Mi minden van egy (Oracle) adatbázisban?

Adatbázisok szintjei

1. Fizikai szint (fájlok)
   Adatfájlok (ebben vannak tárolva ténylegesen az adatok -> ***.dbf**)
   Naplófájlok (ebbe íródnak a módosításokról készült naplóbejegyzések -> ***.log**)
   Vezérlőállományok (mindenféle információk az adatbázisról -> ***.ctl**)
   Paraméterállomány (**init.ora**), jelszóállomány... stb.

2. Logikai szint
   Az adatbázis objektumai (tábla, nézet, index ... stb.)

Séma: egy adott felhasználó tulajdonában lévő összes objektum
(de vannak olyan objektumok is, amelyek egyik sémába se tartoznak bele)
=> Séma objektumok és Sémán kívüli objektumok ('közösek', amikre mindenkinek szüksége van)
A táblák mindig valamilyen sémában vannak! (azért van kivétel)

Például: egy adatbázis kapcsolat (database link) az séma objektum, de lehet public database link-et is létrehozni, ami már sémán kívüli objektum.
Hasonló a helyzet a szinonimával.

Van két speciális séma: **SYS** és **SYSTEM**
(kötelezően létrejönnek, ebben vannak a rendszerkatalógus táblák)

Egy objektumra történő teljes hivatkozás: schema.objektum@database_link
(-> elosztott adatbázis)

Táblák tulajdonságainak vizsgálata

```
describe T1; -- egy tábla oszlopainak és azok típusának kiíratása
```

Érdekesség: az adatbázis-kezelő nem ismer describe-ot, már a kliens 'lefordítja' egy SQL utasításra (valamilyen select-re)
De akkor honnan veszi a rendszer a tábla oszlopaira vonatkozó infokat?
-> Rendszerkatalógus

Rendszerkatalógus: táblákból (nézetekből) áll, melyek az adatbázis telepítésekor létrejönnek és különböző adminisztrációs infókat tárolnak. Nevezik adatszótárnak, rendszer-katalógusnak vagy meta-adatbázisnak is.
Valójában nem is igazi táblák, hanem nézetek.
Általában beszédes nevük van: prefix_nev

Példák prefixre:
USER (az adott user tulajdonában lévő objektumok) pl. USER_TABLES
ALL (amihez joga van az adott usernek) pl. ALL_TABLES
DBA (az adatbázis összes objektuma) pl. DBA_TABLES

Például az egyes táblák oszlopairól a DBA_TAB_COLUMNS táblából lehet infóhoz jutni.

Azért egy igazi rendszerben egy 'mezei' felhasználónak nincs joga mindenhol nézelődni. Önöknek most van.

Az összes katalógus tábla neve és oszlopai -> Oracle doksi: Reference Part II Static Data Dictionary Views

A legfontosabb adatszótár, ahonnan érdemes kiindulni a kereséskor:

DBA_OBJECTS (ALL_OBJECTS)

Táblák létrehozása

```
CREATE TABLE tablanev(o1 tipus ... )
```

Típusok:
- char(n) (fix hossz, statikusan lefoglal n karakternyi helyet)
- varchar2(n) (változó hossz, maximum n karaktert lehet tárolni, de mindig csak annyit használ, amennyi épp szükséges)

Kétfajta karakter-összehasonlítási szemantika van!!
- blank-padded: a rövidebb stringet kiegészíti szóközökkel, hogy ugyanolyan hosszú legyen a két string és utána hasonlít össze
- non-padded: nem egészít ki semmit. Így nyilván a rövidebb string a kisebb.

Mikor fontos? például 'AB ' és 'AB' összehasonlítása esetén
blank-padded: 'AB ' = 'AB'
non-padded: 'AB ' > 'AB'
Fontos, hogy a blank-padded-et két CHAR típusú értéknél használja, minden más esetben a non-padded érvényes!

Numerikus tipusok:
- Number: (a leggyakoribb, és legsokoldalúbb)

Meg lehet neki adni paramétereket
Number(p,s) -> fixpontos tárolás
  p - precision (pontosság - jegyek száma)
  s - scale (tizedes jegyek száma, lehet < 0)
Number(20,-2) -> százasokra kerekít
Number(p) -> ilyenkor nincsenek tizedes jegyek - egész szám

- Integer, Float stb: a legtöbb esetben Numbert csinál belőle az Oracle

Példa:
```
CREATE TABLE num_proba(o_integer integer, o_float float, o_number number,
o_num10 number(10), o_num10_2 number(10,2));

SELECT column_id, column_name, data_type, data_length, data_precision,
       data_scale
FROM dba_tab_columns
WHERE table_name='NUM_PROBA';
```

Megjegyzés: az oracle a katalógus táblákban a neveket csupa nagybetűvel tárolja.

Dátum típus:
- Date: másodperc pontosságot tud

Dátum esetén fontos, hogy a lekérdezéskor milyen formátumban kapom meg a dátumot. Ez kliens függő! (Mármint az alapértelmezés, persze ezt a lekérdezésben meg tudom változtatni)

Konverziók: TO_DATE (karakter -> dátum)
            TO_CHAR (dátum -> karakter)

Aktuális rendszerdátum
```
  select sysdate from dual;
  select TO_CHAR(sysdate, 'yyyy-mm-dd:hh24:mi:ss') from dual;
```
-> lásd a formátummodelleket

```
create table t_datum(o date);
```

Gond lehet a beszúrásnál, mert el kéne találnom az alapértelmezett formátum stringet, amivel várja a dátumot! Megoldás -> to_date

```
insert into t_datum values(to_date('2007-02-19', 'yyyy-mm-dd'));

Vagyis a to_date()-nél úgy adom meg a dátumot, ahogy akarom, csak a második
paraméterben meg kell mondanom neki a formátum stringet, amit használtam.

Egyéb típusok:
  LOB  - Large OBject (több, mint 4 gigabájtos adatot is bele lehet pakolni)
  BLOB - Binary LOB
  CLOB - Characher LOB

ROWID - spec. típus, tulajdonképpen egy pointer, ami egy sornak a fizikai
helyére mutat

Dafault értékek:
Van egy táblám, aminek mondjuk van 3 oszlopa.
Mi történik ha egy INSERT-nél csak az első két oszlopnak adok értéket?
  1. NULL kerül a 3. oszlopba (de sokszor ezt megtiltjuk -> NOT NULL)
  2. Be lehet állítani ilyen esetre Default értéket az oszlopoknak

Példa:
CREATE TABLE t(fizetes number(6) DEFAULT 1000, hire_date date DEFAULT SYSDATE);
```

Szinonima

```
Egy táblának (vagy nézetnek) adhatok egy másodlagos nevet.

CREATE SYNONYM dolg_syn FOR dolgozo;

Akár távoli adatbázisban levő táblákra is létre lehet hozni.
Lehet publikus/nem publikus (ez utóbbi a saját sémámba tartozik)
```

Szekvencia

```
Egy sorszámgenerátor.
Olyan, mint az auto_increment MySQL-ben.
Megadható tulajdonságok:
- Lépésköz
- Kezdeti érték
- Maximális érték
- Körbefuthat-e (ha elérte a max-ot, újrakezdje-e)

Létrehozás: CREATE SEQUENCE

Hogyan lehet használni?

Szekv_nev.NEXTVAL: következő érték lekérése (be lehet írni pl. egy INSERT
utasításba is)
Szekv_nev.CURRVAL: akkor jó, ha beszúrás után kell még a kiosztott sorszám
Például beszúrok egy Osztályt, az okod-ot szekvenciával kérem le. Utána be
akarom szúrni az osztály dolgozóit, ehhez kell az előbb beszúrt osztály kódja.
```

Adatbázis kapcsoló (database link)

```
Egy másik Oracle adatbázishoz való kapcsolódást tesz lehetővé. Így egyszerre két
adatbázisban levő objektumokat is el tudunk érni egyetlen lekérdezésből. Az
egyik adatbázishoz kapcsolódunk a kliens programunkkal (pl. SqlDeveloper), és ez
az adatbázis fog a másikhoz kapcsolódni az adatbázis kapcsolóban megadott
információk segítségével. Ezek az információk tehát tartalmazzák az elérni
```

kívánt adatbázis elérhetőségét, a távoli adatbázisbeli felhasználónevet és jelszót. Például az ullman -> aramis kapcsolat létrehozása:

```
CREATE DATABASE LINK aramis CONNECT TO user1 IDENTIFIED BY jelszó1
USING 'aramis';
```

Az utolsó paraméter ('aramis') az ullman adatbázisszervert futtató számítógépen található tnsnames.ora állományban szereplő leíró részre (connect stringre) utal. A leíró részben szereplő információkat megadhatjuk közvetlenül az idézőjelek között is:

```
CREATE DATABASE LINK aramis2 CONNECT TO user1 IDENTIFIED BY jelszó1
USING 'aramis.inf.elte.hu:1521/aramis.inf.elte.hu';
```

A fentiek után egy lekérdezés, az ullman adatbázisban kiadva:

```
SELECT * FROM dolgozo d, osztaly@aramis o WHERE d.oazon = o.oazon;
vagy
SELECT * FROM dolgozo d, osztaly@aramis2 o WHERE d.oazon = o.oazon;
```

Fizikai tárolás

Azt szeretnénk megvizsgálni, hogy ha van mondjuk egy T1 táblánk, az hol és melyik fájlban van, valamint az abban található adatok ténylegesen hogyan vannak tárolva.

Adatblokk: a legkisebb tárolási egység (2K, 4K, 8K, ...)
(Minden fájlt az adatbázis-kezelő feloszt blokkokra.) A jelenlegi adatbázisunkban 8k a blokkméret.

Extens (extent, kiterjesztés): adatblokkokból álló összefüggő terület. Például ha egy fájl méretét növelni kell, akkor valahány extens-sel növeli a rendszer

Adatfájl: extensekből áll

Szegmens (segment): mindig több extensből áll, és egy szegmens pontosan egy objektumhoz (pl. tábla, index, klaszter) tartozik.

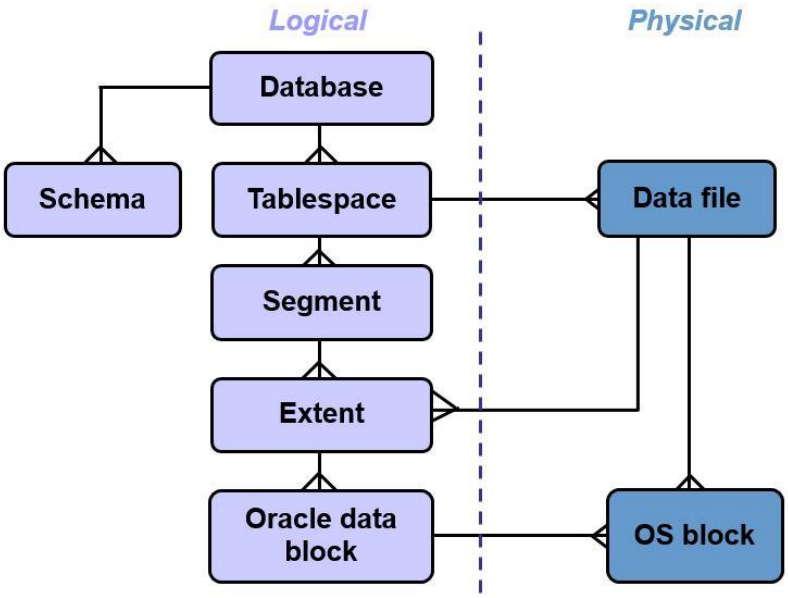
Átnyúlhat több adatfájlra is, de persze egy fájlban lehet több szegmens is.
Egy szegmens ~ Egy objektum

Táblatér (tablespace):

Táblatér - adatfájl kapcsolat 1-N
Táblatér - szegmens kapcsolat 1-N

Vannak kötelező táblaterek is: SYSTEM, SYSAUX stb.

A tárolással kapcsolatos fogalmak és azok egymással való kapcsolata:



Kapcsolódó katalógusok

DBA_DATA_FILES: a rendszer által használt adatfájlok információi
DBA_TEMP_FILES: a temporális táblaterek adatfájljainak információi
DBA_TABLESPACES: táblaterek információi

Szegmensekről bővebben

DBA_SEGMENTS, DBA_EXTENTS

Adatbázis objektumok:
- fizikai tárolással rendelkező objektumok
  pl. tábla, index, cluster

- fiz. tárolással nem rendelkező objektumok
  pl. procedura, nézet, szekvencia (csak a definíció van tárolva)

Szegmens sok mindenhez tarozhat, a DBA_SEGMENTS.SEGMENT_TYPE mutatja, hogy egy adott szegmens mihez tartozik

```
SELECT * FROM dba_segments
WHERE owner='NIKOVITS' AND segment_name='DOLGOZO' AND SEGMENT_TYPE='TABLE'
```

Régebben 1 tábla 1 szegmens volt, azonban mióta nagyon nagy táblákat is képes kezelni a rendszer, már nem célszerű 1 szegmensbe tenni egy nagy táblát.

Partícionálás:
egy táblát több szegmensre osztunk fel. Meg lehet adni, hogy milyen szempontok alapján darabolja fel a táblát. Például intervallumokat adhatunk meg -> Range partícionálás. Vagy van még Lista és Hash partícionálás, sőt lehet olyat is, hogy csinálok egy particionálást és az egyes partíciókat tovább lehet particionálni. Csak két szint mélységig lehet.
Később még lesz róla szó.

Érdekesség, hogy ilyenkor egy tábla már túlnyúlhat egy táblatéren. Csak azt lehet garantálni, hogy minden szegmense (partíciója) pontosan egy táblatéren van. Érdekesség az érdekességben: nem csak azért állhat egy tábla több szegmensből, mert nagyon sok adat van benne, hanem az is lehetséges, hogy

valamilyen LOB típusú adat van benne. Ilyenkor érdemes lehet a LOB típusú adatokat kirakni külön szegmensre.

Tárolási paraméterek

Amikor létrehozunk egy táblát, a CREATE TABLE-ben megadhatunk ún. tárolási paramétereket.

Megj.: nem csak a CREATE TABLE-nél lehet, hanem a CREATE INDEX, CREATE CLUSTER stb. esetén is, vagyis minden tárolást igénylő objektumnál.

```
CREATE TABLE tipus_proba(...)
TABLESPACE users
PCTUSED 50  PCTFREE 20  INITRANS 1 MAXTRANS 255
STORAGE (INITIAL 32K  MINEXTENTS 1  MAXEXTENTS 200  PCTINCREASE 0
         FREELISTS 1  FREELIST GROUPS  1  BUFFER_POOL  DEFAULT);
```

STORAGE(...): hogyan viselkedjenek az extensek, mekkorák legyenek, hogyan bővüljenek stb.
- INITIAL: első extens mérete
- NEXT: következő extens mérete
- PCTINCREASE: milyen mértékben növekedjenek az extensek (%-ban) az előzőhöz képest. (50 azt jelenti, hogy másfélszerese lesz a következő)
- MINEXTENTS: minimális extens darabszám (a tábla létrehozásakor ennyit automatikusan létrehoz)
- MAXEXTENTS: maximális extens darabszám

stb.

Az Oracle nem mindig hallgat ránk, mert vannak olyan peremfeltételek, amiket mindenképpen be kell tartania. Ilyen például a blokkméret. Hiába adunk meg bármekkora értéket is extens méretnek, annak mindenképpen a blokkméret többszörösének kell lennie. Az is előfordulhat, hogy az adott táblatéren valamilyen szabályok alapján osztja ki az oracle az extenseket, mert eleve így hoztuk létre a táblateret (lásd CREATE TABLESPACE utasítás paraméterei). Például van egy legkisebb extens (MINIMUM EXTENT) méret, aminél kisebbet nem hajlandó kiadni, vagy egy adott méret többszörösét adja ki mindig (UNIFORM).

Ilyen esetekben az Oracle a peremfeltételeinek figyelembe vételével hallgat csak a megadott utasításra.

PCTUSED, PCTFREE:
A blokkokkal hogyan gazdálkodjon a rendszer. Általában szokás úgy csinálni, hogy a blokkokat nem rakjuk tele azért, hogy egy későbbi módosítás után is még beférjenek a sorok a blokkba.
- PCTFREE: hány %-ot hagyjon szabadon
- PCTUSED: ha ez alá csökken a blokk foglaltsága, akkor szabadnak nyilvánítja a blokkot, és ismét enged beleírni.

Példa:
```
CREATE TABLE t1 (o1 NUMBER, o2 CHAR(20))
NOLOGGING
TABLESPACE users
PCTFREE 0 PCTUSED 50
STORAGE (INITIAL 100K NEXT 200K MINEXTENTS 3 MAXEXTENTS 5);
```

Mi van akkor, ha nem adok meg ilyen (tárolási) paramétereket?

Minden felhasználónak van egy default táblatere (lásd DBA_USERS-ben)

Minden táblatérnek vannak a tárolásra vonatkozó default értékei (lásd DBA_TABLESPACES)
És még vannak adatbázis-szintű alapértelmezések is.

A tábla növekedésekor az Oracle automatikusan újabb extenseket ad hozzá, ha szükség van rá. Mindezt manuálisan is kezdeményezhetjük, és a nem használt extenseket fel is szabadíthatjuk.

```
ALTER TABLE emp ALLOCATE EXTENT
(SIZE 200K DATAFILE '/big/oracle/oradata/oradb/users01.dbf');

ALTER TABLE emp DEALLOCATE UNUSED;
```

Táblaterek

A tábla sok paraméterét a táblatér beállításaitól örökli ha külön nem adjuk meg.
A táblaterek létrehozásakor megadható paraméterek közül néhány fontosabb:

```
ONLINE | OFFLINE

BLOCKSIZE (táblaterenként szabályozható a blokkméret)

UNDO (nem tehető rá normális objektum, a tranzakció-kezeléshez használatos)

TEMPORARY (csak átmeneti szegmensek kerülhetnek rá, pl. rendezéskor)

DATAFILE … AUTOEXTEND ON | OFF
  Adatfájlonként megadható, hogy azok automatikusan növekedhetnek-e és meddig.

MINIMUM EXTENT (A fragmentáció elkerülése céljából megadható, hogy mekkora lehet
                a legkisebb extens a táblatéren.)

DEFAULT STORAGE (INITIAL, NEXT ...  stb.)
  A tábla innen örökli, ha külön nem adtuk meg.

EXTENT MANAGEMENT {DICTIONARY | LOCAL [AUTOALLOCATE | UNIFORM]}
Az extensek kezelését lehet itt megadni. A lokálisan menedzselt táblatér
extenseinek méretét az Oracle vagy automatikus módon határozza meg, vagy
egyforma méretű extenseket használ.

SEGMENT SPACE MANAGEMENT {MANUAL | AUTO}
Manuális esetben a szegmensen belüli szabad helyeket szabad listák segítségével
kezeli az oracle (lásd FREELISTS), automatikus esetben pedig egy bitmap térkép
segítségével.
```

Példák:

```
CREATE UNDO TABLESPACE undots1 DATAFILE 'undotbs_1a.f'
   SIZE 100M AUTOEXTEND ON NEXT 20M MAXSIZE 400M;

CREATE TABLESPACE tbs_1 DATAFILE 'tabspace_file2.dat' SIZE 200M
   DEFAULT STORAGE (INITIAL 100K NEXT 200K MINEXTENTS 1 MAXEXTENTS 999)
   ONLINE;

CREATE TABLESPACE tbs_4 DATAFILE 'file_1.f' SIZE 100M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 128K;

CREATE TABLESPACE auto_seg_ts DATAFILE 'file_2.f' SIZE 100M
   EXTENT MANAGEMENT LOCAL
   SEGMENT SPACE MANAGEMENT AUTO;

CREATE TEMPORARY TABLESPACE temp
```

```
   TEMPFILE 'temp01.dbf' SIZE 5M AUTOEXTEND ON;
```

(Ez utóbbit nem a DBA_DATA_FILES katalógusban, hanem a DBA_TEMP_FILES katalógusban találjuk)

Indexek

Az indexek szempontjából fontos a **ROWID típus** ismerete

Speciális mutató típus, amivel egy sort lehet azonosítani.
Minden sorról meg tudom mondani, hogy egy adott sor:
- melyik adatfájlban van
- azon belül melyik blokkban
- azon belül hányadik rekord

Ez nincs a táblában tárolva, de mégis úgy viselkedik: pszeudo oszlop

Példa:
SELECT **rowid**, empno, ename, sal FROM nikovits.emp;

18 karakteren íródik ki, a következő formában: OOOOOOFFFBBBBBBRRR
OOOOOO - az objektum azonosítója
FFF - fájl azonosítója (táblatéren belüli relatív sorszám)
BBBBBB - blokk azonosító (a fájlon belüli sorszám)
RRR - sor azonosító (a blokkon belüli sorszám)

A ROWID megjelenítéskor 64-es alapú kódolásban jelenik meg.
Az egyes számoknak (0-63) a következő karakterek felelnek meg:
A-Z -> (0-25), a-z -> (26-51), 0-9 -> (52-61), '+' -> (62), '/' -> (63)

Indexek létrehozása

CREATE INDEX (vagy ALTER INDEX) utasításban adhatók meg az index paraméterei.
A hagyományos index B-fa szerkezetű, a fa leveleiben vannak a bejegyzések, és mellettük a sorazonosító (ROWID). Az index lehet 1 vagy többoszlopos. Ha egy érték többször szerepel a táblában, akkor az indexben is többször fog szerepelni, minden sorazonosítóval külön-külön. A levélblokkok mindkét irányban láncolva vannak, így növekvő és csökkenő keresésre is használható az index. (pl. WHERE o > x vagy WHERE o < y) A csupa NULL érték nem szerepel az indexben bejegyzésként.
Érdemes a NULL értékek helyett DEFAULT-ot használni, épp az előzőek miatt.

Az indexek esetén is megadhatók tárolási paraméterek, hasonlóan a táblákhoz.

Az index létrehozásakor megadható legfontosabb paraméterek:

UNIQUE -> egyedi index létrehozása

ASC | DESC -> növekvő vagy csökkenő sorrend szerint épüljön-e fel az index

```
CREATE UNIQUE INDEX emp1 ON EMP (ename);
CREATE INDEX emp2 ON emp (empno, sal DESC);
```

REVERSE
(Fordított kulcsú index létrehozása)
A kulcsoszlop bájtjai az indexben fordított sorrendben vannak. Ha több oszlopos az index, az oszlopok sorrendje nem változik. Ez főleg akkor hasznos, ha szekvencia alapján töltünk fel egy táblát (amikor a kulcsok sorban egymás után kerülnek kiosztásra), mert a fordított kulcsú index egyenletesen elosztja a bejegyzéseket az indexben. Viszont az ilyen index nem használható intervallum jellegű keresésekhez, mert a szomszédos értékek nem egymás mellett helyezkednek el.

```
CREATE INDEX emp3 ON emp (empno, sal) REVERSE;
```

Index újraépítése
(időnként hasznos, mivel a törölt sorok bejegyzései fizikailag nem törlődtek ki)

```
ALTER INDEX i1 REBUILD TABLESPACE ts1 [REVERSE | NOREVERSE];
```

A fenti utasítással új táblatérre tehető az index, a logikailag törölt bejegyzések helye felszabadul, és fordított kulcsúvá ill. normálissá is tehető az index.

NOSORT
Azt jelezzük vele, hogy nem kell rendezni az index létrehozásakor, mert a sorok már rendezve vannak. Ha mégsem így van, az oracle hibát jelez. Pl.

```
CREATE TABLE ind_t(o1 int, o2 varchar2(20), o3 char(10));
BEGIN
  FOR i IN 1..100 LOOP
    INSERT INTO ind_t VALUES(i, 'sor'||to_char(i)||'-BLABLA', 'ABC');
  END LOOP;
  COMMIT;
END;
CREATE INDEX ind_t_ix ON ind_t(o1) NOSORT;
```

Ha még egy (o1=1) sort beszúrnánk a táblába, akkor már hibaüzenetet kapnánk a fenti NOSORT-ra.

COMPRESS <n>
A kulcs értékek ismételt tárolását szüntetjük meg vele az index első n oszlopában. Vagyis ezek a kulcsértékek csak egyszer lesznek tárolva, és mellettük több sorazonosító lesz, azoknak a soroknak megfelelően, amelyek az adott értékkel rendelkeznek.

```
CREATE INDEX emp4 ON emp (empno, ename, sal) COMPRESS 2;
```

Függvény alapú index
Akkor hasznos, ha a lekérdezésben is e kifejezés szerint keresünk.
(plusz infók a katalógusban -> DBA_IND_EXPRESSIONS)

```
CREATE INDEX ind_t_ix3 ON ind_t (SUBSTR(o2, 1, 5), UPPER(o3));
```

BITMAP
Bitmap index létrehozása
Hasonló a B-fa indexhez, de a levelekben a kulcsérték mellett nem a ROWID-k tárolódnak, hanem egy bittérkép. (Az első és utolsó érintett ROWID valamint a köztük lévő sorokra vonatkozó bittérkép.) Minden sornak egy bit felel meg, ami azokra a sorokra lesz 1-es, amelyek az adott értéket tartalmazzák. Módosításkor az egész bittérképet zárolni kell, így a bittérkép által érintett sorok sem módosíthatók a tranzakció végéig.

```
CREATE BITMAP INDEX ind_t_ix4 ON ind_t (o2);
```

Index szervezett tábla (Index Organized Table -> IOT)
Együtt (egy szegmensen) tárolódik a tábla és az index, ilyen esetben kötelező elsődleges kulcs megadása. Létrejön egy logikai tábla (szegmens nélkül), egy index szegmens, és egy túlcsordulási szegmens (ahová a sorok vége kerül).

PCTTRESHOLD:
  Egy index bejegyzés a blokknak hány százalékát foglalhatja el. Ha ennél nagyobb a sor -> túlcsordulás
INCLUDING:

```
  Mely oszlopok tárolódjanak együtt a kulccsal. A megadott oszlop utániak ->
  túlcsordulási szegmensre
OVERFLOW:
  Ha nem adjuk meg, akkor nem is hozza létre a túlcsordulási szegmenst, csak ha
  szükség lesz rá.

CREATE TABLE cikk_iot
  ( ckod integer,
    cnev varchar2(20),
    szin varchar2(15),
    suly float,
       CONSTRAINT cikk_iot_pk PRIMARY KEY (ckod)  )
ORGANIZATION INDEX
PCTTHRESHOLD 20 INCLUDING cnev
OVERFLOW TABLESPACE users;

A fenti utasítás hatására két szegmens jön létre (egy index és egy tábla),
valamint egy olyan objektum, amihez nem tartozik szegmens, és így nincs is
DATA_OBJECT_ID-ja !!!


Információk az indexekről a katalógusban:
DBA_INDEXES
DBA_IND_COLUMNS  (indexbeli oszlopok)
DBA_IND_EXPRESSIONS  (függvény alapú index kifejezései)

Index szervezett tábla index része:
  DBA_INDEXES -> index_type és table_name oszlopok
Index szervezett tábla tábla része:
  DBA_TABLES -> iot_name és iot_type oszlopok
```

A ROWID egy további használata

```
A ROWID általában minden olyan esetben használható, amikor egy konkrét sorra
mutató pointerre van szükségünk. Például egy constraint utólagos létrehozásakor
megadható, hogy a már létező, a constraintet megsértő sorok sorazonosítói mely
táblába íródjanak be, hogy ezekkel a sorokkal valamit tenni tudjunk.
Alapértelmezés szerint (ha nem adjuk meg a kulcsszót) az EXCEPTIONS nevű táblába
íródnak be, aminek a létrehozását a következő utasítással végezhetjük el:
  CREATE TABLE exceptions(row_id rowid, owner varchar2(30),
         table_name varchar2(30), constraint varchar2(30));
lásd az oracle által előre megadott utlexcpt.sql scriptet

Hozzunk létre egy példa táblát:
CREATE TABLE const_t (o1 number, o2 char(10), o3 date );

Szúrjunk be néhány sort, amelyek közül néhány megsérti az alább megadandó
constraintet.

INSERT INTO const_t VALUES(1, 'Egy', TO_DATE('2004.01.02','yyyy.mm.dd'));
INSERT INTO const_t VALUES(2, 'Ket', TO_DATE('2003.01.02','yyyy.mm.dd'));
INSERT INTO const_t VALUES(3, 'Har', TO_DATE('2005.01.02','yyyy.mm.dd'));

Az alábbi utasítást nem fogja végrehajtani a rendszer, és hibaüzenetet kapunk,
viszont feltölti az exceptions táblát a megfelelő sorokkal.

ALTER TABLE const_t ADD CONSTRAINT const_c
   CHECK(o3 > TO_DATE('2005-jan-01', 'YYYY-mon-dd'))
   EXCEPTIONS INTO exceptions;

Erről az alábbi lekérdezéssel győződhetünk meg:
```

```
SELECT * FROM const_t WHERE ROWID IN (SELECT ROW_ID FROM exceptions);

  O1 O2         O3
---- ---------- -----------
   2 Egy        2003-jan-02
   1 Egy        2004-jan-02
```

Particionálás

Egy tábla fel van osztva több partícióra.
Minden partíció egy-egy szegmens, kivéve ha alpartícionálás is van, mert ilyenkor egy alpartíció alkot egy szegmens-t. Ha ügyesen partícionáltuk a táblát, akkor a lekérdezésnek lehet, hogy nem kell a teljes táblát végignéznie, hanem csak egy (vagy néhány) partíciót. A felosztás történhet intervallumok alapján, hash módszerrel, vagy lista alapján. Az egyes partíciók tovább oszthatók -> SUBPARTITION (alpartíció)
Példák partícionált tábla létrehozására -> **cr_part_table.txt**

Particionált index

Lokális: Megpróbálja szinkronban létrehozni a tábla partíciókat és az index partíciókat. 1 tábla partíció <-> 1 index partíció

Globális: Amikor nincs meg ez az összerendelés.

Prefixelt: Az indexet alkotó oszloplista egy-egy prefixe tartozik egy-egy partícióba. Például ha ckod, cnev alapján indexelek, mondhatom azt, hogy az első partícióba tartozzanak azok a bejegyzések, ahol a ckod kisebb, mint 200, a másodikba azok, ahol a ckod kisebb mint 500 (de nagyobb, mint 200)

Nem prefixelt: amikor nincs meg a fenti összefüggés az index oszlopok és a partícionáló oszlopok között.

Ha a partícionált index lokális, annak is van előnye, ha prefixelt annak is van előnye. Ha egyik előnye sincs meg, akkor nincs értelme az indexet partícionálni.
-> Nem lehet létrehozni globális, nem prefixelt indexet!!!

Példák partícionált index létrehozására -> **cr_part_index.txt**

Információk a partícionált táblákról és indexekről a katalógusban:
DBA_PART_TABLES, DBA_PART_INDEXES, DBA_TAB_PARTITIONS, DBA_IND_PARTITIONS, DBA_TAB_SUBPARTITIONS, DBA_IND_SUBPARTITIONS, DBA_PART_KEY_COLUMNS

Clusterek

Kicsit ellentétes filozófiát képvisel, mint a normalizálás.
Pl. Vannak Osztályok és Dolgozók
Normalizálva:
Osztály(okod, nev, telephely, ...)
Dolgozo(dkod, okod, nev, ...)

Nagy táblák esetén ezzel az a baj, hogy nagyon szét vannak szórva az adatok. Ha például a 10-es kódú osztály dolgozóiról akarok információkat, akkor rengeteget kell olvasgatni a lemezről.

Ami jó lenne: legyenek együtt tárolva az egyes osztály dolgozói

Erre jó a cluster: több táblát fizikailag egy egységként (egy szegmensben) tárol.

Mi kell ahhoz, hogy használhassunk cluster-eket?
Legyen egy vagy több közös oszlopa a tábláknak.
Közös oszlop: az oszlopok típusa kell, hogy megegyezzen (a név mindegy)

A clusterek esetén megint nem igaz, hogy egy tábla egy szegmens, hanem:
Egy cluster = egy szegmens

Cluster létrehozása: CREATE CLUSTER
Meg kell adni a kulcsát, kulcsait is. Utána majd ehhez kell kapcsolni a táblák "közös" oszlopait.

```
CREATE CLUSTER personnel
  ( department_number NUMBER(2) )
  SIZE 512
  STORAGE (INITIAL 100K NEXT 50K);
```

A cluster is egy szegmens, és így megadhatók a tárolásával kapcsolatos paraméterek.
TABLESPACE, STORAGE (…) stb.

Táblák feltevése a clusterre:

```
CREATE TABLE emp_cl
  ( empno NUMBER PRIMARY KEY,ename VARCHAR2(30), job VARCHAR2(27),
    mgr NUMBER(4), hiredate DATE, sal NUMBER(7,2), comm NUMBER(7,2),
    deptno NUMBER(2) NOT NULL)
  CLUSTER personnel (deptno);

CREATE TABLE dept_cl
  ( deptno NUMBER(2), dname VARCHAR2(42), loc VARCHAR2(39))
  CLUSTER personnel (deptno);
```

Érdekességek:
A cluster kulcs csak egyszer tárolódik
Akkor jó igazán, ha egyenlőséges feltételt használok lekérdezésnél, és nem az összes osztály információ kell, hanem csak néhány.
Például csak a 10-es és 20-as osztályokról akarok infót.
Nyílván ez csak akkor lehet hatékony, ha gyorsan el tudok érni egy részt a clusterből

```
|10 osztály sor |
  |    dolgozó sor  |
  |    dolgozó sor  |
  |    dolgozó sor  |
|20 osztály sor |
  |    dolgozó sor  |
  |    dolgozó sor  |
|30 osztály sor |
  |    dolgozó sor  |
stb.
```

Hogyan lehet gyorsan keresni a clusterben?
Kell valamilyen segédstruktúra. Szó szerint KELL, amíg nem csinálok egy ilyet, nem is enged beszúrni a clusterbe sorokat. Ez a segédstruktúra lehet index, vagy hash táblázat.

Index cluster
Úgy működik, ahogy ezt már láttuk
Csak a cluster kulcsra lehet létrehozni az indexet, azaz nem is kell megadni az indexnél az oszlopot, amire az indexet létrehozom.

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

Csak az index létrehozása után enged sorokat beszúrni a táblákba!!!

```
INSERT INTO emp_cl SELECT * FROM emp;
INSERT INTO dept_cl SELECT * FROM dept;
```

Hash cluster
Egy hash függvény által visszaadott érték alapján helyezi el a kulcsértékeket.
Előre meg kell adnom, hogy hány hash érték lesz.

```
CREATE CLUSTER personnel1
( department_number  NUMBER )
  SIZE 512  HASHKEYS 500
  STORAGE (INITIAL 100K  NEXT 50K);
```

Vagyis a cluster létrehozásánál kell megadni, hogy hash clustert csináljon.
Ha nem adjuk meg a HASHKEYS-t, akkor index clustert készít.

Érdekesség: hiába adtam meg 500-at, valójában a következő prímszámot (503) fogja beállítani. (A hash fv működése miatt)
De megadhatok saját hash fv-t is:

```
CREATE CLUSTER personnel2
  ( home_area_code  NUMBER,
    home_prefix     NUMBER )
    HASHKEYS 20
    HASH IS MOD(home_area_code + home_prefix, 101);
```

Ez azért jó, mert így elvileg szabályozhatom, hogy hova kerüljenek az egyes sorok. Az azonos hash értékű sorok fizikailag ugyanoda kerülnek.

Például van két táblám és azt akarom, hogy ezeknek bizonyos sorai egy blokkba kerüljenek.
Megoldás: saját hash fv-t csinálok, mondjuk egy ilyet:

```
     HASH IS MOD(kulcs, 11)
```

és az egyes soroknak olyan cluster kulcs értéket adok, amik mod 11 megegyeznek.
Az elhelyezkedést utólag ellenőrizhetem ha megnézem a sorok ROWID-jét.

Tárolási paraméterek:
SIZE
Megadható, hogy az azonos klaszter kulcs értékkel rendelkező sorok számára mekkora helyet foglaljon le az oracle. Megadása valójában csak akkor fontos, ha azt szeretnénk, hogy egy blokkban több különböző klaszter kulccsal rendelkező sor is tárolódjon. Ha nem adjuk meg, vagy a mérete nagyobb a blokkméretnél, akkor az oracle minden klaszter kulcsot különböző blokkba tesz.

SINGLE TABLE
Egyetlen táblát tartalmazó hash klaszter létrehozása. Ez arra jó, hogy egy olyan táblát tudjunk létrehozni, amelynek a fizikai tárolása hash alapú elérést tesz lehetővé.

```
CREATE CLUSTER personnel3
   (deptno NUMBER)
   SIZE 512 SINGLE TABLE HASHKEYS 500;
```

Példák clusterek létrehozására -> **cr_cluster.txt**

Clusterekkel kapcsolatos katalógusok:
DBA_CLUSTERS
DBA_TABLES (cluster_name oszlop -> melyik klaszteren van a tábla)
DBA_CLU_COLUMNS (táblák oszlopainak megfeleltetése a klaszter kulcsának)
DBA_CLUSTER_HASH_EXPRESSIONS (hash klaszterek hash függvényei)