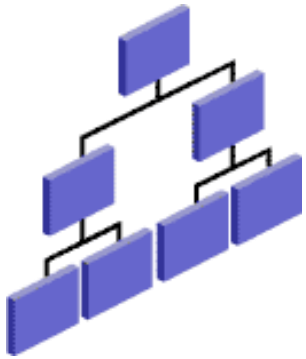
Kerepes Tamás – Czinkóczki László

Adatbázisok

Indexek, nézetek, adatbázisban tárolt programok, egyéb elemek a relációs adatbázisokban

Indexek a relációs adatbázisokban

- Az index egy speciális, belső adatszerkezet.
- A „fizikai” adatbázis része
- A relációs adatbázisokban az indexeknek két célja is lehet:
  - A PRIMARY KEY és UNIQUE kényszerek biztosítása
  - A lekérdezések felgyorsítása
- Az index a táblához hozzárendelt, de attól rendszerint függetlenül tárolt „objektum” az adatbázisban
- A DML-ek automatikusan módosítják (frissítik) az indexet is



Miért kell az index?

- Az egyedi index azért kell, hogy az adatintegritási szabály ellenőrzése az adatbáziskezelő részéről hatékony (gyors) legyen, de emellett gyorsíthatja is az SQL utasítások működést
- A nem-egyedi index a hatékonyságnövelés, azaz gyorsítás egyik eszköze (módja) lehet

Az indexek típusai

- Ha a tábla kulcsa egyoszlopos, akkor az index is arra az egy oszlopra épül (azt indexeli)
- A többoszlopos kulcsok többoszlopos indexeket eredményeznek. Ezeket még „konkatenált” (angolul: concatenated), vagy összetett indexeknek is nevezzük.
- Létezik egyedi és nem egyedi („non unique”) index. Az egyedieket általában nem a felhasználó explicit DDL parancsa hozza létre, hanem a felhasználó a táblának egy PRIMARY KEY vagy UNIQUE kényszerét hozza létre, és az index ilyenkor magától létrejön.
- A nem egyedi indexeket leginkább az adatbázis tervezői (az adatmodell elkészítői), a rendszergazdák vagy a fejlesztők szokták létrehozni

Az indexek explicit létrehozása és eldobása

Amennyiben nem a PRIMARY KEY vagy a UNIQUE kényszer eredményeképpen jön létre, hanem explicit módon, akkor a következő utasítást használjuk

- CREATE INDEX utasítás:

```
CREATE INDEX workers_name_ix
ON workers  (last_name, first_name);

index WORKERS_NAME_IX created.
```

Ha az indexre nincs többé szükségünk, azt eldobhatjuk:

```
DROP INDEX workers_name_ix;

index WORKERS_NAME_IX dropped.
```

Az indexek létrehozása és eldobása automatikusan

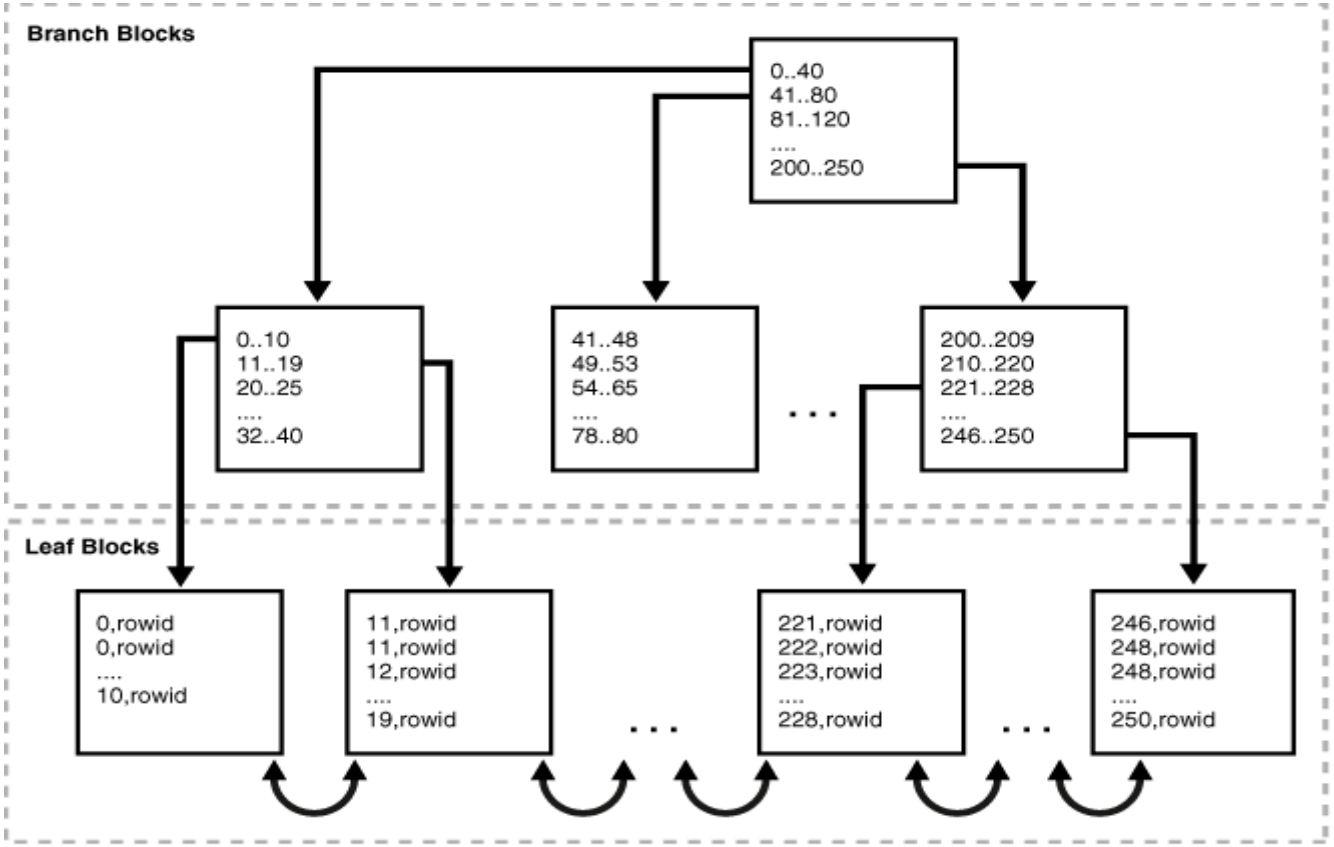
- Amennyiben PRIMARY KEY vagy UNIQUE kényszert hozunk létre, jellemzően azonnal és automatikusan létrejön egy egyedi index is, amely ezt a kényszert hivatott biztosítani (kikényszeríteni).
- Ha a kényszert eldobjuk (vagy csupán kikapcsoljuk), akkor az index automatikusan eldobódik
- Elvileg létezik SQL szintaxis egyedi indexek explicit létrehozására is, de ez kerülendő. Ilyenkor inkább kényszert kell deklarálni.
- Ha egy index automatikusan a kényszer létrehozásakor jött létre (tehát implicit módon), akkor jellemzően explicit módon (DROP INDEX) nem dobható el.

Az indexek típusai

- Nem írja elő a szabvány, hogy milyenek léteznek, de:
  - B-fa (vagy variációk erre a témára)
  - Függvény-alapú de leginkább B-fa típusú
  - Bittérképes
  - Egyéb (egyedi) megoldások egy-egy szakterületen
- A B-fa típusúak OLTP jellegű rendszerekben is beválnak
- A bittérképesek leginkább csak adattárházakban működnek jól (mert jellemzően nem sor szinten lockolódnak)

A B-fa index szerkezete nagyvonalakban

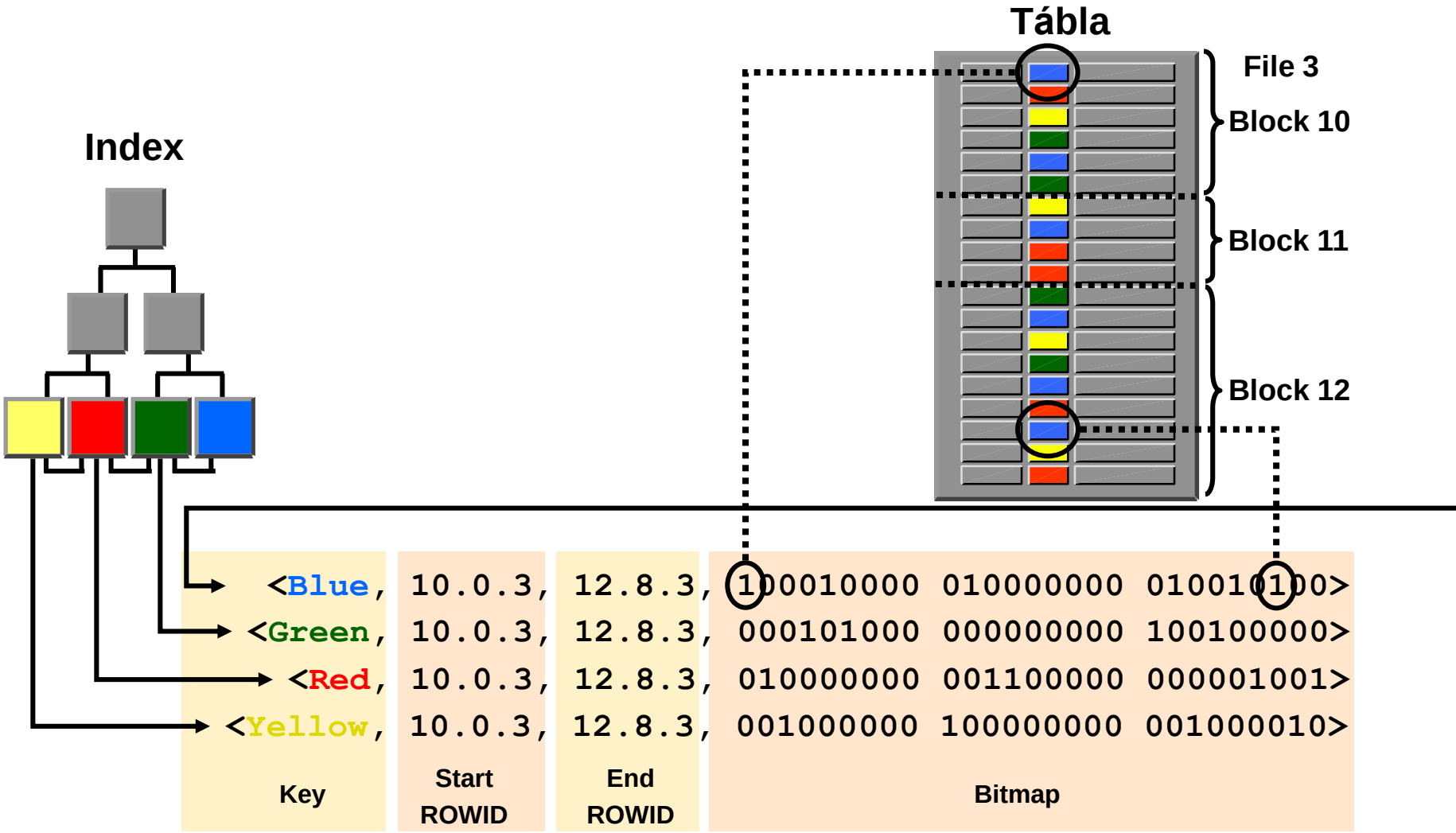
- Gyökér, közbülső csomópont és levél blokkokból áll
- A levélben kulcsértékek és mutatók (címek) vannak
- A tábla ugyan „rendezetlen”, de az index rendezett



A B-fa index alapvető tulajdonságai

- A B nem binárist, hanem „balanszírozottat jelent
- A balanszírozottságnak is elég „fura” a definiciója: minden irányban ugyanannyi lépés van a gyökértől a levélig
- Az index valójában sokkal „terebélyesebb” mint amit az ábrák sugallnak
- A B-fa jellemzően sorszinten zárolódik, csakúgy, mint a tábla
- Ez a sorszintű lockolás teszi lehetővé, hogy OLTP rendszerek jól működhessenek
- Egy-egy INSERT utasításkor jellemzően nehezebb az index frissítése, mint a sor beszúrása a táblába
- Különösen nehéz az olyan UPDATE-et végrehajtani, amelyben az indexelt oszlop értéke változik

A bittérképes index (nagy vonalakban)



A bittérképes index alapvető tulajdonságai

- Jellemzően nem sorszinten lockolódik.
- A folyamatos online karbantartása sokkal nehezebb
- Kis kardinalitású oszlopokra jobban beválik
- A gyártók többségénél a bitvektorok tömörítettek
- OLTP rendszereknél szinte mindig tragikusan rosszak
- Adattárházaknál a SELECT-et nagyon fel tudják gyorsítani
- Különösen az OR feltételek esetén múlják felül a B-fa indexeket a SELECT felgyorsításáában
- Gyakran az ETL folyamat része az is, hogy a bittérképes indexeket a folyamat kezdetén eldobják, majd a végén újraépítik (újraindexelés)

Nézetek a relációs adatbázisban

- Ahelyett, hogy állandóan bonyolult lekérdezéseket írjunk a táblákból, létrehozhatunk egy nézetet („view”) amelyben egy lekérdezést definiálunk és névvel látunk el
- Később ebből a nézetből tudunk majd lekérdezni, és így a komplex SELECT két részre bomlik, és ezzel egyszerűsül a munkánk
- A nézet nem tárol adatot, csupán az eredményt létrehozandó lekérdezést, azaz a nézet definicióját
- A nézet csupán meta-adat
- A nézet elsősorban megkönnyíti (leegyszerűsíti) a lekérdező munkáját
- Lehet még jogosultságkezelési célja is
- A DML műveletek nem mindig megengedettek nézeteken keresztül

Egy egyszerű példa a nézetről

- Először létrehozzuk a nézetet:

```
CREATE VIEW v_preferred_customers AS
  SELECT *
  FROM   customers
  WHERE  credit_rating in ('GOOD','EXCELLENT');
View created.
```

- Később lekérdezhetünk a nézetből:

```
SELECT   customer_id ,customer_name,city
  FROM   v_preferred_customers
  WHERE  city = 'BUDAPEST';

15 rows selected.
```

- Azért a nézetek többnyire jóval bonyolultabbak ennél

Nézet, materializált nézet, vagy tábla?

- Hozzunk létre egy nézetet:

```
CREATE VIEW v_preferred_customers AS
  SELECT *
  FROM   customers
  WHERE  credit_rating in ('GOOD','EXCELLENT');
View created.
```

- Ugyanez tábla formájában:

```
CREATE TABLE preferred_customers AS
  SELECT *
  FROM   customers
  WHERE  credit_rating in ('GOOD','EXCELLENT');
Table created.
```

- És végül materializált nézetként:

```
CREATE MATERIALIZED VIEW mv_preferred_customers AS
  SELECT *
  FROM   customers
  WHERE  credit_rating in ('GOOD','EXCELLENT');
Materialized view created.
```

Amit a kényszerek nem tudnak, de hasznos lehet

- Gyakran lenne szükségünk a már korábban tárgyalt kényszerek mellett azoknál bonyolultabbakat is alkalmazni. Pl. ilyen a következő: „a dolgozó fizetése nem lehet több, mint a főnökének a fizetése”.
- Esetleg olyan „kényszereket” alkalmazni, amelyek nem tartósan állnak fenn, csupán az adatok bevitele pillanatában érvényesek. Pl. ilyen az, hogy adatbevitelkor a WORKERS tábla START_DATE oszlopa nem lehet múltbeli dátum, tehát a pillanatnyi dátumnál kisebb.
- Ezeket kénytelenek leszünk valami más módon biztosítani. Ha nem lehet deklaratív szabályokkal, akkor programkóddal tesszük majd ezt. Erre igen alkalmasak egyes adatbáziskezelő rendszerekben az adatbázis triggerek.

Adatokon kívül kódot is tartalmazhat egy adatbázis

- Ha csak adatok „perzisztálására” használjuk az adatbázist, és kódot ott nem tárolunk, akkor a teljes program-logika „messze van” az adatoktól. Ez gyakran nem célszerű.
- Praktikus és hatékony az adatokat még ott az adatbázisban „feldolgozni”, azok átmozgatása nélkül
- Ezt adatbázisban tárolt eljárásokkal és függvényekkel tehetjük meg
- Az objektumorientált programozás szerint az objektum-típus attribútumokból és metódusokból áll
- A tábláinkban az oszlopok játsszák az attribútumok szerepét. De akkor a metódusok meg programkódok, és ezeket is jó lenne ott tárolni.

Procedurális kód tárolása egy adatbázisban

- Ehhez kell egy procedurális nyelv (mert az SQL nem procedurális). Ilyen nyelv lehet a Java, vagy a gyártók előállhatnak saját nyelvvel is.
- Az Oracle adatbázis esetén ilyen nyelv a PL/SQL
- PL/SQL-ben nyelven tárolt eljárásokat, függvényeket, csomagokat (package), objektumorientált metódusokat és végül adatbázis triggereket írhatunk.
- Ezek szerint egy Oracle adatbáziskezelő rendszer nemcsak az SQL, hanem a Java és a PL/SQL nyelvet is érti. SQL, Java és PL/SQL motorral (futtató rendszerrel) rendelkezik. Ez persze nem kötelező, csupán nagyon előnyös tulajdonság.

Példa: egy egyszerű PL/SQL függvény

- Először létrehozzuk a függvényt, majd használjuk azt:

```
CREATE OR REPLACE FUNCTION worker_salary(p_worker NUMBER)
RETURN number IS
Result number;
BEGIN
  SELECT salary INTO result   FROM workers
  WHERE worker_id=p_worker;
  RETURN  result;
END worker_salary;
/
SELECT salary, worker_salary(worker_id)
FROM workers;
```

	SALARY	WORKER_SALARY(WORKER_ID)
1	8300	8300
2	24000	24000
3	17000	17000
4	17000	17000
5	9000	9000
6	6000	6000

Még egy példa: egy utasítás szintű trigger

```
CREATE OR REPLACE TRIGGER managers
  AFTER INSERT OR UPDATE OR DELETE ON workers
DECLARE
  counter NUMBER;
BEGIN
  SELECT COUNT(*) INTO counter
  FROM workers
  WHERE  UPPER(position_id)='ADMIN_PRES';
    IF counter>1 THEN
  RAISE_APPLICATION_ERROR(-20700,'Only one president allowed');
  END IF;
  SELECT MAX(COUNT(*)) INTO counter
  FROM workers
  GROUP BY manager_id;
  IF counter > 15 THEN
  RAISE_APPLICATION_ERROR(-20701,'Only 15 workers allowed');
  END IF;
END managers;
/
```

A triggerek tesztelése (kipróbálása)

```
UPDATE workers SET position_id = 'ADMIN_PRES' WHERE salary>15000;
```

```
Error starting at line : 17 in command -
UPDATE workers SET position_id='ADMIN_PRES'
WHERE salary>15000
Error report -
SQL Error: ORA-20700: Only one president allowed
ORA-06512: at "LEARNING.MANAGERS", line 7
ORA-04088: error during execution of trigger 'LEARNING.MANAGERS'
```

```
UPDATE workers SET manager_id = 101 WHERE salary>5000;
```

```
Error starting at line : 17 in command -
UPDATE workers SET manager_id=101 WHERE salary>5000
Error report -
SQL Error: ORA-20701: Only 15 workers allowed
ORA-06512: at "LEARNING.MANAGERS", line 11
ORA-04088: error during execution of trigger 'LEARNING.MANAGERS'
```

Mindenféle más objektum is található egy adatbázisban

- Az adatbáziskezelő rendszerek általában még tucatnyi egyéb objektumtípust támogatnak. De ezek erősen gyártófüggőek, és nem szabványosak
- Ezeket is az SQL valamiféle „kiterjesztésével” hozzuk létre (CREATE) , dobjuk el (DROP) vagy módosítjuk (ALTER)
- Ilyenek pl. a:
  - Szinoníma
  - Adatbázis link
  - Kontextus (ritkaság)
  - Szerepkör (nem általános)
  - …