen die net toegevoegd is). De actie kan om het even welke rij van database operaties zijn, zelfs operaties die niets met het triggering event te maken hebben.

Belangrijkste elementen in een trigger:

1. De actie kan uitgevoerd worden *voor*, *na* of *in plaats van* de triggering event.

2. De actie kan zowel naar de oude als de nieuwe waarden van de tuples verwijzen die toegevoegd, verwijderd of aangepast zijn tijdens de event dat de actie triggerde.
3. Met de WHEN kan een conditie gespecificeerd worden: de actie wordt alleen uitgevoerd bij een triggering *en* er moet voldaan zijn aan de voorwaarde op het moment van de event.
4. Er kan gespecificeerd worden of de actie moet uitgevoerd worden
   - ofwel eenmaal voor elke aangepaste tuple;
   - ofwel voor alle tuples die tijdens één database operatie gewijzigd zijn.

Syntax in Oracle voor de creatie van een trigger

CREATE [**OR REPLACE**] **TRIGGER** <**trigger** naam>
BEFORE | **AFTER**
INSERT OR UPDATE [**OF** <kolomnamen>] **OR DELETE ON** <tabelnaam>
[**FOR EACH ROW**]
[**WHEN** (<voorwaarde>)]
[**DECLARE**
<declaraties>]
BEGIN
<PL/SQL statements>
END;

De actie wordt met behulp van PL/SQL statements geschreven. Dit zijn gewone SQL statements en daarnaarst zijn ook enkele controle statements mogelijk zoals in een klassieke procedurele taal.
Voorbeeld 1:

```
create or replace trigger lichter
before update of gewicht on artikels
for each row
when ( old.gewicht > new.gewicht )
begin
     dbms_output.put_line('trigger met '
               ||to_char(:old.gewicht));
     :new.gewicht := :old.gewicht;
end;
```

	toud			tnieuw	
A1	26	Peulis		25	**26**
A2	31	Hever			
A3	17	Grootlo			
A4	24	Peulis		25	*25*
A5	12	Hever			
A6	29	Peulis		25	**29**

Bij elke update van het attribuut *gewicht* (event) wordt voor de update (BEFORE) een conditie getest (WHEN). Indien deze conditie waar is, wordt de aktie uitgevoerd (een SQL statement) en dit voor iedere rij, waarbij het oude en nieuwe tuple kan aangesproken worden.

Bijvoorbeeld, na het uitvoeren van

```
update A
set gewicht = 25 where pla = 'Peulis'
```

is alleen het gewicht van artikel **A4** aangepast.

Andere mogelijkheden:

1. BEFORE kan ook AFTER of INSTEAD OF zijn;
2. Bij een BEFORE trigger kan je de NEW waarden wijzigen en de OLD niet.
   Bij een AFTER trigger kan je zowel de NEW als de OLD waarden niet wijzigen.
3. Bij UPDATE is er sprake van een OLD en een NEW tuple, die door middel van REFERENCING een naam krijgen.
   Bij INSERT is alleen sprake van een NEW tuple.
   Bij DELETE kan men alleen naar het OLD tuple verwijzen.
4. De aktie kan uit meerdere SQL statements bestaan, van elkaar gescheiden door een `';'`.

Met behulp van `FOR EACH ROW` wordt een *row-level trigger* aangegeven. Bij een update van een volledige tabel met behulp van een SQL statement, wordt de trigger voor elke aangepaste rij

uitgevoerd. Bij een *statement-level trigger* wordt de trigger maar eenmaal uitgevoerd. In dit geval kan niet naar het OLD en NEW tuple verwezen worden omdat het hier telkens over een set van tuples gaat die door het statement aangepast worden. Met behulp van `OLD_TABLE` en `NEW_TABLE` kunnen deze sets benoemd worden.
Bij een row level trigger wordt de trigger uitgevoerd bij elke gerelateerde rij.
Voorbeeld 2:

```
CREATE OR REPLACE TRIGGER invoerartikel
BEFORE INSERT OR UPDATE ON a
FOR EACH ROW
BEGIN
    IF :NEW.anaam IS NULL THEN
        raise_application_error ( −20020, 'anaam kan niet NULL zijn');
    END IF;
    IF :NEW.gewicht IS NULL THEN
        raise_application_error ( −20021,
            :NEW.anaam || ' kan geen NULL gewicht hebben');
    END IF;
    IF :NEW.gewicht < 0 THEN
        raise_application_error ( −20022,
            :NEW.anaam || ' kan geen negatief gewicht hebben');
    END IF;
END;
```

:NEW bevat de nieuwe waarden van de rij die toegevoegd wordt of aangepast wordt. In deze trigger wordt dus nagegaan of aan een aantal beperkingen omtrent de waarden van een aantal attributen voldaan is. Indien dit niet zo is, wordt met behulp van raise_application_error een foutboodschap uitgeschreven en wordt de actie afgebroken. Er zal dus geen toevoeging of aanpassing gebeuren.
Gebruik:

```
INSERT INTO a VALUES ('A1','schroef','rood',26,'Peulis');
INSERT INTO a VALUES ('A1','schroef','rood',−4,'Peulis');
```

Voorbeeld 3:

```
CREATE OR REPLACE TRIGGER klantnaam
BEFORE UPDATE ON k
FOR EACH ROW
BEGIN
    IF :NEW.knaam <> :OLD.knaam THEN
        raise_application_error ( −20026, :OLD.knaam ||
            ' kan niet wijzigen in ' || :NEW.knaam);
    END IF;
END;
```

:OLD bevat de oude waarden van de rij die aangepast wordt. Indien aan de voorwaarden voldaan is, wordt de nieuw rij in de plaats gezet van de oude rij.
Gebruik: **UPDATE** k **SET** knaam = 'Lieze' **WHERE** knr = 'K3';

De statement level trigger wordt eenmaal voor alle tuples die tijdens één database operatie gewijzigd zijn, uitgevoerd.
Definitie van een *logging tabel* voor acties op tabel **a**:

```
CREATE TABLE logtabel ( logtekst char(10),
                        loguser   char(20),
                        logtijd   date       );
```

Definitie van de logging trigger:

```
CREATE OR REPLACE TRIGGER alogging
AFTER INSERT OR UPDATE OR DELETE ON a
BEGIN
    IF INSERTING THEN
        INSERT INTO logtabel VALUES ('INSERT', user, sysdate) ;
    END IF;
    IF UPDATING THEN
        INSERT INTO logtabel VALUES ('UPDATE', user, sysdate) ;
    END IF;
    IF DELETING THEN
        INSERT INTO logtabel VALUES ('DELETE', user, sysdate) ;
    END IF;
END;
```

Gebruik: **UPDATE** a **SET** gewicht = 25 **WHERE** pla = 'Peulis';

4.8 Bind variabelen

SQL*Plus voorziet in twee soorten variabelen:

- substitutie variabelen: gebruik bij SQL*Plus scripts;
- bind variabelen: voor de ondersteuning van het gebruik van PL/SQL in SQL*Plus scripts. Bind variabelen worden gebruikt voor het teruggeven van waarden uit een PL/SQL blok naar SQL*Plus, waar deze data kan gebruikt worden in volgende queries of in andere PL/SQL blokken.

Declaratie: VARIABLE <var_naam> <data_type>

VARIABLE is een SQL*Plus command dat kan afgekort worden tot VAR.

<**data_type**> : **number**, **char**(n), **nchar**(n), **varchar2**(n), **nvarchar2**(n).

Het command VAR kan ook gebruikt worden om een lijst te tonen van alle reeds gedeclareerde variabelen of, indien de naam van een bestaande variabele vermeld wordt, de informatie omtrent deze variabele te tonen.
Het *bereik* van een bind variabele is de SQL*Plus sessie waarin de variabele gedeclareerd geweest is. Variabelen die in een PL/SQL blok gedeclareerd worden daarentegen, houden op te bestaan wanneer het blok uitgevoerd is. Bind variabelen zitten dus op een niveau hoger, en kunnen dus door meerdere PL/SQL blokken en queries gebruikt worden.
Om een bind variabele in een PL/SQL blok te gebruiken, moet de naam voorafgegaan worden door een dubbelpunt (:).

```
SQL> VAR artikel CHAR(5)
SQL> BEGIN
        :artikel := 'A1';
     END;
     /
SQL> SELECT * FROM la WHERE anr = :artikel;
```

Het SQL*Plus command EXECUTE (afgekort EXEC) kan gebruikt worden om een enkelvoudig PL/SQL statement uit te voeren:

```
SQL> EXEC :artikel := 'A1'
```

De inhoud van een bind variabele kan op twee manieren getoond worden: met behulp van het PRINT command of door middel van het SELECT statement.

Formaat van het PRINT (afgekort tot PRI) command: PRINT <var_naam>

Wanneer geen variabele gespecificeerd wordt, wordt de inhoud van alle bind variabelen getoond.

Het SELECT statement kan ook gebruikt worden om de inhoud van een variabele te tonen:

```
SQL> VAR artikel CHAR(5)
SQL> EXEC :artikel := 'A2'
SQL> SELECT :artikel FROM dual;
```

In bovenstaand voorbeeld geeft SELECT geen functioneel voordeel ten opzichte van het PRINT command. Gebruik van SELECT wordt wel interessant wanneer informatie uit meer dan één kolom moet getoond worden:

```
SQL> SELECT :artikel || 'geleverd door ' || to_char(count(*))
     FROM la WHERE anr = :artikel;
```

4.9 Functies

Het **declare** gedeelte wordt vervangen door <Blok hoofding> om een *function* of een *procedure* te definiëren.

CREATE [**OR REPLACE**] { **FUNCTION** | **PROCEDURE** } <naam>
[(<parameterlijst>)] [**RETURN** <**type**>] **IS**
[<declaraties>]
BEGIN
<PL/SQL statements>
END;

Een <parameter> in een <parameterlijst>:

<naam> [**IN** | **OUT** | **IN OUT**] <datatype> [{ := | **DEFAULT** } <expressie>]

4.9.1 Een functie zonder argumenten

```
create or replace function een return integer is
begin
        return 1;
end;
```

Gebruik: **SELECT** een **AS** antwoord **FROM** dual;

Ook mogelijk: EXEC dbms_output.put_line(een)

4.9.2 Een functie met eenvoudige argumenten

```
create or replace function som(een in integer,twee in integer) return integer is
begin
        return een+twee;
end;
```

Gebruik: **SELECT** som(1,2) **AS** antwoord **FROM** dual;

4.9.3 Een functie met string argumenten

```
create or replace function concat(een in varchar2,twee in varchar2) return varchar2 is
begin
        return een || twee;
end;
```

Gebruik: **SELECT** concat('appel','tje') **AS** kleintje **FROM** dual;

4.9.4 Een functie met samengestelde argumenten

create or replace function dubbel(x **in** a%**ROWTYPE**) **return integer is**
begin
    return 2 * x.gewicht;
end;

Het is niet mogelijk om de functie op de volgende manier te gebruiken:

SELECT anr, dubbel(?) **AS** droom **FROM** a **WHERE** a.pla = 'Peulis';

4.9.5 Probleem bij een functie

create or replace function ruim_art **return integer is**
begin
    delete from a **where** a.gewicht <= 0;
    return 1;
end;

Gebruik: **SELECT** ruim_art **AS** weg **FROM** dual;

Er wordt volgende fout door SQL*Plus gegeven:

```
select ruim_art as weg from dual
       *
ERROR at line 1:
ORA-14551: cannot perform a DML operation inside a query
ORA-06512: at "E400.RUIM_ART", line 3
ORA-06512: at line 1
```

Een mogelijkheid is gebruikmaken van SQL*Plus variabelen:

```
SQL> VARIABLE res number
SQL> EXECUTE :res := ruim_art
SQL> PRINT res
```

Een andere mogelijk is deze actie te definiëren als een procedure:

create or replace procedure ruim_art **is**
begin
    delete from a **where** a.gewicht <= 0;
end;

en uit te voeren als `EXECUTE ruim_art`

4.10 Procedures

4.10.1 Een procedure met een OUT argument

create or replace procedure ruimop_art(res **OUT integer**) **is**
begin
    delete from a **where** a.gewicht <= 0;
    res := 1;
end;

Gebruik:
```
declare e integer;
begin
    ruimop_art(e);
    dbms_output.put_line('res ' || e );
end;
```

```
Alternatief:    SQL> VARIABLE ee number
                SQL> EXEC  ruimop_art(:ee)
                SQL> PRINT ee
```

4.10.2 Een procedure met IN en OUT argumenten

```
create or replace procedure ig( no IN a.anr%TYPE,
                                ext IN a.gewicht%TYPE, res OUT integer) is
begin
    update a set  a.gewicht = a.gewicht + ext  where a.anr = no;
    res := 1;
end;
```

Opmerking: de naam van een argument (bijv. `no`) mag niet gelijk zijn aan de naam van een veld in de tabel (bijv. `a.anr`).

Gebruik:
```
declare e integer;
begin
    ig ('A3', 5, e);
    dbms_output.put_line('res ' || to_char(e) );
end;
```

```
Alternatief:    SQL> VARIABLE ee number
                SQL> EXEC  ig('A2', 7, :ee)
                SQL> PRINT ee
```

4.11 Cursors

Het verwerken van query resultaten rij per rij gebeurt met behulp van een cursor.
Declaratie: **cursor** <cursornaam> [(<parameterlijst>)] **is** <**select** statement>]

4.11.1 Een FOR loop

```
create or replace function gewogen(mas IN a.gewicht%type) return integer is
        cursor acur IS SELECT * from a;
        som integer := 0;
        x   a%ROWTYPE;
begin
        OPEN acur;
        loop
              FETCH acur into x;
              exit when acur%NOTFOUND;
              if x.gewicht < mas then
                    som := som + x.gewicht;
              end if;
        end loop;
        CLOSE acur;
        return som;
end;
```

Gebruik: **SELECT** gewogen(30) **AS** weinig **FROM** dual;

Attributen van een cursor die kunnen getest worden:

%NOTFOUND : heeft de waarde `NULL` voor de eerste fetch;
heeft de waarde `false` indien de meest recente fetch een tuple gelezen heeft.

%FOUND : logisch tegengestelde van %NOTFOUND.

%ISOPEN : heeft de waarde `true` als de cursor geopend is.

%ROWCOUNT : aantal rijen reeds opgehaald uit de cursor.

4.11.2 Een cursor FOR loop

```
create or replace function gewogen(mas IN a.gewicht%type) return integer is
    cursor acur(m a.gewicht%TYPE) IS SELECT * from a
                                    WHERE gewicht < m;
    som integer := 0;
    x   a%ROWTYPE;
begin
    for x in acur(mas) loop
        som := som + x.gewicht;
    end loop;
    return som;
end;
```

Gebruik: **SELECT** gewogen(30) **AS** weinig **FROM** dual;

4.11.3 Een impliciete cursor

In plaats van een CURSOR naam te declareren en deze te gebruiken, kan de FOR lus gedefinieerd worden met behulp van een SELECT statement tussen haakjes.

```
create or replace function gewogen(mas IN a.gewicht%type) return integer is
    som integer := 0;
    x   a%ROWTYPE;
begin
    for x in (select * from a where gewicht < mas)
    loop
        som := som + x.gewicht;
    end loop;
    return som;
end;
```

Gebruik: **SELECT** gewogen(30) **AS** weinig **FROM** dual;

4.11.4 Aanpassingen bij gebruik van een cursor

```
create or replace function pasaan(mas IN a.gewicht%TYPE) return integer is
    cursor acur(m a.gewicht%TYPE) is SELECT * FROM a WHERE gewicht < m
                                    FOR UPDATE;
    teller  integer := 0;
    x       a%ROWTYPE;
    begin
        for x in acur(mas) loop
            UPDATE a SET gewicht = x.gewicht * 2 WHERE CURRENT OF acur;
```

```
            teller := teller + 1;
        end loop;
        return teller;
    end;
```

Gebruik met behulp van een SQL*Plus variabele:

```
SQL> VARIABLE aantal number
SQL> EXECUTE :aantal := pasaan(25)
SQL> PRINT aantal
```

4.11.5 Een functie met samengestelde argumenten

```
create or replace function dubbel(x IN a%ROWTYPE) return integer is
begin
      return 2 * x.gewicht;
end;
```

Gebruik:

```
declare
    cursor x is select * from a;
    r    integer := 0;
    y    a%ROWTYPE;
begin
    for y in x loop
        r := dubbel(y);
        dbms_output.put_line( y.anaam || '    ' || to_char(r) );
    end loop;
end;
```

Voorbeeld 2:

```
create or replace function teveel(art IN a%ROWTYPE, mas IN integer) return boolean is
begin
    if art.gewicht IS NULL then
        return false;
    end if;
    return art.gewicht > mas;
end;
```

Gebruik:

```
declare
    cursor acur is SELECT * FROM a WHERE pla = 'Hever';
    r   boolean := false;
    y   a%ROWTYPE;
begin
    for y in acur loop
        r := teveel(y,40);
        if r then
            dbms_output.put(y.anr || '   ' );
        end if;
    end loop;
    dbms_output.new_line;
end;
```

4.12 Exceptions

```
create or replace function gekleurd(kleurtje IN a.kleur%TYPE) return varchar2 is
    x   a%ROWTYPE;
begin
    SELECT * INTO x FROM a WHERE kleur = kleurtje;
    return x.anaam;
    exception
        when NO_DATA_FOUND then
            raise_application_error (-20030,'geen artikels in het '|| kleurtje );
        when TOO_MANY_ROWS then
            raise_application_error (-20031,'teveel artikels in het '|| kleurtje );
        when others then
            raise_application_error (-20032,'niet te forceren ');
end;
```

Gebruik:
```
SQL> SELECT gekleurd('groen') AS artikel FROM dual;
SQL> SELECT gekleurd('geel') AS artikel FROM dual;
SQL> SELECT gekleurd('rood') AS artikel FROM dual;
```

Systeem-exceptions:

CURSOR_ALREADY_OPEN : (ORA-06511) openen van een cursor die reeds open is;

INVALID_CURSOR : (ORA-01001) niet toegelaten cursor operatie (bijvoorbeeld het fetchen van een gesloten cursor);

NO_DATA_FOUND : (ORA-01403) geen tuple gelezen (door een SELECT of een FETCH);

TOO_MANY_ROWS : (ORA-01422) een SELECT ... INTO resulteert in meer dan 1 tuple;

ZERO_DIVIDE : (ORA-01476) poging om een deling door nul te doen.

Merk op:

```
create or replace function gekleurd(kleurtje IN a.kleur%TYPE) return varchar2 is
    x   a%ROWTYPE;
begin
    SELECT * INTO x FROM A WHERE kleur = kleurtje;
    if SQL%NOTFOUND then
        raise_application_error (-20030,'geen artikels in het '|| kleurtje );
    end if;
    return x.anaam;
end;
```

SQL cursor : een impliciete cursor die Oracle opent bij het verwerken van een SQL statement dat niet met een expliciete cursor gerelateerd is.
Bijvoorbeeld: SELECT, INSERT, UPDATE, DELETE.
Indien een SELECT INTO statement geen rij terug geeft, wordt door PL/SQL de voorgedefinieerde exception `NO_DATA_FOUND` gegenereerd, onafhankelijk van het feit of het `%NOTFOUND` attribuut van de impliciete SQL cursor getest wordt of niet (zoals in bovenstaand voorbeeld).

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 **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**), ...

Voorbeeld. De basisopleiding master elektronica-ICT (EI), fase 4, bevat twee afstudeerrichtingen: elektronica (E) en informatica (I). Opleidingscoördinator is Jan Meel (JME). De activiteit **smart embedded electronics** van type hoorcollege met docent Toon Goedemé (TGO) met een omvang van 18 contacturen wordt in deze basisopleiding opgenomen. Daarnaast is er een specifiek practicum *besturingssystemen* met 24 contacturen in lokaal A217 met twee begeleidende docenten dat alleen opgenomen is in de afstudeerrichting I. Deze activiteit wordt in twee groepen georganiseerd; beide groepen worden toegewezen aan Kylian Van Dessel (KVD) en Bram Aerts (BAE) samen.
Daarnaast is er opleiding bouwkunde (BK), tweede fase waarin alle practica en oefeningen voor twee groepen moeten ingepland worden. De oefeningenactiviteit *structuurmechanica 2* (18 contacturen) wordt voor twee groepen door Dennie Jansen (DJA) gedaan. De derde groep is toegewezen aan Wendy Busschots (WBU). Programmacoördinator van BK is Inge Deygers (IDE).

Extract van de inhoud van de tabellen

docent			
dafk	dnaam	dgeslacht	dgbd
AVH	Van Haperen An	v	29/12/1970
AVA	Van Assche Ado	m	16/12/1960
BTA	Tanghe Bart	m	12/08/1962
CCA	Cammaert Chris	m	06/11/1962
DMO	Moens David	m	06/03/1963
DPA	Pauwels Danny	m	21/04/1963
GDS	De Samblanx Gorik	m	29/03/1966
HCR	Crauwels Herman	m	21/04/1967
JMA	Mangelschots Johan	m	25/12/1969
LAP	Appels Lise	v	05/05/1971

activiteit		
aid	anaam	ascu
1000	chemie	24
1010	chemie	36
1020	chemie	12
2055	informatietechnolog	24
2057	informatietechnolog	36
2425	machinecomponenten	24
2430	machinecomponenten	12
2431	machinecomponenten	36
4010	databanken	24
4020	databanken	24
4265	biochemie 1	24
4270	biochemie	36
5197	ingebedde systemen	36
5200	objectgericht ontw	24
5210	objectgericht ontw	36
5595	hardware	24
5710	besturingssystemen	24
5720	besturingssystemen	36
5730	systeemprogr	24
5740	systeemprogr	36

lokaal			
lid	lnaam	lcapa	lbeamer
A002	oefeningen	24	n
A014	hoeklessen	67	j
A017	lessen	70	j
A102	oefeningen	32	n
A117	wenklessen	67	j
A118	tafellessen	48	j
F110	massalessen	112	j
K103	basislessen	400	j
K107	basisoefeningen	24	j
D019	fysica	24	j
D110	analytische chemie	16	n
D113	microbiologie	16	n
A111	basiselektronica	24	j
A213	PC-lokaal	24	j
A217	HP terminal-lokaal	18	n
C015	werktuigmachines	14	n

theorie	
taid	tdafk
1000	RDE
2055	HCR
4010	HCR
4265	HRE
5200	HCR
5595	FNA
5710	HCR
6050	JIV

oefpra
oaid
1010
1020
2057
2430
2431
4020
4270
5197
5210
5720
5740

groepoefpra		
aid	dafk	aant
1010	CCA	4
1010	LAP	6
2057	AVH	3
2057	APH	3
2057	FNA	2
2430	DMO	2
2431	DMO	2
4020	AVH	3
4020	APH	2
4270	AVA	2
5197	DPA	1
5210	AVH	2
5720	HCR	2
5720	APH	2
6060	DPA	4

practica		
paid	plid	padoc
1010	D110	1
2057	A213	2
2431	C015	2
4020	A217	1
4270	D113	1
5197	A213	1
5210	A213	1
5720	A217	2
6060	C019	1

basis						
bafk	bnaam	bfase	bastd	bagrp	bpc	bsoort
ab1	aca bach	1	200	10	GDS	z
ab2	aca bach	2	140	7	GDS	z
abei3	aca bach ei	3	10	1	HCR	z
abem3	aca bach em	3	30	2	JIV	m
abc2	aca bach chemie	2	40	2	LAP	z
abc3	aca bach chemie	3	30	2	LAP	m
mei4	master ei	4	30	2	HCR	m

minor					
bafk	mafk	mnaam	mfase	mastd	magrp
abc3	abcp3	acbac c chemie	3	20	2
abc3	abc3	acbac c biochemie	3	10	1
abem3	abemem3	acbac em elektromech	3	20	1
abem3	abemae3	acbac em automotive	3	10	1
mei4	meie4	master ea	4	15	1
mei4	meii4	master ei	4	15	1

basisact	
bafk	aid
ab1	1000
ab1	1010
ab1	1020
ab2	2055
ab2	2057
abem3	2425
abem3	2430
abem3	2431
abem3	4010
abc3	4010
abei3	4010
abem3	4020
abc3	4020
abei3	4020
abei3	5197
abei3	5200
abei3	5210
mei4	5595

minoract		
bafk	mafk	aid
abc3	abc3	4265
abc3	abc3	4270
mei4	meii4	5710
mei4	meii4	5720
mei4	meii4	5730
mei4	meii4	5740

fiche		
fid	fstp	fslu
1000	3	90
1010	2	50
1020	1	30
2055	2	60
2057	2	60
2425	3	90
2430	4	120
2431	4	120
4010	3	90
4020	2	50
4265	3	90
4270	3	90
5197	4	120
5200	3	90
5210	3	90
5595	3	90
5710	3	90
5720	3	90
5730	4	120
5740	4	120

Fysische dataorganisatie

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	+	/

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	+	/

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	+	/

ER diagramma