Indexek és Teljesítményoptimalizálás

Tanulmány az Adatbázisok haladóknak c. tárgyhoz

Ableda Péter

1. évf. MSc Mérnök informatika szak

2012/2013 tanév I. félév

Tartalomjegyzék

1. Bevezetés

A relációs adatbázisok alkalmazása a korábbi megoldásokhoz képest jelentős teljesítménybeli visszaesést jelentett. Az adatbázis piac akkori három meghatározó szereplője különböző megoldásokat dolgozott ki a hatékonyság növelésére.
Az **IBM** a clustering technikában, az **Ingress** a hashing-ben az **Oracle** pedig az indexekben látta a megoldást.
Jelen tanulmányban az Oracle indexelési technikáit fogom bemutatni, valamint az általuk elérhető hatékonyságbeli növekedéssel foglalkozom.

1.1 Előzmények

A mérnök informatikus szak BSc képzésének Adatbázisok[1] című kötelező tárgyában már megismerkedhettünk az alapvető indexelési technikákkal, és azok jelentőségével. A tárgyban részletesen foglalkoztunk a B*-fa megvalósításával, így ebben a dokumentumban ennek részleteivel nem foglalkozom, a meglévő ismereteket fogom kiegészíteni az Oracle fejlettebb megoldásaival.

2. Fogalmak

Szelektivitás (Selectivity): A szelektivitás mindig szűrőfeltétel(ek)hez kötődik, és azt mutatja meg, hogy sorok egy halmazából hány felel meg a feltételnek.

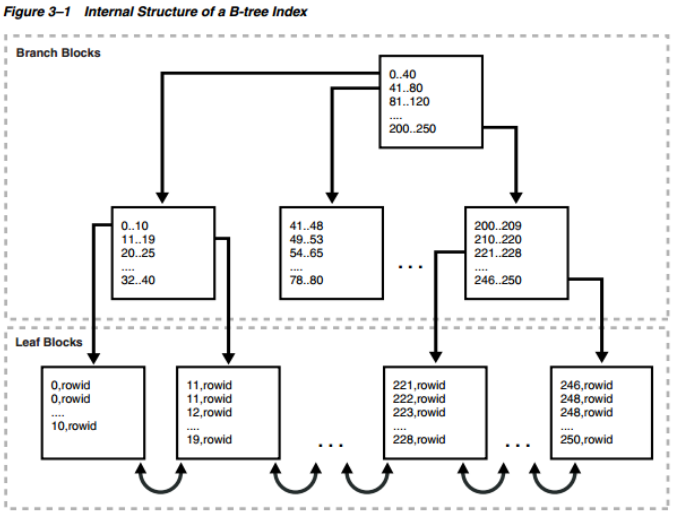
Számosság (Cardinality): A számosság alatt a halmazban található elemek számát értjük. Beszélhetünk tábla számosságáról (ez a sorok számát jelenti), vagy oszlopok számosságáról, (ekkor az oszlopban előforduló különböző elemek számára gondolunk).

[1] https://www.vik.bme.hu/kepzes/targyak/VITMA311/

3. Indexek típusai

3.1 Oracle B-Fa index*

Az Oracle leggyakrabban használt indexelési technikája a B*-fa index.



1. ábra Oracle B*-fa felépítése

A fenti ábrán egy Oracle B*-fa látható. A fa egy gyökérelemmel (*root*) rendelkezik, ami alatt több szinten, köztes (*branch*) csomópontok találhatók, legalul pedig levelek (*leaf*) helyezkednek el. Az ábrán az egyes csomópontok 1-1 blokkot jelölnek. A gyökér és a köztes csomópontok azonos felépítésűek, kulcs-mutató párokból állnak. A levelekben kulcs, *rowid* (sorazonosító) párok találhatók, alapesetben a tábla minden sorára egy ilyen pointer mutat.

Többféle szempont szerint különböztethetjük meg az indexeket. A továbbiakban néhány ilyen csoportosítást ismertetek.
Megkülönböztetünk egyedi (unique) és nem egyedi (non-unique) indexeket:

Unique: Az egyedi indexek garantálják, hogy nincs két azonos érték az adott oszlopban. A unique indexekben egyetlen rowid tartozik minden adatértékhez.

Non-unique: A nem egyedi indexek megengedik a duplikált értékeket az indexelt oszlopokban. Pl.: az Employees tábla First_name oszlopa.

Megkülönböztetünk egyoszlopos (single) és többoszlops (concatenated) indexeket:

Single: Egyoszlopos indexek, minden Primarykey és Unique oszlopra automatikusan létrejönnek.

Concatenated: Többoszlopos, vagy kompozit indexeknek is nevezik őket. Alkalmazásuk során figyelni kell, hogy az oszlopokat milyen sorrendben tároljuk az indexben. A lekérdezések csak akkor tudják használni a kompozit indexeket, ha az első néhány oszlop szerepel az utasítás (select, update, delete) where feltételében.

A null értékek nem kerülnek be az adatbázisba. Ennek előnye, hogy nagy táblák esetén az olyan oszlopokra, amelyekben sok a null elem, kisméretű index tartozhat, hátránya, hogy a NOT NULL kényszert nem tartalmazó oszlopokra létrehozott indexek nem használhatók a sorok számának meghatározásához.

3.2 Reverse key

Speciális, tükrözött kulcsú indexeket hozhatunk létre a Reverse key módszerrel,
amely egy speciális problémára, az ún. jobb oldali index levélblokk ütközésre ((*rightmost index leafblock contention*") nyújt megoldást. Ez akkor fordul elő, ha nagyon nagy számban szúrunk be új

elemeket az indexbe, és az elemek értéke monoton nő. Ekkor minden új elem tipikusan ugyanabba a blokkba fog kerülni, és konkurenciaproblémák jelentkeznek.
A megoldás, a kulcsok tükrözött tárolása, tehát egy 15, 16, 17 egymás utáni sorozatot 51, 61, 71 szorszámként fogunk eltárolni. Ekkor nem minden elem fog ugyanabba a blokkba kerülni, így elkerülhetjük az ütközést. A megoldás hátránya, hogy a tükrözött kulcsú indexek csak egyenlőségvizsgálatra használhatók.

3.3 Function based index

A függvény alapú (function based) indexek esetén nem az oszlop elemeit, hanem valamilyen függvény vagy kifejezés eredményét tároljuk.Ennek két tipikus alkalmazását az alábbi két példa szemlélteti.

```
SELECT *
  FROM employees
  WHERE UPPER(first_name) = 'AUDREY'

SELECT employee_id, 12*salary*commission_pct
  FROM employees
  WHERE (12 * salary * commission_pct) < 30000
```

3.4 Bitmap indexek

Tábla	
Név	**Nem**
Boszorka	lány
Hapci	fiú
Hófehérke	lány
Kuka	fiú
Morgó	fiú
Szende	fiú
Szundi	fiú
Tudor	fiú
Vidor	fiú

Bitmap	
fiú	**lány**
0	1
1	0
0	1
1	0
1	0
1	0
1	0
1	0
1	0

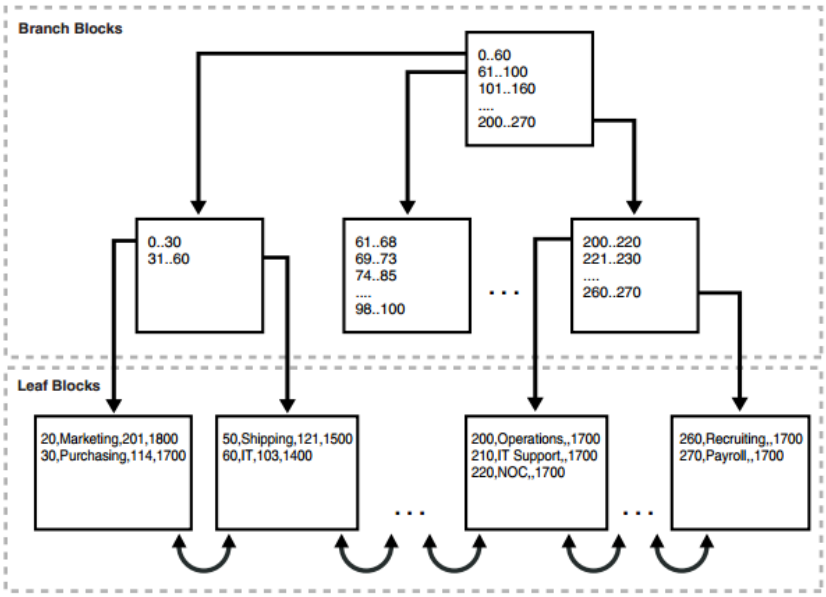
2. ábra Bitmap index használata

A bitmap indexekről bővebb leírás a korábbi előadások kidolgozott jegyzeteiben található. A továbbiakban a gyakorlati információk olvashatók.
Bitmap indexeket adattárházak vagy adatpiacok alacsony kardinalitású oszlopaira érdemes létrehozni. A bitmap index használhatóságát nagyban rontja, ha előfordulnak UPDATE és DELETE utasítások is, hiszen az index csak blokk szinten lockolható. A gyakorlatban a bitmap indexek frissítésére a Drop-Create „módszert” alkalmazzák, azaz az adatok betöltése előtt az indexet eldobják, majd újra létrehozzák őket. Így az adatok beszúrása is gyorsan végbemehet, és az indexek mérete se nő meg jelentősen a sok inaktív bejegyzés miatt.

3.5 Index Organized table

A hagyományos táblákban az adatok rendezetlenül vannak tárolva, ezzel szemben az index szervezésű táblák (index organized table) az elsődleges kulcsuk szerint rendezett B-fában tárolják az adatokat.

Figure 3–3 Index-Organized Table



3. ábra Példa Oracle index szervezésű tábla szerkezetére

A megoldás előnye, hogy az elsődleges kulcs szerinti keresés hatékonyabb, hiszen nincs szükség az indexben végrehajtott kereséshez járulékos blokkok elérésére. Az össz tárolási költség is alacsonyabb, hiszen nincs szükség az elsődleges kulcsra épített külön indexre.
A megoldás hátránya, hogy a másodlagos indexek használata lassabb, mint egyszerű táblák esetében, és hosszabb sorok esetén az adatbázis blokkokba csak kevés adat fér, így az indexfa nagyon magasra nőhet. Utóbbi problémára jelent megoldást az overflow technika, amelynek segítségével az index szervezésű táblák ritkán használt oszlopait az index blokkokon kívül tárolhatjuk, így csökkentve az indexb rekordjainak méretét.

3.6 Domain index

Az application domain indexek az alkalmazásokhoz szabott specifikus indexek. Az adatbázis indexelése ilyen módon kiterjeszthető komplex adattípusokra, például dokumentumokra, képekre, videókra is.
Az indexek működésének szabályozása (tartalmuk és struktúrájuk meghatározása) a cartridge szoftverrel történik. Az adatbázis az alkalmazással együttműködve építi, tartja karban, és használja keresésre a domain indexeket. Az index struktúrát egy különálló fájlban, vagy az adatbázisban tudjuk tárolni index szervezésű táblaként.

4. Lekérdezések optimalizálása

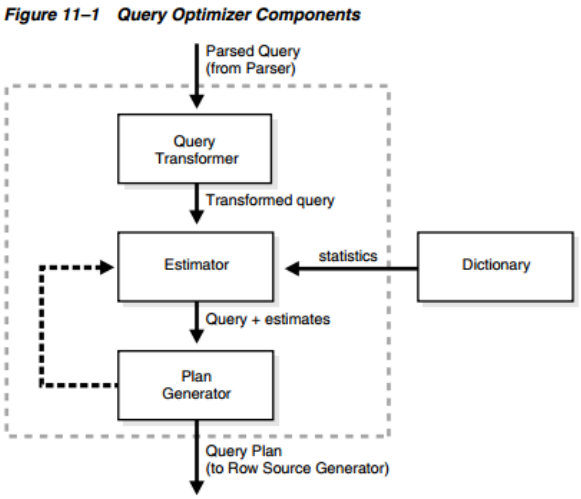
SQL lekérdezések írásakor deklaratív módon fogalmazzuk meg a kéréseinket. Ezeket a kéréseket az adatbázis átalakítja procedurális utasítások sorozatára. A **Query Optimizer** feladata a legjobb műveletsor (végrehajtási terv) előállítása a lekérdezés alapján.

A Query Optimizer a következő komponensekből áll:

QueryTransformer: Feladata a lekérdezések átalakítása, hogy az adatbázis végül hatékonyan tudja végrehajtani a kéréseinket.

Estimator: Mérőszámokat generál, amelyek segítségével az Optimizer meg tudja becsülni a végrehajtási terv egészére vonatkozó költséget.

PlanGenerator: Számos végrehajtási tervet generál, amelyek közül az Optimizer kiválasztja a legjobb megoldást, amit végre fogunk hajtani.



4. ábra A Query Optimizer működése

Az lekérdezés optimalizálásának eredménye egy **végrehajtási terv**, ami alapján az adatbázis dolgozni fog.

A végrehajtási terv lépések sorozata, amelyben minden lépés egy-egy műveletet jelent.

Például:

```
SELECTfirst_name, last_name, salary
  FROMemployees
  WHERE email IS NOT NULL
    AND salary> 500
  ORDER BYsalary;
```

```
ExecutionPlan
----------------------------------------------------------
Planhashvalue: 3447538987

-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   107 |  2889 |     4  (25)| 00:00:01 |
|   1 |  SORT ORDER BY     |           |   107 |  2889 |     4  (25)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES |   107 |  2889 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

PredicateInformation (identifiedbyoperationid):
---------------------------------------------------

   2 - filter("SALARY">500)
```

A végrehajtási tervet mindig belülről kifelé kell olvasni, a DBMS a legjobban behúzott műveleteket hajtja végre legelőször, majd azok eredményeit felhasználva halad kifelé.
A példán látható, hogy a lekérdezés végrehajtása egy TABLE ACCESS FULL művelettel kezdődik, azaz a lekérdezés végrehajtásához az EMPLOYEES tábla teljes beolvasására szükség van. A beolvasás kiegészül egy szűrőfeltétellel, azaz csak azokat a sorokat fogja visszaadni, ahol a Salary oszlop értéke 500-nál nagyobb. A kapott eredményhalmazon rendezést végzünk.

A táblázatban becsült adatokat láthatunk arra vonatkozóan hogy hány sort fog eredményezni a művelet (*Rows*), mekkora méretű lesz a művelet eredményhalmaza (*Bytes*), valamint mennyi a művelet költsége (*Cost*) és ideje (*Time*).

A végrehajtási tervekkel és a lekérdezés optimalizálással bővebben [2] foglalkozik.

4.1 Hozzáférési módok

Minden objektum esetében, amit felhasználunk a végrehajtási tervben, meg kell határozni a hozzáférés módját. Ez megmutatja, hogyan tudjuk elérni az adatokat az adatbázisban. A hozzáférés módjának meghatározásánál figyelembe kell venni a FROM utasításrészben található objektumokat, és a WHERE utasításrészben található kifejezéseket.
A következő szakaszokban az egyes hozzáférési módokat részletezzük.

4.2 Full table scan

A teljes táblaolvasás (full table scan) hozzáférés során beolvassuk a tábla összes sorát, majd kiszűrjük azokat, amelyek megfelelnek a kiválasztási feltételeknek.
A teljes táblaolvasás a legköltségesebb hozzáférési mód. Ezt csak akkor választja az optimalizáló, ha nincs más lehetősége, mert:

- nincs elérhető index a táblán;
- a táblában lévő adatok nagy részére szükség lesz (kis szelektivitás);
- túl kicsi a tábla ahhoz, hogy megérje indexhez nyúlni.

Például:

```
SELECT *
  FROM employees
  WHERElast_name NOT LIKE 'D%';
```

```
ExecutionPlan
----------------------------------------------------------
Planhashvalue: 1445457117

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   102 |  7038 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |   102 |  7038 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

PredicateInformation (identifiedbyoperationid):
---------------------------------------------------

   1 - filter("LAST_NAME" NOT LIKE 'D%')
```

A továbbiakban az indexhozzáférési műveletekkel foglalkozunk.

4.3 Index scan

Egy index hozzáférés során az adatbázis bejárja az indexet az indexelt oszlopok alapján, majd visszaadja azon sorazonosítók halmazát, amelyek megfelelnek a keresési feltételeknek.

Full Index Scan

A full index scan során a teljes indexet sorban végigolvassuk (balról a jobb széléig). A full index scan használható, ha a lekérdezés where feltételében az index egy oszlopára hivatkozunk, és néhány esetben, ha nincs szűrő feltétel.
Ennek a hozzáférési módnak az előnye, hogy az adatokat nem kell rendezni, hiszen azokat az index rendezetten tárolja.

FastFull Index Scan

A fast full index scan hozzáférés során az adatbázis full index scan-t hajt végre. Az adatbázis az adatokat az indexből olvassa ki a táblához való hozzáférés nélkül.
Ez a full table scan alternatívája. Akkor használható, ha az index a lekérdezés minden elemét tartalmazza, és legalább egy indexben szereplő oszlop NOT NULL kényszerrel van ellátva.
Ennek a megoldásnak az előnye, hogy nem kell teljes sorokat beolvasni.

Index Unique Scan

Az index unique scan hozzáférési módot akkor alkalmazzuk, ha legfeljebb egy sorazonosítót kell visszakapnunk. Akkor lehet ezt a műveletet használni, ha garantált az értékek különbözősége, ami a PRIMARY KEY és a UNIQUE oszlopokra teljesül.

Például:

```
SELECT *
  FROM employees
  WHERE employee_id = 5;
```

```
ExecutionPlan
----------------------------------------------------------
Planhashvalue: 1833546154

-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |    69 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    69 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

PredicateInformation (identifiedbyoperationid):
---------------------------------------------------

   2 - access("EMPLOYEE_ID"=5)
```

Index Range Scan

Az index range scan hozzáférés esetében nincs szükség az értékek egyediségére. Ebben a hozzáférési módban, a feltételben szereplő operátor lehet például egyenlőség, kisebb, nagyobb, de bizonyos esetekben LIKE is.
Az eredményhalmaz előállítása során először a gyökérből kiindulva eljutunk a keresett levélig, majd a leveleken az egymásra mutató pointereket felhasználva lépkedünk végig az indexen.

Például:

```
SELECT *
  FROM employees
  WHERE last_name LIKE 'A%';
```

```
ExecutionPlan
----------------------------------------------------------
Planhashvalue: 2077747057

-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |             |     3 |   207 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |     3 |   207 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP_NAME_IX |     3 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

PredicateInformation (identifiedbyoperationid):
---------------------------------------------------

   2 - access("LAST_NAME" LIKE 'A%')
filter("LAST_NAME" LIKE 'A%')
```

Index Skip Scan

Az index skip scan akkor használható, ha csak olyan összetett indexünk van az adott táblán, amelynek vezető oszlopait nem tartalmazza a lekérdezésünk. Az adatok kigyűjtése akkor lesz hatékony, ha a vezető oszlop kardinalitása alacsony.

Például:

Elérhető egy (*cust_gender*, *cust_email*) oszlopokból álló compositeindex.

```
SELECT *
  FROM sh.customers
  WHERE cust_email = 'Abbey@company.com';
```

Az indexben található *cust_gender* értéke M (Male) vagy F (Female). Ekkor megtehetjük, hogy először a férfiak között keressük az adott email címmel rendelkező embereket, majd a nők között.

Ez tulajdonképpen megegyezik azzal, mintha a lekérdezésünket átírnánk a következő alakra:

```
    SELECT *
      FROM sh.customers
      WHERE cust_gender = 'F'     AND cust_email =
    'Abbey@company.com'UNION ALLSELECT *
      FROM sh.customers
      WHERE cust_gender = 'M'
        AND cust_email = 'Abbey@company.com'
```

```
ExecutionPlan
----------------------------------------------------------
Planhashvalue: 1287876719

-------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                       |    33 |  5973 |    10   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| CUSTOMERS             |    33 |  5973 |    10   (0)| 00:00:01 |
|*  2 |   INDEX SKIP SCAN            | CUSTOMERS_GENDER_EMAIL |    33 |       |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

PredicateInformation (identifiedbyoperationid):
---------------------------------------------------

   2 - access("CUST_EMAIL"='Abbey@company.com')
filter("CUST_EMAIL"='Abbey@company.com')
```

Irodalom

[1] LanceAshdown, Tom Kyte: „Oracle® DatabaseConcepts 11g Release 2 (11.2)”, 2009 http://docs.oracle.com/cd/E14072_01/server.112/e10713.pdf

[2] Immanuel Chan, LanceAshdown: „Oracle® Database Performance TuningGuide 11g Release 2 (11.2)”, 2011 http://download.oracle.com/docs/cd/E11882_01/server.112/e16638.pdf

[3] Paul Lane: „Oracle® Data WarehousingGuide 11g Release 2 (11.2)”, 2010, http://download.oracle.com/docs/cd/E11882_01/server.112/e25554.pdf