LEKÉRDEZÉSEK OPTIMALIZÁLÁSA

Készítette:
Salamon András

2002.

Tartalomjegyzék

Kivonat

A dolgozat az SQL lekérdezések optimalizálásáról szól. Bár sokféle adatbázis-szerver létezik, a vizsgálathoz Oracle szervert választottunk. A leírt módszerek egy része használható más adatbázis-szerver esetén is.

Az optimalizálásnál feltételezzük, hogy egy működő adatbázis lekérdezéseit kell optimalizálni, vagyis igyekszünk tartózkodni olyan módosításoktól, melyek miatt a többi lekérdezést is módosítani kellene.

Az adatok elérését jelentősen befolyásolják az indexek, ezért először az Oracle adatbázisszerverben használható indexeket ismertetjük.

Ezután a beépített optimalizáló működési elve következik. A fejezet tartalmazza mind a szabály, mind a költség alapú optimalizáló működésének leírását. Ezek működési elvét azért fontos megismerni, mert az általunk használt lekérdezéseket az Oracle ezek segítségével próbálja optimalizálni.

Ezután általános optimalizálási módszerek, ötletek következnek.

Végül gyakorlati példákon keresztül mutatjuk be a lekérdezések optimalizálását. A példák nagy része mesterséges, kisebb része valós adatbázis valós lekérdezése.

1. fejezet

Indexek

Ez a fejezet az Oracle adatbázis-szerverben használható indexeket mutatja be. Indexekről további információ a [1, 153. oldal] könyvben található. Oracle indexekről részletesebb információ a [3, 10-23. rész] dokumentumban.

Az indexek alapvető célja, hogy felgyorsítsák a táblákban lévő adatok elérését. Egy táblához több index is készíthető, elegendő ha az index oszlopai, vagy akár az oszlopok sorrendje különböző.

Az indexek hiánya vagy jelenléte nem befolyásolja az SQL utasításaink tartalmát. Ez azt jelenti, hogy az index csak a végrehajtás sebességét módosítja, nem szükséges átírni az utasításainkat egy index létrehozása után.

Az indexek a lekérdezések sebességét javítják, az adatbázismódosítások sebességét azonban lassítják, hiszen az index adatait is frissítenie kell az Oracle-nek. Emiatt gondosan el kell dönteni milyen indexeket használunk, hiszen a felesleges indexek túlzottan lelassíthatják az adatbázisszerver működését. Más indexelési stratégiát kell követnünk a gyakran olvasott ritkán írt, és a ritkán olvasott gyakran írt tábláknál.

Az indexek logikailag és fizikailag is elkülönülnek a táblától, és külön tárhelyet igényelnek. Ha a táblát és az indexet külön lemezen tároljuk, akkor a párhuzamos olvasás miatt javul a lekérdezések teljesítménye. Amennyiben mégis együtt szeretnénk tárolni a tábla és az index adatait, akkor index-szervezett táblákat (1.4. rész) használhatunk.

Oracle többféle indexet tud kezelni:

- B-fa indexek (1.2.1. rész)
- B-fa klaszter indexek (1.3.1. rész)
- Tördelő klaszter indexek (1.3.2. rész)
- Fordított kulcsú indexek (1.2.2. rész)
- Bittérkép indexek (1.2.3. rész)

1.1. Indexek jellemzői

1.1.1. Egyedi indexek

Az indexek két fajtáját különböztethetjük meg, az egyedi (unique) és a nem egyedi (nonunique) indexeket. Egyedi indexnél nincs két olyan sor a táblában, amelyeknél minden indexoszlop tartalma megegyezik. Nem egyedi indexeknél nincs ilyen megszorítás.

Nem javasolt explicit módon egyedi indexeket használni. Ha egyediségre van szükségünk, akkor a tábla definíciójakor használjuk a `UNIQUE` megszorítást az oszlopokra. Ebben az esetben az adatbázis-szerver automatikusan egyedi indexet hoz létre.

1.1.2. Összetett indexek

Összetettnek (composite) nevezünk egy indexet, ha a tábla több oszlopát tartalmazza. Az oszlopok tetszőleges sorrendben előfordulhatnak, és nem szükséges, hogy szomszédosak legyenek. Egy ilyen index akkor gyorsítja fel a lekérdezést ha a `WHERE` feltételben az indexoszlopok egy prefixének *összes* mezője szerepel. Vagyis az összetett indexekben a mezők sorrendje fontos, és bizonyos esetekben érdemes olyan összetett indexeket használni, amelyeknél az oszlopoknak csak a sorrendje különbözik.

1.1.3. Függvény-alapú indexek

Definiálhatunk indexet egy a tábla mezőit használó függvényen. A használt függvénynek determinisztikusnak kell lennie. Ezek az indexek akkor hasznosak, ha a `WHERE` feltételben az indexet definiáló függvény szerepel. Például a következő utasítással létrehozott indexet:

```
CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);
```

felhasználhatja az Oracle a következő lekérdezésnél:

```
SELECT a FROM table_1 WHERE a + b * (c - 1) < 100;
```

Egy függvény-alapú indexet vizsgáló tesztet tartalmaz a 4.5. rész.

1.1.4. Kulcs tömörítés

Kulcstömörítés során a kulcsot két részre bontják, a prefix és a suffix részre. A tömörítés legfőbb előnye, hogy a kulcsok azonos prefix részeit elég egyszer eltárolni, így kevesebb tárhelyre van szükség. A kulcsok kiolvasása némileg bonyolultabb.

Tipikus esete a kulcs tömörítésnek amikor a kulcs *(elem, időpont)* alakú. Az egyediséget az időpont eltárolása biztosítja, enélkül rengeteg elem megegyezne. Ha prefixnek az *elem*, suffixnak a *időpont* mezőt választjuk, akkor az azonos elemeket elég egyszer eltárolni.

1.2. Indexek fajtái

1.2.1. B-fa indexek

A legelterjedtebb index fajtánál a kulcsokat egy B-fában tárolja a szerver. A sorokat a B-fában (bővebben: [2, 494. oldal], vagy [1, 184.olda]) tárolt *rowid* alapján azonosítja a rendszer.

1.2.2. Fordított kulcsú indexek

Fordított kulcsú indexeknél az indexoszlopokat fordított bájtsorrendben tárolja az adatbázisszerver. Elosztott rendszereknél hasznos, amennyiben az egymást követő adatmódosító műveleteknél hasonló indexű elemeket módosítanak. Fordított kulcsú indexeket használva elveszítjük azonban a range scan (2.3.2. rész) lehetőségét.

1.2.3. Bittérkép (bitmap) indexek

Bittérkép indexek ([1, 253. oldal]) használatánál minden kulcsértékhez külön bitvektort tárol el a szerver. A bitvektor bitjei a tábla egyes sorainak felelnek meg. Ha a bit értéke 0, akkor a sor nem tartalmazza, ha 1 akkor tartalmazza a kulcsértéket.

Mivel minden kulcsértékhez külön bitvektort tárol a szerver, bittérkép indexeket akkor érdemes használni, ha a lehetséges kulcsértékek száma alacsony. Tipikus esete ennek, amikor egy embernek a nemét vagy titulusát tároljuk. Az első esetben 2, a második esetben kb. 5 különböző kulcsérték létezik.

Ha a `WHERE` feltételben több olyan oszlopra hivatkozunk, melyekhez létezik bittérkép index, akkor bitműveletek segítségével gyorsan kiértékelhető a feltétel teljesülése. Bittérkép indexeket mutatja be a 4.4. rész.

1.3. Klaszterek (Clusters)

Egy alternatív módszer a táblák adatainak tárolására a klaszterek használata. Egy klaszter táblák csoportja, melyek közös adatcsoportban találhatóak, mivel közös mezőket tartalmaznak, melyeket gyakran használunk együtt. A közös mezőt (vagy mezőket) klaszter kulcsnak nevezzük, az egyes kulcsokhoz tartozó sorokat (melyek nemcsak egy táblában találhatóak) azonos helyen tárolja a rendszer.

A következő előnyei vannak a klasztereknek:

- Hatékonyabbak a lekérdezések, ha a lekérdezésben az érintett táblákat összekapcsoljuk.
- A klaszter kulcsot elég egyszer tárolni, függetlenül az érintett sorok számától, így tárterületet spórolunk. (Helyesen megválasztott klaszter méretnél)

A következő hátrányai vannak a klasztereknek:

- Ha a klaszter kulcs értékét módosítjuk, akkor a sort fizikailag át kell helyeznie az adatbázis-szervernek. Olyan oszlopot tehát nem érdemes klaszter kulcsnak választani, melynek értéke gyakran módosul.
- Mivel egy adatblokkban több tábla adatait tárolja a rendszer, egy tábla összes adatának eléréséhez több adatblokkot is érinteni kell.

Hasonlóan az indexekhez a klaszterek használatakor sem kell az alkalmazásunkat módosítani, ugyanolyan SQL utasításokkal érhetjük el az adatokat, mint klaszterek használata nélkül.

1.3.1. Klaszter index

A klaszter létrehozása után létre kell még hozni a klaszter indexet is. Az indexet még a táblák adatokkal való feltöltése előtt kell létrehozni. A klaszter index törlése esetén az adatok nem érhetőek el, amíg az új indexet nem hozzuk létre.

Ezt az indexet használja a rendszer, hogy megtalálja, az adott kulcsértékhez tartozó sorok mely adatblokkban találhatóak.

1.3.2. Tördelő klaszterek

Ebben az esetben egy tördelőfüggvény ([1, 200. oldal]) határozza meg, melyik klaszterben tároljuk a sort. A függvény inputja a klaszter kulcs értéke. A tördelőfüggvény körülbelül az általunk meghatározott `HASHKEYS` különböző értéket ad vissza. Nagyobb értéket használva csökkentjük az ütközés (amikor két különböző kulcshoz azonos tördelő érték tartozik) esélyét. A tördelésről bővebben olvashatunk *Knuth* könyvében. [2, 528. oldal].

Tördelőfüggvény

Tördelőfüggvényként használhatjuk az Oracle beépített függvényét, vagy magunk is definiálhatunk saját tördelőfüggvényt.

A beépített tördelőfüggvény használhatjuk egyszerű és összetett indexnél is. A `LONG` és `LONG RAW` típusokat leszámítva minden típusra működik.

Ha a klaszter kulcs értékek egyenletesen oszlanak el, akkor a kulcs értékét is használhatjuk tördelő értéknek. Természetesen ha az érték nagyobb mint `HASHKEYS`, akkor a `HASHKEYS` -szel való osztás maradékát veszi a rendszer.

Ha a klaszter kulcsok nincsenek kellően elosztva, és a beépített tördelőfüggvénnyel sem vagyunk megelégedve, akkor saját magunk is definiálhatunk tördelőfüggvényt.

1.4. Index-szervezett táblák

Index-szervezett tábláknál a tábla tartalmát az indexben tárolja a szerver. A B-fában a sor azonosítója helyett a sor tartalma kerül eltárolásra.

Az ilyen táblákat a többi táblához hasonló módon kezelhetjük, de a tábla adatain végzett módosításokat az index módosítása helyettesíti.

A fő előnye az index-szervezett tábláknak az adatok gyorsabb elérése. A gyorsulás akkor jelentkezik ha a kulcs alapján olvassuk ki a sorokat. További előny a valamivel kisebb helyfoglalás. Ugyanakkor számos dolgot nem használhatunk az ilyen táblákkal (pl. UNIQUE constraint, LONG adattípus).

2. fejezet

Az optimalizáló

2.1. SQL feldolgozás

Az SQL feldolgozás szerkezetét a 2.1. ábra mutatja. A feldolgozása a következő részekből áll:

- Elemző (Parser)

  Két feladata van: szintaktikai és szemantikai ellenőrzés.

- Optimalizáló

  Az SQL feldolgozás lelke, két fő optimalizálási módszer van, a szabály alapú (RBO) (2.4. rész) és a költség alapú (CBO) (2.3. rész).

- Végrehajtási terv generátor (Row source generator)

  Az optimalizáló szerinti optimális terv alapján SQL végrehajtási tervet generál. (A végrehajtási tervet az `EXPLAIN PLAN` utasítás segítségével ellenőrizhetjük.)

- SQL végrehajtó

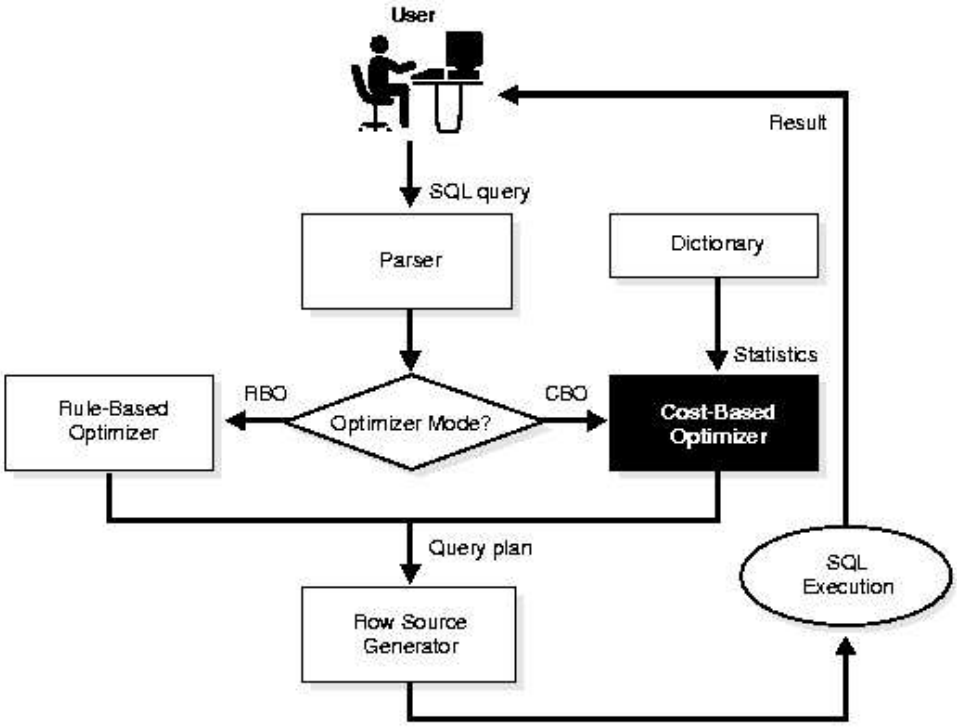
  Ez a komponens adja vissza a lekérdezés eredményét.

2.2. Az optimalizáló célja

Alapvetően két célja lehet az optimalizálónak:

- Teljes idő minimalizálása (best throughput)

  Alapértelmezés szerint a cél az, hogy a lekérdezés teljes végrehajtásához szükséges erőforrások mennyisége minimális legyen.



2.1. ábra. SQL feldolgozás (forrás: [4])

Ha a lekérdezés teljes eredményére szükségünk van a további munkához, akkor ezt a módot érdemes választani.

- A válaszidő minimalizálása

  Ebben az esetben a cél az, hogy a válasz első sorának előállításához szükséges erőforrások mennyisége minimális legyen.

  Interaktív alkalmazásokban lehet hasznos ez a megközelítés, hiszen a felhasználó már dolgozhat a kapott részeredményekkel, miközben a további eredmények még generálódnak.

Az optimalizáló viselkedését többféle módon is befolyásolhatjuk:

- `OPTIMIZER_MODE` inicializáló paraméterrel
  - `CHOOSE`

    Az optimalizáló az alapján dönt CBO és RBO között, hogy létezik-e legalább egy érintett táblához statisztika. A statisztika hiánya esetén a szabály alapú, megléte esetén a költség alapú optimalizálást választja az optimalizáló.

  - ALL_ROWS

    A statisztikák jelenlététől függetlenül CBO-t használja az optimalizáló. Az optimalizálás célja a teljes idő minimalizálása.

  - FIRST_ROWS

    A statisztikák jelenlététől függetlenül CBO-t használja az optimalizáló. Az optimalizálás célja a válaszidő minimalizálása.

  - RULE

    A statisztikák jelenlététől függetlenül RBO-t használja az optimalizáló.

- Statisztika készítésével

  A költség alapú optimalizáló sikeres működéséhez az optimalizálónak sok információra (statisztikára) van szüksége a táblákról, oszlopokról, klaszterekről, indexekről és partíciókról. Statisztikák készítésére, karbantartására a DBMS_STATS csomagot, az ANALYZE utasítást és a CREATE or ALTER INDEX utasítás COMPUTE STATISTICS részét használhatjuk. Az Oracle statisztika kezeléséről bővebb információt a [4, 8. fejezet] dokumentumban találhatunk.

- Az ALTER_SESSION utasítás OPTIMIZER_GOAL paraméterével

  A paraméter lehetséges értékei megegyeznek az OPTIMIZER_MODE paraméter lehetséges értékeivel.

- Tippek (Hints) használatával.

  A tippek lehetséges értékei megegyeznek az OPTIMIZER_MODE paraméter lehetséges értékeivel.

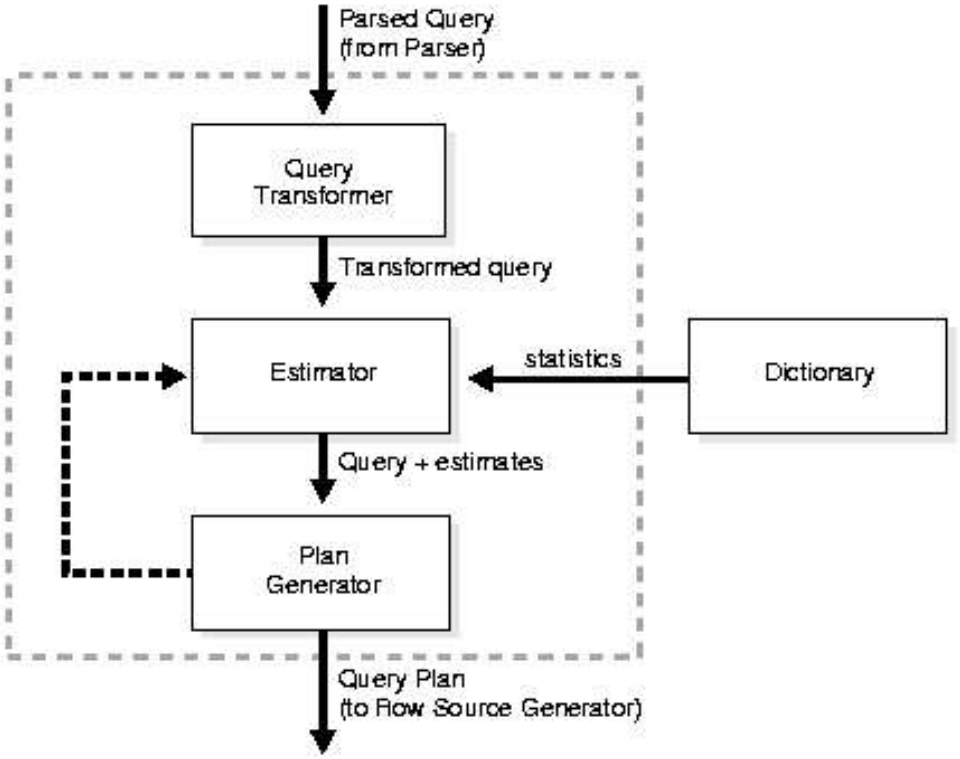
2.3. Költség alapú optimalizálás (CBO)

A költség alapú optimalizálásról bővebben a [1, 410. oldal] könyvben olvashatunk.

Speciális esetektől eltekintve mindig érdemesebb CBO-t használni mint RBO-t. A használatához azonban szükség van friss statisztikákra. Hiányzó statisztika esetén a módszer becsült statisztikát (lásd [4, 8-11. rész])) használ. Ha a becsült érték jelentősen eltér valóditól, akkor az optimalizáló által készített terv nagyon rossz is lehet.

A költség alapú optimalizálás szerkezetét a 2.2. ábra mutatja. Az optimalizálás a következő lépésekből áll:

- Több lehetséges terv generálása.
- Az egyes tervek költségének megbecslése.



2.2. ábra. CBO felépítése (forrás: [4])

A költség a felhasznált erőforrásoktól függ. Az optimalizáló figyelembe veszi például a következő erőforrásokat: (I/O, memória). Az erőforrások nagy részét az *I/O-költség dominanciája* miatt hanyagolhatjuk el (lásd: [1, 67. oldal]).

A költséges becsléséről bővebb információt találhatunk a [5] könyvben.

- A legkisebb költséggel rendelkező terv kiválasztása.

2.3.1. CBO architektúrája

CBO a következő komponensekből áll:

- Lekérdezés átalakító

  Inputja az elemző által blokkokra bontott lekérdezés. Az átalakító fő célja annak eldöntése érdemes-e a lekérdezést más alakra alakítania, olyan alakra, melyhez sikeresebb terv generálható.

  A következő transzformációk használhatóak ebben a lépésben:

  - View összevonás

    A lekérdezésben használt view-khoz kétféle módon generálhatunk tervet. Az első módszer szerint a view-kat külön blokknak tekintjük, melyet önállóan optimalizálunk. Ezt a módszert használva legtöbbször szuboptimális tervet generálhatunk.

    A második módszer szerint a view blokkját összevonjuk a view-t tartalmazó blokkal, és az így kapott blokkot optimalizáljuk.

    A lekérdezés átalakító a legtöbb esetben elvégzi az összevonást.

  - Részlekérdezés átalakítás

    A view-khoz hasonlóan a részlekérdezéshez is külön blokk tartozik, mely nehezíti az optimalizálást. A legtöbb részlekérdezést átalakíthatjuk összekapcsolássá, ezzel is elősegítve az optimalizálást.

  - Materializált view-k használata

    A materializált view olyan mint egy lekérdezés, melynek eredményét egy táblában tároljuk. Ha a felhasználó által írt lekérdezés kompatíbilis egy materializált view-val, akkor a lekérdezés olyan formára alakítható, mely használja a materializált view-t.

- Becslő (estimator)

  Három egymással összefüggő mennyiséget próbál megbecsülni:

  - Szelektivitás (Selectivity)

    A szelektivitás azt határozza meg, hogy egy szűrő hatására ez eredeti soroknak hányad része kerül kiválasztásra. Értéke 0.0 (nem lesz sor kiválasztva) és 1.0 (minden sor kiválasztásra kerül) között van.

  - Számosság (Cardinality)

    A számosság határozza meg, a sorok számát a lekérdezésben. Egy táblában ez a statisztika alapján könnyen meghatározható. Ha szűrőfeltételek is találhatóak a lekérdezésben, akkor a szűrő szelektivitás értékeit is figyelembe kell venni.

    Összekapcsolás esetén a számosságokat összeszorzódnak. Figyelembe kell azonban venni az összekapcsoló predikátum szelektivitás értékét is.

    Bővebben lásd [1, 396. oldal].

  - Költség (Cost)

    Költségen a szükséges lemezműveletek számát értjük. Ez függ egyrészt a relációk számosságától, másrészt attól, milyen módon érjük el a táblák adatait.

- Terv generáló

A tervgeneráló fő célja különböző tervek készítése, és a legkisebb költségű kiválasztása. A különböző elérési módok, összekapcsolás típusok és összekapcsolási sorrend miatt, többnyire sok különböző terv közül kell választania.

Először az allekérdezésekhez (például nem összevont view-k) készít terveket, melyeket megpróbál optimalizálni. A blokkok közül először a legbelsőkkel foglalkozik, a legkülső blokk optimalizálása az utolsó.

Mivel a lehetséges tervek száma exponenciálisan nő az összekapcsolt táblák számával, nem lehetséges az összes terv kipróbálása, az algoritmus csak a tervek egy részét dolgozza fel. A feldolgozás során az addig talált legjobb terv határozza meg, hogy az algoritmus mennyi tervet próbál ki. Nagymértékben javítja az algoritmus működését, ha a legelső kipróbált összekapcsolási sorrend sikeres, ezért ezt a sorrendet egy egyszerű heurisztika alapján próbálja meghatározni a terv generáló.

2.3.2. Elérési módok (access paths)

Az optimalizáló egyik legfontosabb feladata annak eldöntése, miként érje el az adatbázisban lévő adatokat. A következő elérési módok ismertek:

- Teljes tábla elérés (Full Table Scan)

  Ebben az esetben a tábla összes sorát olvassa az Oracle, és egyenként dönti el, megfelel-e a `WHERE` feltételnek.

- Minta tábla elérés (Sample Table Scans)

  Ez a mód véletlenszerűen olvas adatokat a táblából. Akkor használatos, ha a `FROM` után `SAMPLE` vagy `SAMPLE BLOCK` utasítás található. Az utasítások segítségével a táblából véletlenszerűen választhatunk ki sorokat.

- Tábla elérés *rowid* alapján

  A sor azonosítója alapján olvas ki bizonyos sorokat. Egy adott sor elérésének ez a leggyorsabb módja. Az azonosítókat például index scan segítségével kaphatjuk meg.

- Klaszter scan

  Azokat a sorokat adja vissza, ahol a sorok klaszter kulcs értéke azonos. Oracle először klaszter index (1.3.1. rész) segítségével meghatározza a sorok azonosítóját, majd ez alapján olvassa ki a sorokat.

- Hash scan

  Egy hash klaszterben (1.3.2. rész) tárolt sorokat olvashatunk ki a módszer segítségével. Először a hash függvény segítségével a klaszter kulcsból kiszámítja a hash értéket, majd ezután kiolvassa a sorokat.

- Index scan

  Egy index segítségével éri el az adatokat. A következő típusai vannak a módszernek:

  - Unique scan

    Egyetlen sor azonosítót ad vissza. Akkor használható, ha primary key vagy unique key biztosítja, hogy egyetlen sor teljesíti csak a feltételt.

  - Range scan

    Nulla vagy több sor azonosítót ad vissza. Akkor használható, ha nem biztosított, hogy egyetlen sor teljesíti a feltételt.

  - Full scan

    A teljes indexet végignézi. A blokkokat egyesével olvassa be.

  - Fast full scan

    Az előző egy alternatív módszere. Egyszerre több blokkot olvas, és párhuzamosítható.

  - Index összekapcsolás

    Olyan indexeket kapcsol össze, melyek együtt tartalmazzák a tábla összes szükséges mezőjét. Az összes adatot ki tudjuk tehát nyerni az indexekből, nem szükséges a táblák elérése.

  - Bittérkép

    Bittérkép indexek (1.2.3. rész) elérési módja.

2.3.3. Kibővítés

Lehetőség van arra, hogy a CBO fő komponenseinek viselkedését befolyásoljuk (bővebben lásd [4, 4-32. rész]). Lehetőségünk van például saját statisztikák készítésére, a szelektivitás és a költség számításának befolyásolására.

2.4. Szabály alapú optimalizálás (RBO)

Szabály alapú optimalizálásnak Oracle nem veszi figyelembe a statisztikákat (vagy azért mert nincsenek, vagy mert explicit módon utasítottuk erre). Helyette érdemesebb költség alapú optimalizálást használni.

Szabály alapú optimalizálás során az optimalizáló csak a lekérdezés szerkezetével, a rendelkezésre álló indexekkel foglalkozik, nem törődik a táblák tartalmával, a rendelkezésre álló statisztikákkal.

2.4.1. Elérési módok (access paths)

Egy heurisztika segítségével dönti el az Oracle, hogy melyik elérési módot használja. Többnyire az alacsonyabb rangú elérési módot választja.

1. Egy sor sorazonosító alapján (Single Row by Rowid)

   Akkor használható a `WHERE` feltétel rowid alapján azonosítja a sort, vagyis `WHERE ROWID='` *sor azonosító* `'` szerepel a lekérdezésben.

2. Egy sor klaszter összekapcsolás alapján (Single Row by Cluster Join)

   Azonos klaszterben tárolt táblák elérésére használható a mód, ha a táblákat a klaszter kulcs minden mezőjét használó egyenlőségekkel kötjük össze, és biztosítva van (egy primary vagy unique kulcs segítségével), hogy a lekérdezés csak egy sort ad vissza.

3. Egy sor tördelő klaszter kulcs alapján (Single Row by Hash Cluster Key with Unique or Primary Key)

   Egy tördelő klaszter kulcs és egy egyediséget garantáló kulcs együttes megléte esetén használható elérési mód.

4. Egy sor unique vagy primary kulcs alapján (Single Row by Unique or Primary Key)

   Akkor használható, ha a `WHERE` feltételben egy primary vagy unique kulcs összes oszlopa egyenlőséggel szerepel. További megszorítás, hogy összetett kulcsoknál az egyenlőségeket `AND` kulcsszóval kell összekötni.

5. Klasztert használó összekapcsolás (Clustered Join)

   Azonos klaszterben tárolt táblák elérésére használható a mód, ha a táblákat a klaszter kulcs minden mezőjét használó egyenlőségekkel kötjük össze.

6. Tördelő klaszter kulcs (Hash cluster key)

   Azonos tördelő klaszterben tárolt táblák elérésére használható a mód, ha a táblákat a klaszter kulcs minden mezőjét használó egyenlőségekkel kötjük össze.

7. Indexelt klaszter kulcs (Indexed cluster key)

   Azonos indexelt klaszterben tárolt táblák elérésére használható a mód, ha a táblákat a klaszter kulcs minden mezőjét használó egyenlőségekkel kötjük össze.

8. Összetett index (Composite Index)

   Akkor használható, ha a `WHERE` feltétel egy összetett index összes mezőjére egyenlőséggel hivatkozik, és az egyenlőségek `AND` operátorral vannak összekötve.

9. Nem-összetett index (Single-Column Indexes)

   Akkor használható, ha a `WHERE` feltétel egy vagy több egyetlen mezőt tartalmazó indexre hivatkozó egyenlőséget tartalmaz. Ha több indexre hivatkozunk a `WHERE` feltételben, akkor `AND` operátorral kell összekötni őket.

10. Korlátos tartomány keresés indexelt oszlopokon (Bounded Range Search on Indexed Columns)

    Akkor használható, ha egy egyszerű index oszlopát, vagy egy összetett index kezdő oszlopait használó, korlátos tartományt meghatározó feltételt tartalmaz a lekérdezés.

11. Nem korlátos tartomány keresás indexelt oszlopokon(Unbounded Range Search on Indexed Columns)

    Akkor használható, ha egy egyszerű index oszlopát, vagy egy összetett index kezdő oszlopait használó, nem korlátos tartományt meghatározó feltételt tartalmaz a lekérdezés.

12. Rendezésen alapuló összekapcsolás (Sort-Merge Join)

    Nem azonos klaszterben tárolt táblák egyik összekapcsolási módszere. (2.5. rész)

13. Indexelt oszlop maximuma vagy minimuma (MAX or MIN of Indexed Column)

    Egy `WHERE` és `GROUP BY` nélküli lekérdezésben egy egyszerű index oszlopát, vagy egy összetett index kezdő oszlopait használó maximum vagy minimum függvény esetén használható elérési mód.

14. `ORDER BY` indexelt oszlopon (`ORDER BY` on Indexed Column)

    Indexelt oszlopon értelmezett `ORDER BY` esetén használható, ha garantált, hogy legalább egy oszlop értéke nem `NULL` .

15. Teljes tábla elérés (Full Table Scan)

    Bármilyen lekérdezés esetén használható módszer. Mivel ez a legalacsonyabb rangú módszer, csak akkor választja az RBO, ha a fenti módszerek közül egyik sem használható.

2.5. Összekapcsolások optimalizálása

Az optimalizálás egyik legfontosabb lépése az összekapcsolások optimalizálása. Három egymással összefüggő döntést kell hoznia az optimalizálónak:

- Az összekapcsolandó táblákhoz elérési mód választása.

- Összekapcsolási algoritmus választása. A következő algoritmusok közül választhat a rendszer:

  – Beágyazott ciklusú (Nested Loop (NL)) összekapcsolások

    A [1, 303. oldal] könyvben részletesebben ismertetett módszer lényege, hogy a két tábla közül kiválasztjuk melyik legyen a belső és a külső, és két egymásba ágyazott ciklus segítségével az összes sorpárra megvizsgáljuk, hogy megfelel-e az összekapcsolás feltételének.

  – Rendezésen alapuló algoritmusok

    A [1, 308. oldal] könyvben részletesebben ismertetett kétmenetes algoritmus csak egyenlőséget használó összekapcsolásnál használható. A táblákat külön-külön rendezzük (ha korábban nem voltak rendezve), majd a rendezett lista alapján összefuttatjuk.

  – Tördelésen alapuló algoritmusok

    A [1, 321. oldal] könyvben részletesebben ismertetett kétmenetes algoritmus szintén csak egyenlőséget használó összekapcsolásnál használható. A módszer nem használható az RBO-val.

    A táblákat kisebb partíciókra bontása után, a partíciókhoz tördelőtáblázatokat készít a rendszer, majd ezek segítségével kapcsolja össze a táblákat.

  – Klaszter összekapcsolás

    Azonos klaszterben tárolt táblák összekapcsolására használható módszer. Az azonos blokkban tárolt sorokban a klaszter kulcs értéke azonos, így a (klaszter kulcs alapján történő) összekapcsolás hatékonyan hajtható végre.

  A beágyazott ciklusú összekapcsolás nagy táblák esetén nem hatékony. RBO ilyen esetben a rendezésen alapuló algoritmust választja, míg CBO többnyire a tördelésen alapulót. Az egyes összekapcsolási módszerek pontosabb költségbecsléséről (figyelembe véve a rendelkezésre álló memória méretét) olvashatunk a [5, 11. fejezet].

- Összekapcsolási sorrend meghatározása

  Az összekapcsolási sorrend meghatározása másként történik CBO és RBO esetén, vannak azonban alapvető dolgok, melyeket azonos módon kezelnek az optimalizálók:

  – Ha biztosított, hogy bizonyos táblákból csak egy sort kell kiválasztani, akkor az optimalizáló ezeket a táblákat helyezi a sor elejére.

  – Külső összekapcsolásnál a két tábla egymáshoz viszonyított helyzete adott, nem veszi számításba az optimalizáló azokat a sorrendeket, amelyek megsértenék ezt a szabályt.

Sorrend meghatározása CBO-nál. A költség alapú optimalizáló több különféle összekapcsolási sorrendet és módot generál, majd megpróbálja ezek költségét megbecsülni.

Beágyazott ciklusú összekapcsolásnál a költség a külső tábla kiválasztott sorainak, és a belső táblában ezekhez a sorokhoz kapcsolható sorok olvasási költségét tartalmazza. Rendezésen alapuló összekapcsolásnál a költség az összes sor beolvasásának és a rendezésnek a költsége. Természetesen ezeket a költségértékeket befolyásolhatják egyéb tényezők, például a rendezés számára rendelkezése álló memória mérete, illetve a beolvasandó blokkok egymáshoz viszonyított helyzete.

A sorrendet explicit módon befolyásolhatjuk az `ORDERED` tipp használatával. Ha a tipp által megadott sorrend nem felel meg a külső összekapcsolásoknak, akkor az optimalizáló nem veszi figyelembe a tippet.

Sorrend meghatározása RBO-nál. Az RBO többnyire nem veszi figyelembe a táblák `FROM` utáni sorrendjét. Az optimalizáló egy ciklusban mindig az egyik legmagasabb rangú táblaelérési móddal rendelkező táblát választja ki, és helyezi el a sor elejére. Azt, hogy milyen módon kapcsolja ezt a táblát a már korábban összekapcsolt táblákhoz, a táblaelérési mód rangja alapján dönti el az optimalizáló.

A következő optimalizálási lépésben az RBO megpróbálja maximalizálni azon beágyazott ciklusú összekapcsolások számát, ahol a belső táblát index segítségével éri el. Döntetlen esetén további heurisztikák segítenek a döntésben.

2.6. Azonos alkifejezést tartalmazó lekérdezések optimalizálása

Ez az optimalizálási heurisztika felismeri azokat az azonos alkifejezéseket, melyek a lekérdezés különböző diszjunktív ágaiban találhatóak. Csak akkor működik a módszer, ha az összes diszjunktív ágban megtalálható a közös kifejezés.

A heurisztikát mutatja be a 4.9. teszt.

2.7. Kifejezések és feltételek kiértékelése

Bizonyos kifejezéseket ekvivalens formára alakít át az optimalizáló. Vagy azért mert az átalakított formát gyorsabban tudja kiértékelni, vagy azért mert az eredeti forma szintaktikailag megegyezik az újjal, és csak kényelmi funkciókat szolgál. Ezeket az

átalakításokat tehát felesleges nekünk megtenni, nem tudjuk ilyen módon javítani a lekérdezéseink teljesítményét.

A következő kifejezéseket alakítja át az Oracle:

- Konstansok

  A konstansokkal végzett műveleteket csak egyszer értékeli ki a rendszer. Például a `BASIC_SALARY > 12 * 100000` kifejezés a `BASIC_SALARY > 1200000` kifejezéssel ekvivalens.

- `LIKE` operátor

  Ha a `LIKE` operátor paramétere nem tartalmaz helyettesíthető karaktereket (pl. `NAME like 'Nagy'`), és változó hosszú adattípusra (pl. `VARCHAR2`) vonatkozik, akkor egyenlőséggel helyettesíthető (`NAME = 'Nagy'`).

- `IN` operátor

  Az `IN` operátor helyettesíthető egyenlőségekkel és `OR` operátorokkal:

  ```
  TYPE IN ('A', 'B', 'C')
  TYPE = 'A' OR TYPE = 'B' OR TYPE = 'C'
  ```

- `ANY` vagy `SOME` operátor

  Az `ANY` vagy `SOME` operátor is `OR` operátorral összekapcsolt egyszerűbb kifejezésekkel helyettesíthető:

  ```
  BASIC_SALARY > ANY(SAL1, SAL2)
  BASIC_SALARY > SAL1 OR BASIC_SALARY > SAL2
  ```

  Ha `ANY` után egy allekérdezés található, akkor `EXIST` operátorra van szükség:

  ```
  x > ANY (SELECT sal FROM emp WHERE job = 'Analyst')
  EXISTS (SELECT sal FROM emp
  WHERE job = 'Analyst' AND x > sal)
  ```

- `ALL` operátor

  Az `ALL` operátor `AND` operátorral összekötött egyszerűbb kifejezésekkel helyettesíthető:

  ```
  BASIC_SALARY > ALL(SAL1, SAL2)
  BASIC_SALARY > SAL1 AND BASIC_SALARY > SAL2
  ```

  Allekérdezésnél `ANY` -re van szükség:

  ```
  x > ALL (SELECT sal FROM emp WHERE job = 'Analyst')
  ```

```
NOT( x <= ANY
(SELECT sal FROM emp WHERE job = 'Analyst'))
```

Az előzőleg megismert szabállyal ez tovább transzformálható:

```
NOT EXISTS (SELECT sal FROM emp
WHERE job = 'Analyst' AND x<=sal)
```

- `BETWEEN` operátor

  A `BETWEEN` operátor egyszerű `<=` , `>=` segítségével ekvivalens formára hozható:

  ```
  BASIC_SALARY BETWEEN 100000 AND 200000
  ```

  ```
  BASIC_SALARY >= 100000 AND BASIC_SALARY <= 200000
  ```

- `NOT` operátor

  A `NOT` operátor elhagyható, ha a kifejezést negáljuk:

  ```
  NOT dept_id =
  (SELECT dept_id FROM emp WHERE contract_id = 11011)
  ```

  ```
  dept_id !=
  (SELECT dept_id FROM emp WHERE contract_id = 11011)
  ```

  Ha a `NOT` operátor bonyolult kifejezés előtt áll, akkor a cél a kifejezés egyszerűsítése, még akkor is, ha ezzel esetleg növekszik a `NOT` operátorok száma.

- Tranzitivitás

  A csak CBO-nál használható módszernél új — a tranzitivitás szabályából levezethető — feltételekkel bővíti az optimalizáló a lekérdezést.

  ```
  SELECT *
  FROM depts, emp
  WHERE emp.dept_id = 'NATI' AND
  emp.dept_id = depts.dept_id
  ```

  ```
  SELECT *
  FROM depts, emp
  WHERE emp.dept_id = 'NATI' AND
  emp.dept_id= depts.dept_id AND
  depts.dept_id = 'NATI'
  ```

  Ha a `depts` tábla `dept_id` oszlopán van index, akkor a második lekérdezés gyorsabban hajtható végre.

2.7.1. Determinisztikus függvények

Ha determinisztikus függvényeket használunk a lekérdezéseinkben, akkor ezeket elég egyszer kiértékelni, hiszen azonos inputra mindig azonos eredményt adnak. Egy függvény akkor számít determinisztikusnak, ha a `DETERMINISTIC` kulcsszóval definiáljuk. A rendszer nem ellenőrzi azt, hogy a függvény valóban determinisztikus-e, ezért vigyázni kell, hogy csak valóban determinisztikus függvénynél használjuk ezt a kulcsszót.

Egy determinisztikus függvényeket használó tesztet mutat be a 4.6. rész.

2.7.2. Transzformációk

Bizonyos esetekben az Oracle átalakítja a lekérdezést egy olyan alakra, melyet gyorsabban tud végrehajtani:

- `OR` átalakítása `UNION ALL` -ra.

  Ha a feltétel több egymással `OR` operátorral összekapcsolt feltételt tartalmaz, akkor a lekérdezés átírható több lekérdezés uniójára. Akkor hatékony az átalakítás, ha az új lekérdezésekben különböző indexek segítségével érhetjük el az adatokat.

```
SELECT  *
FROM  emp
WHERE  dept_id='ECON'    OR  contract_type_id     = 'BPFT'
```

```
SELECT  *
FROM  emp
WHERE  dept_id='ECON'
UNION  ALL
SELECT  *
FROM  emp
WHERE  contract_type_id     = 'BPFT'
```

- Összetett lekérdezések összekapcsolássá alakítása

  Ha egy összetett lekérdezést összekapcsolássá alakít a rendszer, akkor alkalmazhatóak mindazon módszerek, melyek az összekapcsolásokat optimalizálják.

```
SELECT  * FROM  emp
WHERE  dept_id  IN
(SELECT  dept_id  FROM  depts)
```

```
SELECT  emp.*
FROM  emp,  depts
WHERE  emp.dept_id    = depts.dept_id
```

3. fejezet

Lekérdezések optimalizálása

3.1. Megközelítési módok

Az Oracle dokumentációja [4] öt különböző módot javasol, melyek segítségével optimalizálhatjuk lekérdezésünket:

- Indexek átszervezése
- Lekérdezések átszervezése
- Triggerek módosítása
- Adatok átszervezése
- Statisztika frissen tartása

A továbbiakban ezeket vizsgáljuk meg alaposabban.

3.1.1. Indexek átszervezése

Indexek alkalmazásával felgyorsíthatjuk lekérdezéseinket. Egy lassú lekérdezésnél érdemes megvizsgálni, nem tudunk-e egy olyan indexet létrehozni, ami javítaná a teljesítményt. Gyakran automatikus módon rengeteg indexet készítünk tábláinkhoz, ez jelentősen lelassíthatja az adatmódosító műveleteket. Ha az indexet nem használják lekérdezéseink (például alacsony szelektivitása miatt), akkor érdemes elgondolkodni az index törlésén.

3.1.2. Lekérdezések átszervezése

Az SQL elég általános nyelv ahhoz, hogy egy adott célt többféle módon is leírhassunk. Ha két SQL lekérdezés ugyanazt az eredményt adja, az nem jelenti azt, hogy a sebességük is azonos, vagyis érdemes lehet a lekérdezés átírása.

- `NOT IN` átírása `NOT EXISTS` -re
- Lehetőség szerint mindig használjunk egyenlőséget használó összekapcsolást (equjoin)
- Válasszunk előnyös összekapcsolási sorrendet.

  Az összekapcsolt táblák sorrendje jelentősen befolyásolja a lekérdezés sebességét. Az SQL optimalizálás fő célja a felesleges munkavégzés (felesleges sorok olvasása) elkerülése. Három általános szabályt kell betartani ehhez:

  – Kerüljük el a teljes tábla elérési módot (2.3.2. rész), ha a szükséges sorokat egy index segítségével is megkaphatjuk.
  – Ha a főtáblán két indexet is használhatunk, és az egyik 10000, a másik 100 sort ad vissza, akkor azt érdemes választanunk, amelyik 100 sort ad vissza.
  – Úgy válasszuk meg az összekapcsolás sorrendjét, hogy kevesebb sort kelljen összekapcsolni.

- Transzformálatlan mezőértékeket használjunk ha lehetséges.

  A következő két lekérdezés közül, ha lehetséges, válasszuk az elsőt, mert transzformált mezőértékeknél nincs lehetőségünk indexek használatára.

```
WHERE  a.order_no    = b.order_no

WHERE  TO_NUMBER(    SUBSTR(
a.order_no,     instr(a.order_no,      '.')  -1))  =
TO_NUMBER(    SUBSTR(
b.order_no,     instr(b.order_no,      '.')  -1))
```

- Kerüljük a kevert-típusú kifejezéseket.

  Az implicit típuskonverziók miatt gyakran nem szükséges, hogy egy reláció mindkét oldalán azonos típusú értékeket használjunk. Az implicit típuskonverziók azonban nagyon lelassíthatják a lekérdezést.

  Ha a `WHERE` feltételben az oszlop helyett egy az oszlopot használó függvény szerepel, akkor a szerver nem tudja az oszlopon lévő indexet használni.

- Külön lekérdezések írása különböző célokra

  Ha hasonló célokra kell SQL lekérdezéseket írnunk, akkor gyakran egyetlen lekérdezést készítünk, amely a kapott paraméterektől függően különböző célokat valósít meg. Bár a módszernek számos előnye van, a lekérdezés sebessége többnyire lassabb lesz, mintha külön-külön lekérdezésékkel valósítanánk meg a különböző célokat.

Mivel a lekérdezés optimalizálása még a paraméterek kiértékelése előtt történik, ha a paraméterek értékétől függ egy index használhatósága, akkor abban az esetben sem fogja az optimalizáló az indexet használni, ha a paraméter bizonyos értekei ezt engedélyeznék.

- Tippek (hint) használata.

  Bár különböző trükkökkel is elérhetjük, hogy az optimalizáló más elérési utat válasszon, sokkal szerencsésebb ha tippeket használunk helyette. Ha teljes táblaelérést szeretnénk index elérés helyett használni, akkor ahelyett, hogy egy üres stringet adunk a mező értékeihez, használjuk inkább a `/*+ FULL */` tippet.

```
SELECT e.ename
FROM emp e
WHERE e.job || '' = 'CLERK'

SELECT /*+ FULL (emp) */ e.ename
FROM emp e
WHERE e.job = 'CLERK'
```

- Óvakodjunk az adatértékek listájának használatakor

  Ha lehetséges adatértékek listáját kell használnunk, akkor többnyire az adatbázisból hiányzik egy mező. Ahelyett hogy felsorolnánk például azokat a tanszékeket melyek nem valódi tanszékek (csak technikai okokból szerepelnek az adatbázisban), vezessünk be a egy olyan mezőt, mely minden tanszékre megmondja valódi, vagy technikai tanszékről van-e szó.

- Használjunk `INSERT, UPDATE, DELETE ... RETURN` utasítást, ha egyszerre szeretnénk végrehajtani egy `SELECT` és egy adatmódosító utasítást.

- Vigyázzunk a view-kkal.

  Bár a view-k megkönnyítik a lekérdezések írását, gyakran okozhatnak hatékonytalanságot. Ha egy túl általános célú view-t használunk, akkor felesleges számításokkal terheljük a rendszert. Szintén nagyon lassú, ha külső összekapcsolásban használunk view-kat.

3.1.3. Triggerek módosítása

Túl sok trigger használata lelassíthatja alkalmazásainkat, érdemes lehet kikapcsolni a felesleges triggereket.

3.1.4. Adatok átszervezése

Ha a korábbi módszerek nem segítenek, akkor szükség lehet az adatok átszervezésére. Egy már meglévő rendszerben — ahol több alkalmazás is olvassa a táblákat — ez gyakran nehezen megoldható, hiszen a többi alkalmazás módosítására is szükség lehet.

Egy új rendszer tervezésénél (vagy kibővítésénél) azonban hasznos ezeket az elveket figyelembe venni.

3.1.5. Statisztika frissen tartása

A költség-alapú optimalizálás hatékony működéséhez szükség van statisztikákra. Hiányos, vagy elavult statisztikák lelassíthatják lekérdezéseinket, ezért a lekérdezés optimalizálása után is szükség van a statisztikák karbantartására.

Statisztika készítésénél el kell dönteni mely objektumokra készítünk statisztikákat. Minél több objektumot analizálunk, annál több időre és tárhelyre van szükségünk, és nem garantált, hogy a teljesítmény ezzel arányosan növekszik. Az is előfordulhat, hogy több statisztika használata lelassítja bizonyos lekérdezéseinket.

Azt is el kell dönteni, hogy mely esetekben használunk becslést az időigényesebb alapos számítás helyett.

3.2. Gyakorlati ötletek

A következő részben olyan ötleteket ismertetünk, mely a költség alapú optimalizálóval (2.3. rész) használható.

- Kerüljük el a hagyományos szabály alapú optimalizálási trükköket.

  Szabály alapú optimalizálásnál különböző trükkökkel tudjuk befolyásolni, hogy az optimalizáló használjon-e bizonyos indexeket. Költség alapú optimalizálásnál erre nincs szükség, az optimalizáló csak akkor használ egy bizonyos indexet, ha a költségelemzés alapján érdemesnek tűnik.

  Hasonlóan felesleges az összekapcsolás során használt táblák helyes sorrendjének meghatározása, az optimalizáló meg tudja határozni az optimális sorrendet.

  A gyakorlatban azonban előfordulhat, hogy bizonyos esetekben a költség alapú optimalizáló téved, és mégis szükség van tippek használatára.

- Kerüljük el a bonyolult kifejezéseket

  Amennyiben lehetséges kerüljük a következő összetett kifejezéseket:

  - `col1 = NVL(:b1,col1)`
  - `NVL(col1, -999)`

- `to_date()`, `to_number()`

  Ezek a kifejezések megakadályozzák, hogy az optimalizáló helyesen becsülje meg szelektivitást vagy számosságot.

- Kerüljük el a bonyolult sok célra megfelelő, emiatt túlzottan általános és nehezen optimalizálható lekérdezések írását.

- Az összetett logikát kezeljük alkalmazásainkban.

  A lekérdezések helyett optimálisabb az összetett logikai összefüggéseket alkalmazásainkban kezelni. Egy C program többnyire gyorsabban el tudja végezni a munkát, mint egy nehézkesen megírt lekérdezés.

  Természetesen ha több különböző programozási nyelven írt alkalmazásunk is használja a rendszert, akkor ez a módszer nehezen megvalósítható.

3.3. Tuning tippek

A 3.1. táblázat az Oracle által javasolt tuning tippeket foglalja össze, melyeket az SQL utasításaink tervezése során használhatunk.

3.3.1. Predikátum összeomlasztás (predicate collapsing)

Akkor használhatunk predikátum összeomlasztást, ha egy kifejezésben egynél több paraméter szerepel. A következő kifejezésben például két paramétert is találhatunk (`col = DECODE(:b1, '', :b3, col)`). A kifejezésnél az optimalizáló nem tudja a `col` oszlopon lévő indexet használni. Ha az első paraméter értéke `NULL`, akkor a kifejezést `col = :b3` formában, ha nem `NULL`, akkor `col = col` formában írhatjuk át. Első esetben lehetőségünk van az index használatára, míg utóbbi esetben a feltételre nincs is szükség. Az eredeti kifejezést tehát két kifejezéssel helyettesíthetjük, melyeket `UNION` köt össze.

A predikátum összeomlasztást bemutató teszteredményt a 4.3. részben találhatjuk.

3.3.2. Lemez és puffer használat figyelés

A lemez és puffer műveletek számát a `SET AUTOTRACE ON` parancs segítségével ellenőrizhetjük. A két legfontosabb jellemző a `consistent gets` és a `physical reads`. Ha ezek száma túl nagy a visszaadott sorok számához képest, akkor feleslegesen olvas a szerver a lekérdezés futtatása során, vagyis érdemes a lekérdezést tovább optimalizálni.

Segíthet az `SQL Trace` és a `TKPROF` használata is, ezekről bővebben a [4] dokumentáció 6. fejezetében olvashatunk.

3.1. táblázat. Tuning tippek

Tipp	Megjegyzés
Végezzük el a munkát gyorsabban, vagy végezzünk kevesebb munkát.	Törekedjünk arra, hogy minél kevesebb sort válasszunk ki.
Bontsuk szét `JOIN` utasításainkat.	Ellenőrizzük az összes `JOIN` utasítást, és ellenőrizzük feltétlenül szükség van-e rájuk.
View-k ellenőrzése.	Ha egy view-t használ a lekérdezésünk, akkor vizsgáljuk meg azt is, hogy a view optimalizált-e, és azt, hogy teljesen szükségünk van-e a view-ra, lehetséges-e, hogy egy egyszerűbb view is megfelel célunknak.
Ne aggódjunk a teljes tábla eléréstől (2.3.2. rész), különösen kis táblák esetén.	Bizonyos esetekben olcsóbb ez az elérési mód mint az indexek használata, különösen kis tábla, vagy alacsony szelektivitású indexek esetén.
A végrehajtási tervet vizsgáljuk meg alaposan.	Az index elérések és a beágyazott ciklusú összekapcsolások (`NL JOIN`) nem mindig optimálisak.
Hosszú ideig futó lekérdezéseknél alapvető számításokkal ellenőrizzük érdemes-e optimalizálni.	A lekérdezésre szánt idő és a szükséges adatolvasás mennyiségéből kiszámíthatjuk azt az adatolvasási sebességet, melyet ha hardverünk nem tud biztosítani, akkor optimalizálás után is lassú lesz lekérdezésünk.
Vizsgáljuk a lemez és puffer használatot.	Bővebben lásd a 3.3.2. részt.
Összekapcsolások vizsgálata.	Vizsgáljuk meg az outer joinokat, és az összekapcsolások allekérdezéssel való helyettesítésének lehetőségét.
Válasszunk `EXISTS` és `IN` között.	Válasszuk azt, amelyik optimálisabb. Bővebben lásd a 3.3.3. részt.
Predikátum összeomlasztás.	Bővebben lásd a 3.3.1. részt.
Tipikus esetre optimalizálás	

3.3.3. EXISTS és IN használata

Az EXISTS és az IN kulcsszavak hasonló célokat szolgálnak, többnyire átírhatjuk úgy lekérdezésünket, hogy a másikat használja. Részletesebb tesztet mutat be a 4.2. rész.

4. fejezet

Teszteredmények

A tesztek során a korábban ismertetett elméleti módszerek egy részét próbáltam ki a gyakorlatban.

4.1. Tesztkörnyezet

A teszteléshez egy Linux (Redhat 7.1) operációs rendszer alatt futó Oracle 8.1.7-es adatbázist használtunk. Az adatbázisszervert nem a tesztelés céljára installáltuk, sőt a tesztek alatt is teljesítette elsődleges feladatát. Ez elvileg befolyásolhatta volna a tesztek eredményét, de a tesztelés időpontjában az adatbázisszerver nem volt leterhelve, és a lekérdezéseket ellenőrzésként többször is lefuttattuk.

A teszteket *SQL WorkSheet*ben futtattam, az idő méréséhez a beépített időmérést (`SET TIMING ON`) használtam, a végrehajtási tervet a szerverhez mellékelt (és kismértékben módosított) `utlxpls` script segítségével készítettem.

4.2. `EXISTS` és `IN` összehasonlítása

Az `EXISTS` -et használó lekérdezés csak teljes tábla eléréssel tudja elérni a nagyobb méretű `emp` táblát. A lekérdezés 2.42 másodperc alatt fut le.

```
SELECT COUNT(*)
FROM emp
WHERE (EXISTS (SELECT dept_id
FROM depts
WHERE (emp.dept_id = depts.dept_id AND
ISACADEMIC='Y')))
```

```
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                                |   Name                |   Rows  | Bytes|    Cost    |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT   STATEMENT                       |                       |      1  |     4  |     525  |
|  SORT  AGGREGATE                         |                       |      1  |     4  |          |
|    FILTER                                |                       |         |        |          |
|     TABLE   ACCESS   FULL                |EMP                    |     24K|     95K|      525  |
|     TABLE   ACCESS   BY INDEX   ROWID    |DEPTS                  |      1  |     5  |        1  |
|      INDEX   UNIQUE   SCAN               |PK_DEPT                |      1  |        |          |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
```

A lekérdezés `IN`-re átírt változata lényegesen gyorsabb, 0.52 másodperc elég a lefutásához. A gyorsulás oka, hogy a nagyméretű `emp` táblát index segítségével érhetjük el.

```
SELECT   COUNT(*)
FROM   emp
WHERE   dept_id   in (SELECT   dept_id
FROM   depts   WHERE   isacademic='Y')
```

```
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                                |   Name                |   Rows  | Bytes|    Cost    |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT   STATEMENT                       |                       |      1  |     9  |     215  |
|  SORT  AGGREGATE                         |                       |      1  |     9  |          |
|    HASH   JOIN                           |                       |    199K|      1M|     215  |
|     TABLE   ACCESS   FULL                |DEPTS                  |     39  |   195  |        1  |
|     INDEX   FAST   FULL   SCAN           |EMP_DEPT_IDX           |    491K|      1M|     106  |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
```

4.3. Predikátum összeomlasztás

Az itt ismertetett példa — ami a predikátum összeomlasztás technikáját mutatja be — egy, az adatbázisban előforduló összetett példán alapszik, de ahhoz képest radikálisan leegyszerűsített.

Tanórák bizonyos paramétereit több szinten határozhatjuk meg. A tanszék specifikus paramétereket a `class_dept_specs` , a szak specifikus paramétereket a `class_stream_specs` táblában tároljuk. Ha van szak specifikus paraméter, akkor azt kell figyelembe venni, ha nincs, akkor a tanszék specifikust.

A következő egyszerű példa kilistázza azokat a tanórákat, ahol a `wl_priority` (várakozólista prioritás) paraméter értéke 3.

```
SELECT   class_dept_specs.act_class_id
FROM   class_dept_specs,      class_stream_specs
WHERE   class_dept_specs.act_class_       id =
class_stream_specs.act_class_       id(+ )
and  NVL(stream_wl_priority,dept_w       l_pri orit y)  = 3
```

Az NVL kifejtésével bonyolultabb, de remélhetően gyorsabb lekérdezést kapunk:

```
SELECT  class_dept_specs.act_class_id
FROM
class_dept_specs,      class_stream_specs
WHERE  class_dept_specs.act_class_       id =
class_stream_specs.act_class_       id
AND  stream_wl_priority      IS  NOT  NULL
AND  stream_wl_priority      = 3
UNION  ALL
SELECT  class_dept_specs.act_class_id
FROM
class_dept_specs,      class_stream_specs
WHERE  class_dept_specs.act_class_       id =
class_stream_specs.act_class_       id(+ )
AND  stream_wl_priority      IS  NULL
AND  dept_wl_priority     = 3
```

4.3.1. Első teszt

A táblák néhány ezer sort tartalmaznak, analizáltak, de nincs index a két táblát összekapcsoló act_class_id mezőkre.

Az első lekérdezés végrehajtásához szükséges idő 0.24 másodperc, a végrehajtási terv a következő:

```
-------------------------------------------          -------  -------  -----
| Operation                    |  Name     |  Rows  | Bytes|   Cost   |
-------------------------------------------          -------  -------  -----
| SELECT  STATEMENT            |           |     5K|    46K|      21  |
|  FILTER                      |           |       |       |          |
|   HASH  JOIN  OUTER          |           |       |       |          |
|    TABLE  ACCESS  FULL       |CLASS_DEP  |     5K|    25K|       3  |
|    TABLE  ACCESS  FULL       |CLASS_STR  |     3K|    12K|       2  |
-------------------------------------------          -------  -------  ------
```

A második lekérdezés végrehajtásához 0.15 másodperc szükséges. A lekérdezési tervet megvizsgálva látható, hogy a gyorsulás abból adódik, hogy a második esetben az egyik ágon nem szükséges külső összekapcsolást használni:

```
-----------------------------------------          -------  -------  -----
| Operation                    |  Name     |  Rows  | Bytes|   Cost   |
-----------------------------------------          -------  -------  -----
| SELECT  STATEMENT            |           |   185  |    1K|      14  |
|  UNION-ALL                   |           |        |      |          |
|   HASH  JOIN                 |           |    53  |  371 |       7  |
|    TABLE  ACCESS  FULL       |CLASS_STR  |    42  |  168 |       2  |
|    TABLE  ACCESS  FULL       |CLASS_DEP  |    5K  |   15K|       3  |
|   FILTER                     |           |        |      |          |
|    HASH  JOIN  OUTER         |           |        |      |          |
|     TABLE  ACCESS  FULL      |CLASS_DEP  |    65  |  325 |       3  |
|     TABLE  ACCESS  FULL      |CLASS_STR  |    3K  |   12K|       2  |
-----------------------------------------          -------  -------  -----
```

4.3.2. Második teszt

A két táblát összekapcsoló `act_class_id` mezőkre indexet és statisztikát készítettem.

A végrehajtási idők lényegében nem változtak, a második lekérdezés végrehajtási ideje (hibahatáron belül) 0.14 másodpercre csökkent. A második lekérdezés végrehajtási tervét megvizsgálva, látható, hogy az indexek használatával nem csökken lényegesen a költség.

```
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                                |  Name                |  Rows  | Bytes|   Cost   |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT  STATEMENT                        |                      |   137  |    1K|      13  |
|  UNION-ALL                               |                      |        |      |          |
|   NESTED  LOOPS                          |                      |      5 |   35 |       6  |
|    TABLE  ACCESS  BY  INDEX  ROWID       |CLASS_STREAM_SPECS    |      4 |   16 |       2  |
|     INDEX  RANGE  SCAN                   |STREAM_WL_IDX         |      4 |      |       1  |
|    INDEX  RANGE  SCAN                    |DEPT_ACT_CLASS_IDX    |      5K|   15K|       1  |
|   FILTER                                 |                      |        |      |          |
|    HASH  JOIN  OUTER                     |                      |        |      |          |
|     TABLE  ACCESS  FULL                  |CLASS_DEPT_SPECS      |    65  |  325 |       3  |
|     TABLE  ACCESS  FULL                  |CLASS_STREAM_SPECS    |      3K|   12K|       2  |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
```

4.3.3. Harmadik teszt

Az index előnyét jobban mutatandó, a táblák méretét mesterségesen nyolcszorosára növeltem, így az első lekérdezés 5, a második 3.7 másodpercig fut.

Az első lekérdezés végrehajtási tervének szerkezetében nem található változás, a második terv azonban kismértékben módosult:

```
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| Operation                                |   Name                |  Rows  | Bytes|    Cost   |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| SELECT  STATEMENT                        |                       |     7K|    62K|       70 |
|  UNION-ALL                               |                       |       |       |          |
|   HASH  JOIN                             |                       |     3K|    22K|       31 |
|    TABLE  ACCESS  FULL                   |CLASS_STREAM_SPECS     |   331 |     1K|       11 |
|    INDEX  FAST  FULL  SCAN               |DEPT_ACT_CLASS_IDX     |    42K|   123K|       10 |
|   FILTER                                 |                       |       |       |          |
|    HASH  JOIN  OUTER                     |                       |       |       |          |
|     TABLE  ACCESS  FULL                  |CLASS_DEPT_SPECS       |   280 |     1K|       21 |
|     TABLE  ACCESS  FULL                  |CLASS_STREAM_SPECS     |    25K|   100K|       11 |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
```

A tesztek megmutatják a módszer legnagyobb hátrányát is, az új lekérdezés sokkal bonyolultabb az eredetinél. A valódi adatbázisban szereplő példában (aminek leegyszerűsített változatát használtam itt) négy NVL szerepel, vagyis a kifejtés után 16 lekérdezést kellene összeuniózni. Ez nagymértékben megnehezíti a lekérdezés karbantartását, így csak speciális esetben éri meg.

4.4. Bittérképindexek

Az emp tábla oszlopain hiába definiáltam hagyományos B-fa indexeket, a következő lekérdezés végrehajtásakor mégis teljes tábla elérést választott a rendszer, a futáshoz 0.89 másodpercre volt szükség:

```
SELECT  * FROM  emp
WHERE  sex_id='F'
AND  marital_stat_id='M'
AND  citizenship_id     IN  ('HUN',   'RUS',   'USA')
AND  salary  > 10000
AND  dept_id='HIST'
```

```
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| Operation                                |   Name                |  Rows  | Bytes|    Cost   |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| SELECT  STATEMENT                        |                       |      8 |  552  |     525  |
|  TABLE  ACCESS  FULL                     |EMP                    |      8 |  552  |     525  |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
```

Az indexeket azért nem használta a rendszer, mert mindegyik túl sok sort ad vissza, vagyis alacsony a szelektivitásuk.

A B-fa indexeket bittérképindexekkel helyettesítve már fel tudja használni a rendszer az indexeket, mint az a végrehajtási terven is látszik:

```
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| Operation                                |   Name                |  Rows  | Bytes|    Cost   |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| SELECT  STATEMENT                        |                       |      8 |  552  |     147  |
|  TABLE  ACCESS  BY  INDEX  ROWID         |EMP                    |      8 |  552  |     147  |
|   BITMAP  CONVERSION   TO  ROWIDS        |                       |        |       |          |
|    BITMAP  AND                           |                       |        |       |          |
|     BITMAP  INDEX  SINGLE  VALUE         |EMP_DEPT_BM_IDX        |        |       |          |
|     BITMAP  INDEX  SINGLE  VALUE         |EMP_MAR_BM_IDX         |        |       |          |
|     BITMAP  INDEX  SINGLE  VALUE         |EMP_SEX_BM_IDX         |        |       |          |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
```

Az indexek közül a `citizenship_id` mezőn definiáltat ugyan nem használta a rendszer, de a futási idő így is jelentősen felgyorsult, 0.19 másodpercre volt már csak szükség.

4.5. Függvény-alapú index

4.5.1. Hagyományos index használata

A mesterségesen feltöltött, 491000 sort tartalmazó `emp` táblán a `name` mező alapján történő keresés nagyon gyors (0.10 másodperc), hiszen a rendszer használhatja a `name` mezőn lévő indexet:

```
SELECT  * FROM  emp  WHERE  name  = 'AAAA'
```

```
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| Operation                                |   Name                |  Rows  | Bytes|    Cost   |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| SELECT  STATEMENT                        |                       |      2 |  138  |       2  |
|  TABLE  ACCESS  BY  INDEX  ROWID         |EMP                    |      2 |  138  |       2  |
|   INDEX  RANGE  SCAN                     |EMP_NAME_IDX           |      2 |       |       1  |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
```

Ha az összehasonlításnál nem szeretnénk különbséget tenni a kis és nagybetűk között (ami meglehetősen gyakori eset), akkor a lekérdezést kismértékben módosítani kell. A módosított lekérdezés nem tudja használni az indexet, ezért egy nagyságrenddel lassabb (1.02 másodperc).

```
SELECT  * FROM  emp  WHERE  LOWER(name)  = 'aaaa'
```

```
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| Operation                                |   Name                |  Rows  | Bytes|    Cost   |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
| SELECT  STATEMENT                        |                       |     4K|   330K|      525  |
|  TABLE  ACCESS  FULL                     |EMP                    |     4K|   330K|      525  |
-----------------------------------------          ------- ------- ------ ------- ------- ------ -
```

4.5.2. Függvény-alapú index használata

Függvény-alapú index készítése és használata lényegesen nehezebb mint a hagyományos index kezelése. Egyrészt külön jogosultság kell hozzá (a tesztben `system` felhasználóként hoztam létre), másrészt használatához módosítani kell a `session` egy paraméterét:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE
```

A módosítás után azonban a sebesség lényegében megegyezik az eredeti, `lower` függvény nélküli változattal (0.09 másodperc). A végrehajtási tervből látszik, hogy a lekérdezés használta az újonnan létrehozott indexet:

```
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
| Operation                                 |  Name                |  Rows  | Bytes|    Cost   |
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
| SELECT STATEMENT                          |                      |     4K|   330K|        4  |
|  TABLE ACCESS BY INDEX ROWID              |EMP                   |     4K|   330K|        4  |
|   INDEX RANGE SCAN                        |EMP_LLNAME_IDX        |     4K|       |        3  |
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
```

4.6. Determinisztikus függvény

Egy függvényt hoztam létre, mely minden országhoz meghatároz egy limitet, amit a következő lekérdezésben használok. Az egyszerűség kedvéért a limit egységesen 1000.

```
CREATE FUNCTION test1 (cit IN VARCHAR2)
RETURN NUMBER IS
BEGIN
RETURN 1000;
END;
```

A következő — test1 függvényt tesztelő — lekérdezés 16.48 másodpercig fut:

```
SELECT * FROM emp WHERE salary < test1(citizenship_id)
```

A végrehajtási terv a lekérdezéshez hasonlóan egyszerű:

```
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
| Operation                                 |  Name                |  Rows  | Bytes|    Cost   |
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
| SELECT STATEMENT                          |                      |    24K|     1M|      525  |
|  TABLE ACCESS FULL                        |EMP                   |    24K|     1M|      525  |
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
```

Ha a függvényt a `DETERMINISTIC` kulcsszóval determinisztikusnak definiáljuk, akkor a dokumentáció szerint a lekérdezésnek gyorsulni kellene, hiszen így minden országhoz csak egyszer kellene kiszámítani a `test1` függvény értékét. A tesztek nem támasztották ezt alá, mind a lekérdezési terv, mind a futási idő változatlan maradt.

Mivel a függvény nagyon egyszerű, készítettem egy olyan tesztet is, ahol a függvény valódi számítást végzett, de abban az esetben sem volt különbség a determinisztikus és a nem-determinisztikus függvény között.

4.7. Tábla önmagával összekapcsolásása

Az előzőleg már többször használt `emp` táblát kapcsoljuk össze saját magával, az `emp_id` mező segítségével:

```
SELECT  count(*)
FROM  emp  e1,  emp  e2
WHERE  e1.emp_id  = e2.emp_id
```

4.7.1. Rendezésen alapuló összekapcsolás

A lekérdezés 7.02 másodpercig fut, és a lekérdezési tervet megvizsgálva látszik, hogy rendezésen alapuló összekapcsolást választott a CBO:

```
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                               |   Name                |   Rows  | Bytes|    Cost   |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT  STATEMENT                       |                       |      1  |     8  |   4476  |
|  SORT  AGGREGATE                        |                       |      1  |     8  |         |
|    MERGE  JOIN                          |                       |    491K|      3M|    4476  |
|     INDEX  FULL  SCAN                   |PK_EMP                 |    491K|      1M|     927  |
|     SORT  JOIN                          |                       |    491K|      1M|    3459  |
|      INDEX  FAST  FULL  SCAN            |PK_EMP                 |    491K|      1M|      90  |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
```

4.7.2. Tördelésen alapuló összekapcsolás

Természetesen tippek segítségével lehetőségünk van arra, hogy a többi összekapcsolási módot is kipróbáljuk. A tördelésen alapuló lényegesen lassabb (15.69 másodperc):

```
SELECT  /*+  USE_HASH  (e1  e2)  */  count(*)
FROM  emp  e1,  emp  e2
WHERE  e1.emp_id  = e2.emp_id
```

```
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                               |   Name                |   Rows  | Bytes|    Cost   |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT  STATEMENT                       |                       |      1  |     8  |  35647  |
|  SORT  AGGREGATE                        |                       |      1  |     8  |         |
|    HASH  JOIN                           |                       |    491K|      3M|  35647  |
|     INDEX  FAST  FULL  SCAN             |PK_EMP                 |    491K|      1M|      90  |
|     INDEX  FAST  FULL  SCAN             |PK_EMP                 |    491K|      1M|      90  |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
```

Összekapcsolási mód	Hagyományos táblák	Klaszterben tárolt táblák
MERGE	38.24	95.96
HASH	119.54	140.42
NL	47.73	53.77

4.1. táblázat. Három tábla összekapcsolása

4.7.3. Beágyazott ciklusú összekapcsolás

A beágyazott ciklusú összekapcsolás a végrehajtási terv alapján nem túl biztató, ugyanakkor gyorsabb mint a két korábbi összekapcsolási mód (4.93 másodperc).

```
SELECT /*+ USE_NL (e1 e2) */ count(*)
FROM emp e1, emp e2
WHERE e1.emp_id = e2.emp_id
```

```
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
| Operation                                |  Name                |  Rows  | Bytes|    Cost   |
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
| SELECT STATEMENT                         |                      |     1 |     8 | 491090   |
|  SORT AGGREGATE                          |                      |     1 |     8 |          |
|    NESTED LOOPS                          |                      |   491K|     3M|  491090  |
|     INDEX FAST FULL SCAN                 |PK_EMP                |   491K|     1M|       90 |
|     INDEX UNIQUE SCAN                    |PK_EMP                |   491K|     1M|        1 |
-------------------------------------------  -------  -------  ------  -------  -------  ------  -
```

4.8. Három tábla összekapcsolása

Ebben a tesztben 3 táblát kapcsoltam össze, a már korábban használt `emp` táblát, és a `emp_sub1`, `emp_sub2` táblákat. E két tábla az `emp_id` mezővel kapcsolódik az `emp` táblához. A két új tábla háromszor annyi sort tartalmaz mint az `emp` tábla.

```
SELECT COUNT(*) FROM emp, emp_sub1, emp_sub2
WHERE emp.emp_id = emp_sub1.emp_id
AND emp.emp_id = emp_sub2.emp_id
```

A lekérdezés végrehajtásához az optimalizáló rendezésen alapuló összekapcsolást választ, a futás 38.24 másodpercig tart. A másik két összekapcsolási módszert is kipróbáltam az eredmény a 4.1. táblázatban található. Látható, hogy ebben az esetben már nem a beágyazott ciklusú összekapcsolás az optimális, és az is, hogy az optimalizáló helyesen választott a módszerek közül.

A tesztet elvégeztem arra az esetre is, amikor a táblákat egy klaszterben tárolom. Minden táblának elkészítettem a másolatát egy klaszterben (a táblákat megkülönböztető, a táblák neve elé `c_` került).

Minden tipp nélkül lefuttatva a lekérdezés 53.77 másodpercig fut. A végrehajtási

tervet megvizsgálva látszik, hogy a táblákat klaszter összekapcsolással (ami a beágyazott ciklusú összekapcsolás egy altípusa) kapcsolja össze a rendszer.

```
-------------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                                 |   Name                |   Rows  | Bytes|    Cost   |
-------------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT   STATEMENT                        |                       |      1  |    12  |1964089   |
|  SORT  AGGREGATE                          |                       |      1  |    12  |          |
|    NESTED   LOOPS                         |                       |     4M|     50M|1964089     |
|     NESTED   LOOPS                        |                       |     1M|     11M|  491089   |
|      INDEX   FAST   FULL   SCAN           |C_PK_EMP               |   491K|      1M|       89  |
|      TABLE   ACCESS   CLUSTER             |C_EMP_SUB1             |     1M|      5M|        1  |
|     TABLE   ACCESS   CLUSTER              |C_EMP_SUB2             |     1M|      5M|        1  |
-------------------------------------------          -------  -------  ------  -------  -------  ------  -
```

Tippek segítségével természetesen módosíthatjuk az összekapcsolási módszert. A tesztek eredményeit a 4.1. táblázat mutatja. Nem meglepő, hogy a tördelésen és rendezésen alapuló módszer lassabb a klasztert használó esetben, az azonban már inkább, hogy a beágyazott ciklusú is lassabb. Az optimalizáló azonban ebben az esetben is helyesen választott a három módszer közül, a leggyorsabbat választotta.

A klaszterek viszonylagos sikertelenségének legfőbb oka valószínűleg az egy klaszter kulcshoz tartozó sorok viszonylagosan alacsony száma. A mesterséges példában minden kulcshoz csak 7 sor tartozik. A másik, ezzel összefüggő ok a klaszter létrehozásakor beállított `SIZE` paraméter értéke. A paraméter értéke akkor optimális, ha az egy kulcshoz tartozó sorok összméretével egyezik meg. A példában az értéke valószínűleg túl nagynak bizonyult, ezt támasztja alá az is, hogy a klaszter helyfoglalása nagyobb volt mint a külön táblák helyfoglalásának összege.

4.9. Közös kifejezés

A közös kifejezés tesztelésére egy négy olyan lekérdezést készítettem, melyek nagyon hasonlóak:

```
SELECT  count(emp.name)
FROM  emp,  depts
WHERE  (emp.dept_id    = depts.dept_id    AND
emp.citizenship_id      = 'HUN'  AND  depts.isacademic=:p1)
OR  (emp.dept_id    = depts.dept_id    AND
salary  > 40000  AND  depts.isacademic     = :p2);
```

A két — `OR` operátorral összekapcsolt — ágban, az egyik feltétel a paraméterek értékétől függetlenül mindig közös (`emp.dept_id = depts.dept_id`), sőt ha `p1` és `p2` paraméterek érékei is megegyeznek, akkor a `depts.isacademic=:px` feltételek is közösek. A két paraméter `Y` és `N` értékeket vehet fel, így négy lekérdezést teszteltem. A végrehajtási terv mind a négy esetben azonos szerkezetű, nem mutatja meg azt, hogy a közös kifejezések összevonása megtörténik-e, ezért a futási idő alap-

ján tudjuk (nagy valószínűséggel) megállapítani, hogy megtörtént-e az összevonás. A következő táblázat mutatja a négy lekérdezéshez szükséges időt. Látszik, hogy az idők alacsonyabbak azokban az esetekben, ha `p1` és `p2` paraméterek értékei megegyeznek, vagyis az összevonás nagy valószínűséggel valóban megtörtént.

p1	*p2*	*idő*
N	N	1.32
Y	Y	1.16
Y	N	1.75
N	Y	1.71

4.10. Gyakorlati példák

Az ebben a részben leírt példák a valódi adatbázisból származó valódi példák, az optimalizálásnak lehetőleg a teljes menetét leírom akkor is, ha a lépések egy része utólag zsákutcának bizonyult.

4.10.1. Függvény használata lekérdezésben

A feladat. A következő példa egy riportkészítő eljárásban található meg. A táblákon kívül két view-t is használ a lekérdezés:

SWO_ACTIVE_STUDENT_PERIODS:

```
SELECT sten_id,period_id
FROM swo_active_student_intervals sasi, swo_periods sp
WHERE
((sp.open_date<=sasi.start_da te AND
(sp.finance_end_date IS NULL OR
sp.finance_end_date>=sasi.sta rt_d ate)) OR
(sasi.start_date<=sp.open_dat e AND
(sasi.end_date IS NULL OR
sasi.end_date>=sp.open_date)) OR
(sasi.start_date<=sp.open_dat e AND
(sasi.end_date IS NULL OR
sasi.end_date>=sp.finance_end _dat e)))
```

SWO_ACTIVE_STUDENT_INTERVALS

```
SELECT /*+ FIRST_ROWS */
saa_start.sten_id sten_id,
saa_start.autolog_timestamp start_date,
MIN(saa_end.autolog_timestamp ) end_date
FROM swo_as_autolog saa_start,
```

```
swo_as_autolog      saa_end
WHERE
saa_start.autolog_chg_type='I        ' AND
saa_end.autolog_chg_type(+)='        D' AND
saa_start.sten_id=saa_end.STE        N_ID (+) AND
saa_start.autolog_timestamp<s        aa_e nd.au tolo g_tim esta mp(+)
GROUP BY saa_start.sten_id,saa_start.a        utol og_ti mest amp
```

A lekérdezés amit optimalizálni kell a következő:

```
SELECT DISTINCT p.last_name,     p.first_name,
p.public_id,     cc.country_name,
ssi.sten_id,     NVL(sft.ftype_name,'       ') ftype_name,
NVL(sft.ftype_id,0)      ftype_id,
Swoman.total_huf_nocash_nonam        e(ss i.ste n_id ,ssi. peri od_id )
total_huf
FROM swo_rel_acad     sra,  rel_student_deptstream       rsd,
swo_student_items      ssi,  student_details     sd,
persons   p,  country_codes     cc,  swo_student_types      sst,
swo_financial_types      sft,  swo_active_student_periods        sasp
WHERE  sra.swo_stream_id=39       AND
rsd.dstream_id=sra.dstream_id          AND
ssi.sten_id=rsd.sten_id        AND
rsd.sten_id=sasp.sten_id        AND
sd.stud_id=rsd.stud_id       AND
p.person_id=sd.person_id        AND
cc.country_id(+)=p.citizenshi        p_id  AND
sst.sten_id(+)=ssi.sten_id        AND  sft.ftype_id(+)=sst.ftype_id
ORDER  BY  ftype_name    DESC,   p.last_name,     p.first_name
```

Az eredeti feladatban `swo_stream_id` értéke paraméter, még egy szűrés van a lekérdezésben, és egy cikluson belül többször is meghívódik ez a lekérdezés.

Eredeti eredmények

A lekérdezést lefuttatva a következő időeredményt kapjuk:

```
Parse                19.87   (Elapsed)         0.00   (CPU)
Execute/Fetch       148.25   (Elapsed)         0.12   (CPU)
Total               168.12                     0.12
```

A korábbi példákkal ellentétben itt már jelentős az elemzés ideje is. A végrehajtási tervet megvizsgálva feltűnik, hogy költség alapú optimalizálást használt a rendszer.

A következő táblázat csak — az egyébként meglehetősen hosszú — végrehajtási terv elejét mutatja.

```
-------------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                                 |  Name               |  Rows  | Bytes|    Cost   |
-------------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT  STATEMENT                         |                     |    27M|    10G|       25M|
|  SORT  ORDER  BY                          |                     |    27M|    10G|       25M|
|   SORT  UNIQUE                            |                     |    27M|    10G|       13M|
|    CONCATENATION                          |                     |       |       |          |
|     HASH  JOIN  OUTER                     |                     |     9M|     3G|  171892  |
```

Szabály alapú optimalizálás

Mivel rendszerünkben alapvetően szabály alapú optimalizálást használunk, először azt vizsgáltam meg, hogy miért választotta itt az optimalizáló a költség alapú optimalizálást. Ennek két oka volt, egyrészt az egyik view-ban használt — vélhetően egy korábbi optimalizálás során odakerült — `FIRST_ROWS` tipp, másrészt az, hogy az egyik tábla (`REL_STUDENT_DEPTSTREAM`) a táblák többségével ellentétben analizálva volt.

Első megközelítésként szabály alapú optimalizálásra tértem át. Kétféle módon is elértem ezt, egyrészt a `FIRST_ROWS` tipp, és a statisztika törlésével, másrészt a `RULE` tipp explicit használatával. A futási eredmények között nincs jelentős különbség, egyiket megvizsgálva észrevehetjük, hogy a gyorsulás oka az elemzés felgyorsulása, a lekérdezés valódi végrehajtásához szükséges idő nem változott jelentősen.

```
Parse          0.02  (Elapsed)       0.00  (CPU)
Execute/Fetch  148.07  (Elapsed)     0.10  (CPU)
Total          148.09                0.10
```

Költség alapú optimalizálás

A lekérdezésben szereplő táblák analízise után már azt várhatjuk, hogy sikeresebben tudja a rendszer a költség alapú optimalizálást használni. Az időeredményeket megvizsgálva látható némi gyorsulás, de a gyorsulás mértéke nem túl jelentős.

```
Parse          0.69  (Elapsed)       0.00  (CPU)
Execute/Fetch  139.81  (Elapsed)     0.11  (CPU)
Total          140.50                0.11
```

A végrehajtási tervet megvizsgálva (a hossz miatt itt csak az elejét mutatom) látható, hogy a optimalizáló sok teljes tábla elérést és tördelésen alapuló összekapcsolást alkalmaz:

```
-------------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                                 |   Name                |   Rows  | Bytes|    Cost    |
-------------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT   STATEMENT                        |                       |     83  |    10K|      318  |
|  SORT   ORDER   BY                        |                       |     83  |    10K|      318  |
|    SORT   UNIQUE                          |                       |     83  |    10K|      315  |
|     CONCATENATION                         |                       |         |       |           |
|      MERGE   JOIN                         |                       |     26  |     3K|      104  |
|       SORT   JOIN                         |                       |    236  |    27K|      102  |
|        HASH   JOIN   OUTER                |                       |    236  |    27K|       96  |
|         HASH   JOIN   OUTER               |                       |    236  |    22K|       92  |
|          HASH   JOIN                      |                       |    236  |    21K|       87  |
|           HASH   JOIN   OUTER             |                       |     48  |     4K|       61  |
|            HASH   JOIN                    |                       |     48  |     3K|       59  |
|             HASH   JOIN                   |                       |    140  |     7K|       28  |
|              HASH   JOIN                  |                       |    141  |     3K|       13  |
|               NESTED   LOOPS              |                       |    141  |     2K|        7  |
|                TABLE   ACCESS   FULL      |SWO_REL_ACAD           |      2  |    10 |        1  |
|                TABLE   ACCESS   FULL      |REL_STUDENT_DEPTSTR    |     5K  |    68K|        3  |
|               TABLE   ACCESS   FULL       |STUDENT_DETAILS        |     5K  |    52K|        3  |
```

Észrevehető az is, hogy a becsült költség mértéke drasztikusan lecsökkent, 25 millióról 318-ra, ugyanakkor ez nem járt a végrehajtási idő drasztikus csökkenésével.

Tovább segítendő a költség alapú optimalizáló munkáját a táblák mezőit is analizáltattam. Először csak a fontosabbnak tűnő mezőket, majd az összes mezőt. Az analizálás sajnos jelentősen rontott a helyzeten, a korábbi idő körülbelül tízszerese alatt sem futott le a lekérdezés, így többször is meg kellett futás közben állítanom.

A SWOMAN Csomag

Az a tény, hogy az összes mező analízisekor kapott végrehajtási terv sokkal egyszerűbb mint a korábbi végrehajtási terv (körülbelül fele annyi sort tartalmaz), és az, hogy a becsült költség nagyon alacsony, arra utal, hogy a lekérdezés tartalmaz olyan részt amivel a költség alapú optimalizáló nem törődik, és ez vezeti félre.

A lekérdezést megvizsgálva a legesélyesebb ilyen rész a SWOMAN csomag függvényének meghívása. Próbaként a lekérdezésből kivettem ezt a részt, és lefuttattam a szabály alapú és költség alapú optimalizálást használva a lekérdezéseket. Az eredmények megerősítették a sejtést, szabály alapú lekérdezéssel 3.09, költség alapú lekérdezéssel 0.29 másodperc kellett a futtatáshoz. Vagyis egyrészt az idő nagy része a csomag függvényének kiértékelésével telik el, másrészt a költség alapú optimalizáló a csomag nélküli lekérdezést jelentősen javítja. Ez egyrészt egy nagyságrendnyi különbség, azonban az eredeti futási időhöz képest lényegében elhanyagolható a közel három másodpercnyi gyorsulás.

A csomag függvényét megvizsgálva azt tapasztaltam, hogy a függvény (ami egy újabb lekérdezést tartalmaz) elég gyors (<0.1 s), és a lekérdezés eredményéül kapott 82 sor mindegyikére végrehajtva sem okozhatja a nagymértékű lassulást. A lassulás oka az, hogy mivel DISTINCT -et használunk, a függvény nem 82 alkalommal fut le, hanem ennél lényegesen többször, annyiszor, ahány sor kapnánk, ha nem lenne

DISTICT . A mi esetünkben ez 5812 sor, vagyis több mint ötezerszer feleslegesen fut le az eljárás.

Ezt — az utólag nyilvánvalónak tűnő — dolgot felismerve könnyen átalakítható a lekérdezés egy új view használatával:

```
CREATE VIEW test_swoman_nelkul1 as
SELECT DISTINCT p.last_name, p.first_name,
p.public_id, cc.country_name,
ssi.sten_id, ssi.period_id,
NVL(sft.ftype_name,' ') ftype_name,
NVL(sft.ftype_id,0) ftype_id
FROM swo_rel_acad sra,
rel_student_deptstream rsd,
swo_student_items ssi,
student_details sd,
persons p,
country_codes cc,
swo_student_types sst,
swo_financial_types sft,
swo_active_student_periods sasp
WHERE sra.swo_stream_id=39 AND
rsd.dstream_id=sra.dstream_id AND
ssi.sten_id=rsd.sten_id AND
rsd.sten_id=sasp.sten_id AND
sd.stud_id=rsd.stud_id AND
p.person_id=sd.person_id AND
cc.country_id(+)=p.citizenshi p_id AND
sst.sten_id(+)=ssi.sten_id AND
sft.ftype_id(+)=sst.ftype_id
ORDER BY ftype_name DESC, p.last_name, p.first_name

SELECT DISTINCT last_name, first_name,
public_id, country_name,
sten_id, NVL(ftype_name,' ') ftype_name,
NVL(ftype_id,0) ftype_id,
Swoman.total_huf_nocash_nonam e(st en_id ,per iod_i d)
total_huf
FROM test_swoman_nelkul1
```

Szabály alapú optimalizálást használva a futási idő így lényegesen alacsonyabb:

```
Parse 0.02 (Elapsed) 0.00 (CPU)
```

```
Execute/Fetch        7.35   (Elapsed)        0.58   (CPU)
Total                7.37                    0.58
```

Áttérve költség alapú optimalizálásra nem változik lényegesen a futási idő.

```
Parse                2.25   (Elapsed)        0.00   (CPU)
Execute/Fetch        5.99   (Elapsed)        0.55   (CPU)
Total                8.24                    0.55
```

Tapasztalatok

A lekérdezés elemzése során több tapasztalatra is szert tehettünk:

- Óvakodjunk a `DISTINCT` és függvények együttes használatától.

  Az első megfigyelés egy egyszerű hibára hívja fel figyelmünket, a hibát elkerülve a lekérdezés sebessége nem lassult volna le ennyire.

- View-k optimalizálásának veszélye

  Bár ebben a példában nem okozott gondot, hogy egy view optimalizálása miatt költség alapú optimalizálásra tért át a rendszer, mivel a szóban forgó rendszer alapvetően szabály alapú optimalizálást tartalmaz, gyakran használt view-knál kerülni kell az ilyen optimalizálást.

- A költség alapú optimalizáló nem foglalkozik a függvényekkel

  Mivel az optimalizálás célja a lassú lekérdezések javítása, és a lassú lekérdezések gyakran összetettek és függvényeket hívnak, ha a függvények végrehajtásához szükséges idő jelentős, akkor az optimalizáló könnyen eltéveszti a célt.

- Mezők analízise drasztikus sebességromláshoz vezethet

  Sajnos a teszt példa volt arra is, hogy mezők analízisével gyakran több kárt okozunk mint hasznot.

4.10.2. Bonyolult elemzés

A következő lekérdezés is egy riportban található, és túl lassúnak bizonyult.

```
SELECT DISTINCT p.last_name, p.first_name,
p.public_id, p.citizenship_id,
NVL(sft.ftype_name,'      ') ftype,
sit.template_name,     ssi.item_value,     so.office_name,
p2.last_name    || ', ' || p2.first_name    coord_name
FROM swo_rel_acad    sra,
```

```
rel_student_deptstream       rsd,
swo_student_items      ssi,
swo_item_templates      sit,
swo_offices    so,
student_details     sd,
persons   p,
swo_coords    sc,
persons   p2,
swo_student_types      sst,
swo_financial_types      sft
WHERE  sra.swo_stream_id=13      AND
ssi.period_id=27     AND
rsd.dstream_id=sra.dstream_id         AND
ssi.sten_id=rsd.sten_id       AND
sit.template_id=ssi.template_        id  AND
so.office_id=ssi.office_id        AND
sd.stud_id=rsd.stud_id       AND
p.person_id=sd.person_id       AND
sc.coord_id=ssi.coord_id       AND
p2.public_id=sc.public_id       AND
sst.sten_id(+)=ssi.sten_id       AND
sft.ftype_id(+)=sst.ftype_id
ORDER  BY  ftype  DESC,  p.last_name,     p.first_name,
so.office_name,sit.template_n        ame
```

A 13 és 27 természetesen paraméterek, amiket a riportot használó Java program állít be. A lekérdezés futási idejét megmérve a következő eredményt kapjuk:

```
Parse               227.59   (Elapsed)        0.00   (CPU)
Execute/Fetch         0.61   (Elapsed)        0.07   (CPU)
Total               228.20                    0.07
```

Vagyis az eredményből az látszik, hogy a lekérdezés valódi végrehajtási ideje (0.61 s) elhanyagolható az elemzés idejéhez képest (227.59 s). A lekérdezési tervet megvizsgálva láthatjuk, hogy mi okozza a jelenséget:

```
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| Operation                                |   Name                |   Rows  | Bytes|    Cost    |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
| SELECT   STATEMENT                       |                       |     17M|     11G|       29M|
|  SORT  ORDER  BY                         |                       |     17M|     11G|       29M|
|   SORT  UNIQUE                           |                       |     17M|     11G|       14M|
|    HASH  JOIN                            |                       |     17M|     11G|    47089   |
|     TABLE  ACCESS  FULL                  |PERSONS                |      7K|    277K|       10  |
|     HASH  JOIN                           |                       |    229K|    148M|      846  |
|      TABLE  ACCESS  FULL                 |PERSONS                |      7K|    412K|       10  |
|      HASH  JOIN                          |                       |      2K|      1M|      103  |
|       HASH  JOIN  OUTER                  |                       |    122  |     71K|       81  |
|        HASH  JOIN                        |                       |    122  |     54K|       73  |
|         TABLE  ACCESS  FULL              |SWO_OFFICES            |     82  |     11K|        1  |
|         HASH  JOIN                       |                       |    149  |     46K|       69  |
|          TABLE  ACCESS  FULL             |SWO_ITEM_TEMPLATES     |     82  |      3K|        1  |
|          HASH  JOIN  OUTER               |                       |    182  |     49K|       66  |
|           HASH  JOIN                     |                       |     74  |     18K|       62  |
|            TABLE  ACCESS  FULL           |SWO_REL_ACAD           |      1  |     26  |        1  |
|            HASH  JOIN                    |                       |      7K|      1M|       36  |
|             HASH  JOIN                   |                       |    129  |     27K|       22  |
|              TABLE  ACCESS  FULL         |SWO_COORDS             |     82  |      1K|        1  |
|              TABLE  ACCESS  FULL         |SWO_STUDENT_ITEMS      |    157  |     29K|       19  |
|             TABLE  ACCESS  FULL          |REL_STUDENT_DEPTSTR    |      5K|     67K|        3  |
|            TABLE  ACCESS  FULL           |SWO_STUDENT_TYPES      |    246  |      6K|        1  |
|          TABLE  ACCESS  FULL             |SWO_FINANCIAL_TYPES    |     82  |     11K|        1  |
|         TABLE  ACCESS  FULL              |STUDENT_DETAILS        |      2K|     50K|        3  |
-----------------------------------------          -------  -------  ------  -------  -------  ------  -
```

A költség alapú optimalizáló túlbecsüli a költségeket, ezért a szükségesnél lényegesen több időt tölt az optimalizálással. Az általa talált legjobb költség 25 millió, a beolvasott adatmennyiséget 11GB-nak becsüli, ami nagyságrendekkel nagyobb a valódi adatmennyiségnek.

Mivel a hibás becslés oka a hiányos statisztika, két megoldás közül választhatunk. Egyrészt pótolhatnánk a szükséges statisztikákat, ebben az esetben a költség alapú optimalizáló várhatóan pontosabb becslésekkel tudna dolgozni, másrészt valamilyen tipp segítségével meggátolhatnánk az optimalizáló túlzásba vitt munkáját.

Mivel rendszerünkben alapvetően szabály alapú optimalizálást használunk, a második megoldást választottam, és a `RULE` hint segítségével szabály alapú optimalizálásra kényszerítettem az optimalizálót.

A módosított lekérdezés futási ideje több mint két nagyságrenddel kisebb. Mint a számokból látszik a változás legfőbb oka az elemzés felgyorsulása.

```
49 rows selected.
Parse                   0.01  (Elapsed)       0.01  (CPU)
Execute/Fetch           0.72  (Elapsed)       0.44  (CPU)
Total                   0.73                  0.45
```

A végrehajtási terven látszik, hogy valamivel egyszerűbb szerkezetű mint az előző esetben, és az is, hogy a szabály alapú optimalizáló inkább a beágyazott ciklusú összekapcsolást választja, szemben a korábbi tördelésen alapulóval.

Operation	Name	Rows	Bytes	Cost
SELECT STATEMENT				
SORT ORDER BY				
SORT UNIQUE				
NESTED LOOPS				
NESTED LOOPS				
NESTED LOOPS				
NESTED LOOPS				
NESTED LOOPS OUTER				
NESTED LOOPS OUTER				
NESTED LOOPS				
NESTED LOOPS				
NESTED LOOPS				
NESTED LOOPS				
TABLE ACCESS FULL	SWO_STUDENT_ITEMS			
TABLE ACCESS BY INDEX R	REL_STUDENT_DEPTSTR			
INDEX UNIQUE SCAN	PK_REL_STDS			
TABLE ACCESS BY INDEX RO	STUDENT_DETAILS			
INDEX UNIQUE SCAN	PK_STUDENT_DETAILS			
TABLE ACCESS BY INDEX ROW	PERSONS			
INDEX UNIQUE SCAN	PK_PERSONS			
TABLE ACCESS BY INDEX ROWI	SWO_REL_ACAD			
INDEX UNIQUE SCAN	AK_DSTREAM_ID			
TABLE ACCESS BY INDEX ROWID	SWO_STUDENT_TYPES			
INDEX RANGE SCAN	PK_SWO_STUDENT_TYPE			
TABLE ACCESS BY INDEX ROWID	SWO_FINANCIAL_TYPES			
INDEX RANGE SCAN	PK_SWO_FINANCIAL_TY			
TABLE ACCESS BY INDEX ROWID	SWO_COORDS			
INDEX RANGE SCAN	PK_SWO_COORDS			
TABLE ACCESS BY INDEX ROWID	PERSONS			
INDEX UNIQUE SCAN	UK_PUBLIC_ID			
TABLE ACCESS BY INDEX ROWID	SWO_OFFICES			
INDEX RANGE SCAN	PK_SWO_OFFICES			
TABLE ACCESS BY INDEX ROWID	SWO_ITEM_TEMPLATES			
INDEX RANGE SCAN	PK_SWO_ITEM_TEMPLAT			

5. fejezet

Összefoglalás

A vizsgált adatbázis-szerver alapvetően szabály alapú optimalizálást (2.4 rész) használ. A lekérdezések optimalizálására jelenleg a következő módokat használják:

- A kérdéses lekérdezések tábláinak elemzése.

  Ez jelenleg csak a táblák kisebb részét érinti. Bár a módszer hatására az optimalizálandó lekérdezések felgyorsulnak, az elemzett táblákat használó többi lekérdezés gyakran lassabbá válik.

- A lekérdezések átírása.

  Gyakran van szükség arra, hogy a túl lassú lekérdezéseket más — az eredetivel kompatíbilis — formára alakítsák át. A próbálgatáson alapuló módszer során jónéhány hasznos heurisztikát használnak.

A tanulmány tesztjei azt mutatják, hogy a szerver lekérdezéseinek gyorsítása érdekében költség alapú optimalizálásra (2.3. rész) lenne szükséges áttérni. Pusztán a táblák elemzése jelentős sebességnövekedést okozna, és lehetőséget nyújtana fejlettebb optimalizálási módszerek használatára.

A táblák elemzése óhatatlanul lelassítani bizonyos lekérdezéseket, ezeknél legegyszerűbben a `/*+ RULE */` tipp használatával visszatérhetnénk a szabály alapú optimalizálásra. A későbbiek folyamán ezeket a lekérdezéseket egyenként megvizsgálva várhatóan teljesen eliminálható lenne a szabály alapú optimalizáló.

A statisztikák frissen tartása érdekében a táblákat éjjelente újra kell elemezni. Ha ehhez túl sok erőforrásra lenne szükség, a táblák elemzése történhet egy hosszabb időtartam alatt elosztva.

A tesztek azt sugallják, hogy bizonyos esetekben az indexek átszervezése is javíthatná a lekérdezéseket. Az adatbázis tervező eszköz által generált B-fa indexek (1.2.1. rész) egy részének törlése, illetve bittérkép indexre (1.2.3. rész) cserélése javasolt. Mivel az indexek generálása során nincsenek figyelembe véve a lekérdezések, manuális

úton kell az egyes lekérdezéseket felgyorsító összetett indexeket (1.2.3. rész) létrehozni.

Klaszterek használata nem ajánlott, a tesztek azt mutatják, hogy az adatbázis mérete illetve szerkezete miatt jelentős sebességnövekedést nem okoznának.

Irodalomjegyzék

[1] Jennifer Widom Hector Garcia-Molina, Jeffrey D. Ullman. *Adatbázisrendszerek megvalósítása*. Panem Könyvkiadó, 2001.

[2] Donald E. Knuth. *A számítógép programozás művészete*, volume 3. Műszaki könyvkiadó, 1994.

[3] Oracle. *Concepts Release 2 (8.1.6)*. December 1999. A76965-01.

[4] Oracle. *Designing and tuning for Performance Release 2 (8.1.6)*. December 1999. A76992-01.

[5] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems*. Computer Science Press, 1988.