Lekérdezés-feldolgozás és optimalizálás

Segédanyag az Adatbázisok c. tárgyhoz

Gajdos S. 2008.

Az anyag a „relációs lekérdezések feldolgozása és optimalizálása” gazdag témakörének csak egy viszonylag kis, de alapvető részét fedi le. Nem foglalkozunk a lekérdezés fordításával és szintaktikai ellenőrzésével, a dinamikus programozás alkalmazásával, adaptív technikákkal, és még számos más kapcsolódó problémával sem. Az anyag csupán egy bevezetést kínál a témakörbe kifejezetten a BSc 3. MI Adatbázisok kurzus hallgatói számára.

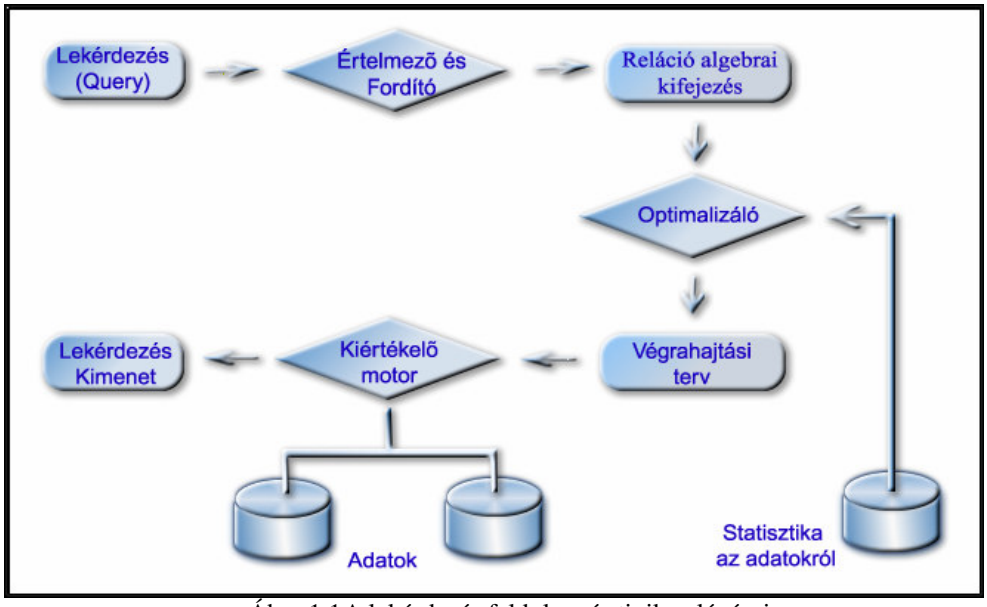
Tartalom

1. Áttekintés

A lekérdezés feldolgozás elsődleges célja az adatok adatbázisból való kinyerése. Az egyértelműen megfogalmazott célhoz a megoldás megtalálása azonban korántsem egyszerű. Számos igen bonyolult részfeladatot kell az adatbáziskezelő-rendszernek megoldania, amíg eljut a kívánt végeredményhez. Ezek a következők:

1. Elemzés (szintaktikus), fordítás
2. Költség optimalizálás
3. Kiértékelés

Az első lépés az elemzés (szintaktikus) és a fordítás. Egy magas szintű nyelven (tipikusan az SQL valamilyen dialektusában) megfogalmazott kérést a számítógép számára használhatóbb formába kell hozni. Az SQL-t, mint kommunikációs interfészt az emberi igényekhez tervezték, minden SQL mondat megfeleltethető egy természetes nyelv (speciálisan az angol) egy mondatának. Sajnálatos módon azonban a számítógép ezt az idegen nyelvet csak tolmács segítségével beszéli. A legtöbb adatbáziskezelő-rendszer anyanyelve a relációs algebra kiterjesztett változatára épül.



Ábra 1.1 A lekérdezés feldolgozás tipikus lépései

Persze a fordítás csak akkor valósulhat meg, ha az SQL nyelvet a felhasználó helyesen beszéli, ezért a fordítást minden esetben megelőzi egy szintaktikai elemzés. Az elemzésnél megvizsgáljuk a lekérdezés szintaktikáját: pl. meggyőződünk arról, hogy a lekérdezésben szereplő relációnevek ténylegesen előfordulnak-e az adatbázisban, stb. A lekérdezés elemeit ez után le kell fordítani és valamilyen belső – általában reláció algebra alapú – reprezentációba átalakítani.

Miután sikerült egy matematikailag helyes kifejezést konstruálnunk az SQL mondatból, érdemes pár dolgot végiggondolni. Vajon egyértelmű-e egy lekérdezés, ill. a hozzárendelt belső reprezentáció a kiértékelés sorrendjét tekintve? Lehetséges-e formális módszerekkel a lekérdezésünkkel ekvivalens másik lekérdezés(eke)t konstruálnunk? Mert ha lehetséges, akkor nyilvánvalóan több végrehajtási út létezik, amelyek sebességben, végrehajtási időben, lemezhasználatban stb.-ben esetleg eltérnek egymástól. Ebben az esetben érdemes lenne

különböző végrehajtási terveket is kidolgoznunk. Ha már kidolgoztunk több tervet, akkor ezeket valamilyen szempont szerint össze kellene hasonlítani. Nyilvánvalóan az optimális megoldást keressük a problémára, de mi az a paraméter, amely alapján az optimalitást értelmezzük? Esetleg nem egy, hanem több paramétert is számításba kell venni? Mivel bizonyosan lesznek eleminek tekintett műveleteink, vajon milyen algoritmusok léteznek a végrehajtásukra, és melyiket válasszuk? Összefoglalva: optimalizációs stratégiák alapján végrehajtási terveket kell készíteni és kiértékelni, majd ezekből a legjobbat kiválasztani és végrehajtani.

A hálós és a hierarchikus modellben a lekérdezés optimalizálás legtöbbször a programozó feladata. Ennek oka, hogy az adatmanipulációs nyelven megfogalmazott kifejezések általában a programokba beágyazva szerepelnek és ezeknek optimális hálós illetve hierarchikus lekérdezéssé transzformálása az egész program ismerete nélkül általában nem megoldható. Az optimalizációhoz tehát rendkívül bonyolult algoritmus futtatására lenne szükség, amely túl nagy terhelést jelentene a rendszer számára – különösen az akkori rendszerek számára.

Ezzel szemben a relációs algebrai modellekben az optimalizáció megvalósítható (jelentős részben a deklaratív szemlélet miatt) a programozó közbeavatkozása nélkül is. Mivel a dominánsan deklaratív szemléletű mondatok elsősorban nem azt mondják meg, hogy hogyan hajtsunk végre valamit, hanem leginkább azt, hogy mit szeretnének eredményül kapni, ezért a belső megvalósítás rejtve maradhat. A deklaratív elvek realizálása algebrai eszközökkel megoldható. Köszönhetően a formális matematikai módszereknek viszonylag könnyű lesz egyazon kérdéshez több, eredményét tekintve ekvivalens alakot találni, és kiválasztani közülük a legkevésbé költségeset.

A továbbiakban megnézzük, hogy az algebrai formák hogyan támogatják a lekérdezések optimalizálását. Formális lépéseken keresztül megkeressük egy adott SQL mondat több különböző ekvivalens alakját. Az eltérő alakok eltérő kiértékelési sorrendet jelenthetnek. Megpróbáljuk megbecsülni, melyik végrehajtása mekkora terhelést jelentene a rendszer számára. Egy egyszerű példával illusztrálva a lépéseket nézzük az alábbi SQL mondatot:

select *balance*
from *account*
where *balance < 2500*

Egy bank nyilvántartásából szeretnénk megtudni, milyen 2500 egységnél kisebb értékű egyenlegek léteznek. A lekérdezés átalakíthatjuk például a következő két relációalgebrai alakba:

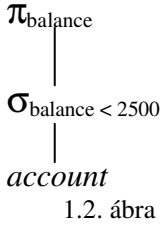
$$\pi_{balance}(\sigma_{balance<2500}(account))$$

$$\sigma_{balance<2500}(\pi_{balance}(account))$$

Láthatóan a két alak két különböző végrehajtási sorrendet kínál. Az első esetben az *account* relációból kiválasztjuk azokat az elemeket, amelyben az egyenleg értéke kisebb 2500-nál, majd ezután végrehajtunk egy projekciót az így kapott relációra. A második forma pont a fordított sorrendet írja le, először egy projekció, majd egy szelekció műveletet kell elvégezni. Ezután el kell döntenünk, melyik végrehajtási sorrendet kövessük.

Az optimum mértéke lehet a lekérdezés teljes (fiktív) "költsége", amelynek a kiszámításához szükséges az egyes eleminek tekintett műveletek költségeinek ismerete. Sajnos a helyzet feloldása nem ennyire egyszerű, mert egy elemi művelet költsége más és más eltérő környezetekben. Tekintsük például a szelekció műveletet, amelynek végrehajtási ideje, ha lineáris keresést alkalmazunk arányos a reláció összes elemének számával, azonban ha valamilyen indexet állítottunk a szelekció feltételére a költséget nagyságrenddel csökkenthetjük. Érezhetően nem mindegy, hogy milyen algoritmusokat tudunk alkalmazni az elemi operációk vagy *kiértékelési primitívek* végrehajtásánál. A primitívek összefoghatóak egy nagyobb munkafolyamati egységbe, egy *pipeline*-ba. A csővezetékben lévő egyik operáció bemenetét az előtte álló primitív kimenete szolgáltatja. A primitív feldolgozza a bemenetét, majd a kimenetét

átadja a sorban utána álló műveletnek. A primitív műveletek szekvenciája a *lekérdezési terv*. Az 1.2-es ábra a példánkra mutat be egy lehetséges tervet.



1.2. ábra

A különböző tervek megalkotása még önmagában nem elegendő, hiszen meg kell mondani, hogy az egyes elemi műveletek végrehajtásához pontosan milyen algoritmusokat alkalmazunk. A lehetséges tervek közül az optimális megtalálása nem egyszerű feladat.

2. Katalógus költségbecslés

A lekérdezés feldolgozáshoz a megfelelő végrehajtási stratégia kiválasztása valamilyen költségmérték ***becsült*** hozzárendelése alapján végezhető. A becslés tényét nem lehet eléggé hangsúlyozni, nem szabad és nem is érdemes egzakt számokat várni a költségbecslés eredményétől. Az adatbázis-kezelő rendszernek a relációkról különböző statisztikákat, mérőszámokat kell karbantartania, amelyek alapján elvégezhetőek a költségbecslések. A statisztikákat az adatbázis-kezelők egyéb más rendszerleíró paraméterek mellett egy ú.n. katalógusban tárolják.

2.1. katalógusban tárolt egyes relációkra vonatkozó információk:

n_r : az r relációban levő rekordok (elemek) száma (**n**umber)

b_r : az r relációban levő rekordokat tartalmazó blokkok (**b**locks) száma

s_r : egy rekord nagysága (**s**ize) byte-okban

f_r : mennyi rekord fér egy blokkba (blocking **f**actor)

V (A, r): hány különböző értéke (**V**alues) fordul elő az A attribútumnak az r relációban: $V(A, r) = |\pi_A(r)|$; speciálisan ha az A kulcs, akkor $V(A, r) = n_r$.

SC(A, r) : azon rekordok átlagos száma, amelyek kielégítenek egy egyenlőségi feltételt az A attribútumra (**S**election **C**ardinality); feltéve, hogy legalább egy rekord kielégíti ezt az egyenlőségi feltételt. Például, ha az A egyediséget biztosít, akkor SC(A,r) = 1. Ha az A nem biztosít egyediséget, és feltesszük, hogy a V (A, r) különböző érték egyenletesen oszlik el a rekordok között, akkor $SC(A,r) = n_r/V(A,r)$.

Az utóbbi két mennyiség definiálható tetszőleges A attribútumhalmazra is: V (A, r) illetve SC(A, r).

Megfigyelés: Ha a relációk rekordjai fizikailag együtt vannak tárolva, akkor: $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$

2.2. Katalógus információk az indexekről

Az indexek – leggyakrabban B* fák – olyan segédstruktúrák, amelyek gyorsíthatják a relációkban adott attribútum vagy attribútumok szerinti keresést. Indexek létrehozása a rendszer számára nagyobb adminisztrációs költséggel jár, ami a használatuk során térülhet meg. Az egységes tárgyalás érdekében itt a hash-táblákat is speciális „indexnek" tekinthetjük.
Az indexeket az alábbi paraméterekkel fogjuk jellemezni:

f_i: az átlagos pointer-szám a fa struktúrájú indexek csomópontjaiban, mint pl. a B* fáknál, azaz a csomópontokból induló ágak átlagos száma

HT_i: az i index szintjeinek a száma, azaz az index magassága B* fáknál (Height of Tree)
$HT_i = \left| \log_{f_i} V(A,r) \right|$, ill. hash-indexnél $HT_i = 1$.

LB_i: az i index legalsó szintű blokkjainak a száma, azaz a levélszintű indexblokkok száma (Lowest level index Block)

A statisztikákat elvileg minden adatbázis módosító művelet után módosítani kellene, de ez óriási terhelést jelente a rendszer számára. A valós alkalmazásokban a statisztika felülírása ezért csak akkor történik meg, amikor a rendszernek „van rá ideje". Ebből következően a statisztika nem mindig konzisztens a rendszer állapotával, de elég jól leírja a rendszerben zajló folyamatokat. Tehát egy régebbi statisztika alapján nem is végezhetünk pontos költségszámítást, de egy elfogadható becslésnek jó kiindulási alapja lehet.

2.3. A lekérdezés költsége

A bevezetőből kiderül, hogy különböző kiértékelési tervek végrehajtási költsége más és más. Definiáljuk most pontosabban, hogy *költség* és *optimalitás* alatt mit kell érteni. A lekérdezés kiértékelés költségének meghatározása történhet az igényelt és a felhasznált erőforrások alapján, ez lehet például a felhasznált processzoridő, a háttértárhoz fordulás ideje vagy elosztott rendszerekben a kommunikációra fordított idő stb. Egy másik kézenfekvő lehetőség a válaszidő alapján történő költség-becslés. Azonban, ha jobban végiggondoljuk ez nem is olyan jó alternatíva, hiszen a válaszidő erősen függ a környezet állapotától is (egy túlterhelt rendszer valószínűleg lassabban generálja le a végeredményt, mint egy kevésbé leterhelt). A válaszidő érezhetően nem biztosít elemi költség-meghatározási szempontot.

A nagy adatbáziskezelőkben a **költség becslésére a háttértár blokkműveletek számát használják,** mivel ez lényegében független a rendszer terhelésétől és mert ennek időigénye nagyságrenddel nagyobb, mint a processzor- és memóriaműveletek időigénye. A használható költségmérték megalkotásához azonban szükséges a probléma megfelelő szintű egyszerűsítése. Nem szabad különbséget tennünk az egyes blokkok elérési ideje között, azaz alapfeltételezés, hogy a diszken elhelyezkedő minden blokkhoz azonos idő alatt férünk hozzá. Nem vesszük figyelembe a lemez forgási irányát, a fej mozgását; ezeket nem tudjuk megbecsülni, és nem tudunk különbséget tenni az egyes írások és olvasások között sem. Ez alapján legyen a *költség* a diszk blokkok olvasásának és írásának a száma azzal a további megszorítással, hogy *az írásba csak a köztes blokkírások számát számítjuk bele*, hiszen a végeredmény kiírása mindenképpen szükséges.

Jelölés: E_{alg} = az algoritmus becsült költsége (estimate) .

3. Műveletek költsége

A relációs adatbázis-kezelők világában szükséges a relációkon végezhető alapműveletek definiálása, ezek lesznek azok az elemi építőkövek, amelyek segítségével minden más lekérdezés megszerkeszthető. Jelen anyagban csak az alapesetekkel foglalkozunk.

3.1. Szelekció

A lekérdezés feldolgozásban egy reláció végigolvasása a legalacsonyabb szintű művelet. Egy adott érték megtalálását az adott szelekciós feltételek figyelembevétele mellett valamilyen keresési algoritmus alapján kell elvégezni. A relációk egyszerű esetben egyetlen állományban tárolódnak, ezért a keresési műveletet csupán egy fájlra korlátozódik.

3.1.1. Alap szelekciós algoritmusok

A szelekció művelet megvalósítását lehetővé tevő két alapalgoritmus a következő:

A1: Lineáris keresés („full table scan”): Minden rekordot beolvasunk, és megvizsgáljuk, hogy kielégíti-e a szelekció feltételét. $E_{A1} = b_r$

A2: Bináris keresés. Bináris keresést csak akkor tudunk végrehajtani, ha a blokkok folyamatosan helyezkednek el a diszken, a fájl az A attribútum szerint rendezett és a szelekció feltétele az egyenlőség az A attribútumon. Összesen SC(A, r) darab ilyen rekordunk van, ezért az algoritmus költsége: $E_{A2} = \lceil \log_2 b_r \rceil + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil - 1$

(Az első ilyen blokk megtalálása a relációban lévő rekordokat tartalmazó blokkok logaritmusával arányos, az összeg második része a szelekció feltételét kielégítő összes rekord tárolásához szükséges maximális blokkméret. A –1 azért szükséges, mert az első és a második tagja az összegnek már tartalmazza az első megfelelő blokk olvasásának költségét.) Ha az A attribútum egyediséget biztosító attribútum, akkor a keresés költsége: $\lceil \log b_r \rceil$

3.1.2. Indexelt szelekciós algoritmusok

A szakirodalom megkülönbözteti az elsődleges és másodlagos indexeket. Az elsődleges index a rekordok olyan sorrendben való olvasását teszi lehetővé, amely megfelel a rekordok fizikai tárolási sorrendjének. Minden egyéb indexet másodlagos indexnek tekintünk. A legfontosabb indexelt szelekciós algoritmusok ezek alapján a következők:

A3: Elsődleges index használatával, egyenlőségi feltételt a kulcson vizsgálunk. Az algoritmus költsége $E_{A3} = HT_i + 1$, az index szintek plusz az adatblokk olvasása.

A4: Elsődleges index használatával egyenlőségi feltétel nem a kulcson. $E_{A4} = HT_i + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil$ Az egyenlőségi feltételt SC(A,r) rekord elégíti ki, amelyhez $SC(A,r)/f_r$ blokkművelet szükséges.

A5: Másodlagos index használatával. $E_{A5} = HT_i + SC(A, r)$ (a második tag mutatja, hogy mennyi különböző blokkon lehetnek). Ha az A egyediséget biztosít, akkor $EA_5 = HT_i + 1$.

3.1.3. Összehasonlítás alapú szelekció

Tekintsük a $\sigma_{A\leq v}(r)$ alakú lekérdezéseket. Ha v értékét nem ismerjük, azt mondhatjuk, hogy átlagosan $n_r/2$ rekord elégíti ki a feltételt. Ha ismerjük a v értékét, és egyenletes eloszlást feltételezünk az A attribútum maximális (max(A,r)) és minimális értéke között (min(A,r)), akkor:

$n_{átlagos} = n_r(\frac{v - \min(A,r)}{\max(A,r) - \min(A,r)})$ rel rekord elégíti ki átlagban a feltételelt.

A6: Elsődleges index használatával. $E_{A6}=HT_i+br/2$ (átlagosan a keresési feltételt a rekordok fele kielégíti). Ha a v-t ismerjük, és c jelöli azon rekordok számát, ahol A≤v, akkor: $E_{A6} = HT_i + \left\lceil \frac{c}{f_r} \right\rceil$

A7: Másodlagos index használatával. $E_{A7} = HT_i + \frac{LB_i}{2} + \frac{n_r}{2}$ A második tag jó becslés, ha a levélszintű indexblokkok legalább fele kielégíti a feltételt. Az utolsó tagra azért van szükség, mert ha a rekordok legalább fele kielégíti a feltételt, akkor ezeket a másodlagos index jellegéből következően csak egyesével, azaz egy blokkművelet költséggel tudjuk elérni.

A másodlagos indexek használatával kapcsolatban meglepő következtetésre juthatunk. Az összehasonlítás alapú lekérdezéseknél szerencsétlen esetben kifizetődőbb egy egyszerű lineáris keresést alkalmazni, mert az kevesebb blokkműveletet igényel.

3.2. Join operáció

A join (illesztés vagy összekapcsolás) művelet általános értelemben két reláció Descartes szorzatának adott feltétel (predikátum) szerinti szelekciója (Theta-join): $R_1 \triangleright\triangleleft_\theta R_2 = \sigma_\theta(R_1 \times R_2)$
A továbbiakban bemutatjuk a legfontosabb join típusokat, megbecsüljük, hogy két reláció illesztéséhez várhatóan mekkora tárterület szükséges, majd áttekintjük a join megvalósítását lehetővé tevő algoritmusokat.

3.2.1. Az illesztés fontosabb típusai:

- ***Természetes összekapcsolás*** (Natural join)
  Két tábla között a megegyező nevű attribútumok létesítenek kapcsolatot. Általában, tekintsük az A és B attribútumhalmazok feletti $R_1(A)$ és $R_2(B)$ sémákat, ahol $X = A \cap B$ nem üres. Az R_1 és R_2 feletti T_1 és T_2 táblák természetes összekapcsolása egy R(A U B) feletti T tábla, amelyet a következőképp definiálunk:
  $$T = \pi_{A\ U\ B}(\sigma_{R1.X=R2.X}(T_1 \times T_2))$$
  Vagyis, a két tábla Descartes-szorzatából kiválasztjuk azokat a sorokat, amelyek az $R_1.X$ és $R_2.X$ attribútumokon megegyeznek, majd a projekcióval a duplán szereplő X-beli attribútumokat csak egy példányban tartjuk meg (az A U B halmazelméleti unió, vagyis benne az X elemei csak egyszeresen szerepelnek).

- ***Külső összekapcsolás*** (Outer join)

A természetes összekapcsolás veszélye, hogy általában a kapcsolt táblák nem minden sora szerepel az eredménytáblában. Ha egy sor nem párosítható a másik tábla egyetlen sorával sem, akkor *lógó sor*nak nevezzük. A *külső összekapcsolás* (outer join) garantálja az összekapcsolt két tábla egyikénél vagy mindkettőnél valamennyi rekord megőrzését. Egy elterjedt implementáció jelölési konvenciója (+) alapján megkülönböztetjük:

- *Bal oldali külső összekapcsolás:* T_1*(+)T_2. Azt jelenti, hogy az eredménytáblában T_1 azon sorai is szerepelnek, amelyek T_2 egyetlen sorával sem párosíthatók. Ezen sorokban a T_2-beli attribútumok értéke NULL.
- *Jobb oldali külső összekapcsolás:* T_1(+)*T_2. Hasonlóan a T_2 táblára.
- *Teljes külső összekapcsolás:* T_1(+)*(+)T_2. Itt mindkét tábla nem párosított rekordjai megőrződnek.

- ***Theta-összekapcsolás*** (theta-join)
  Általános illesztés. A táblák Descartes-szorzatából tetszőleges feltétel szerint választunk ki sorokat: $T = \sigma_{\text{feltétel}} (T_1 \times T_2)$)

3.2.2. Nested-loop join (egymásba ágyazott ciklikus illesztés)

Az egymásba ágyazott ciklikus illesztés egy általános algoritmus két reláció (r és s) theta-join műveletének implementálására. Az algoritmus logikája könnyen végigkövethető a megadott pszeudokód alapján.

```
FOR minden t_r ∈ r rekordra DO BEGIN
    FOR minden t_s ∈ s rekordra DO BEGIN
        teszteljük (t_r, t_s ) párt, hogy kielégíti-e a θ-join feltételt
        IF igen, THEN adjuk a t_r.t_s rekordot az eredményhez
    END
END
```

(ahol a . művelet a konkatenációt jelöli)

Láthatóan ez egy elég költséges eljárás, mert minden egyes t_r,t_s párt külön megvizsgál. A legrosszabb esetben n_r*b_s+b_r blokkműveletre van szükségünk a teljes algoritmus lefuttatására (az r reláció végigolvasása b_r blokkművelet, egy r-beli rekordhoz az összes s-beli blokk végignézése b_s blokkművelet). Ha a két reláció befér a memóriába, akkor b_r+b_s blokkműveletre van szükség a beolvasáshoz. Ha a memória csupán az egyik reláció tárolását teszi lehetővé, akkor is b_r+b_s lesz a költség. Legyen az algoritmus szerinti s reláció az, amely elfér a memóriában, olvassuk be s-et (b_s költség), így minden r-beli rekordhoz az összehasonlítást gyorsan, azaz költség nélkül megtehetjük, ehhez járul még az r-beli rekordok beolvasási költsége.

3.2.3. Block nested-loop join (blokkalapú egymásba ágyazott ciklikus illesztés)

A blokkalapú egymásba ágyazott ciklikus illesztés algoritmus okosabb, mint az első algoritmusunk, mert kihasználja a tárolás fizikai sajátosságait. A gyorsítást azáltal éri el, hogy blokkalapú rekord-összehasonlítást végez. A belső két ciklus összehasonlítja a két reláció egy-egy beolvasott blokkjának minden rekordját, a külső kettő pedig végigmegy a két reláció összes blokkján.

```
FOR minden b_r ∈ r blokkra DO BEGIN
    FOR minden b_s ∈ s blokkra DO BEGIN
        FOR minden t_r∈ b_r rekordra DO BEGIN
            FOR minden t_s ∈ b_s rekordra DO BEGIN
                teszteljük le a (t_r,t_s) párt
            END
        END
    END
END
```

Az algoritmus „worst-case” költsége $b_r * b_s + b_r$, kedvezőbb esetben (az első algoritmusnál bemutatott gondolatmenet alapján) $b_r + b_s$.

3.2.4. Indexed nested-loop join (Indexalapú egymásba ágyazott ciklikus illesztés)

Az indexelt egymásba ágyazott ciklikus illesztés algoritmus kihasználja, hogy az egyik relációhoz van indexünk. Ha az első esetben bemutatott algoritmus belső ciklusába az indexelt relációt tesszük, akkor nem szükséges minden egyes s-beli rekordot végigvizsgálnunk, hogy megfelel-e a feltételnek, hiszen a keresés index alapján kisebb költséggel is elvégezhető. Az eljárás költsége $b_r + n_r * c$, ahol c a szelekció költsége s-en, amely nyilván a konkrét indexstruktúra függvénye.

3.2.5. Merge join (Összefésülés alapú illesztés)

Az illesztés úgy is elvégezhető, ha mindkét relációt először rendezzük az illesztési feltételnek megfelelő attribútum értékeinek megfelelően. Ez után már elég csak egyszer-egyszer végigolvasni mindkét relációt, hiszen az illeszkedő elemek a rendezés következtében egymás után kerültek. Az ilyen módon végzett illesztés költsége b_r+b_s+a rendezések költsége. Ha a relációk igen nagyok, és hatékonyan tudunk rendezni, akkor gyakran a legkisebb költségre vezet.

Megjegyzés: A join megvalósítására számos egyéb algoritmus is létezik.

.

3.3. Egyéb műveletek

A két legfontosabb művelet, a select és a join bemutatása után pár mondatban kitérünk egyéb gyakran használt műveletekre is.

- *Ismétlődés kiszűrése:* (Ha ugyanabból a rekordból több példány van, akkor csak egy maradjon) Először rendezést hajtunk végre. Az azonos rekordok közvetlenül egymás után fognak megjelenni, ekkor már könnyen törölhetők. Költség: a rendezés költsége.

- *Projekció*: Minden rekordra végrehajtjuk, aztán kiküszöböljük a másodpéldányokat a fenti módszerrel. Ha a rekordok eleve rendezettek, akkor a költség b_r, általános esetben b_r+ a rendezés költsége.
- *Unió*: Először mindkét relációt rendezzük, majd összefésülésnél kiszűrjük a duplikációkat.
- *Metszet*: Mindkét relációt rendezzük, az összefésülésnél csak a közös rekordokat vesszük figyelembe.
- *Különbség*: Mindkét relációt rendezzük, összefésülésnél csak azok a rekordok maradnak, amelyek csak az első relációban szerepelnek.

4. Kifejezés kiértékelés

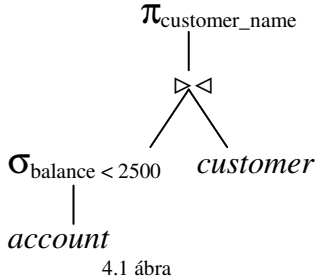
Áttekintettük az elemi műveletek néhány algoritmikus megvalósítását. Nem foglalkoztunk azonban még az összetett, több elemi műveletből álló kifejezések kiértékelésének módjával.

A legkézenfekvőbb stratégia, hogy az összetett kifejezésnek egyszerre egy műveletét értékeljük ki valamilyen rögzített sorrend szerint (materializáció). Ezzel azonban van egy nagy probléma, minden művelet végrehajtása után a keletkezett eredményt a későbbi felhasználás miatt a háttértárra kell kiírni, ezért a módszer rengeteg blokkműveletet igényel. Egy másik kiértékelési alternatíva: egy „csővezetékben" egyszerre több elemi művelet szimultán kiértékelése folyik, egy művelet eredményét – a háttértár bevonása nélkül – azonnal megkapja a sorban következő művelet operandusként (pipelining).

4.1. Materializáció (megtestesítés, létrehozás)

A $\pi_{customer\_name}(\sigma_{balance<2500}(account) \triangleright\triangleleft customer)$ kifejezést a műveletek mentén a 4.1 ábra szerinti műveleti fába transzformálhatjuk.

A fa leveleiben vannak a relációink, a belső csomópontokban pedig a műveletek. A fa alapján a materializációs stratégia lépései könnyen nyomon követhetőek. Az első lépésként hajtsunk végre egy olyan műveletet, amelyekhez az operandusaink rendelkezésre állnak. Ez a példánkban csak a szelekciós műveletre teljesül. Az így kapott ideiglenes relációt ezután illesszük a *customer* relációval, majd hajtsuk végre a projekciót.



4.1 ábra

A stratégia minden relációt kiszámít a fában, közben létrejönnek közbülső relációink is. A materializáció költsége tehát a végrehajtott operációk költségének összege, plusz a közbülső relációk tárolásának a költsége.

4.2. Pipelining

A kiértékelés hatékonysága növelhető, ha redukáljuk az ideiglenesen tárolásra kerülő rekordok számát. Ha megszervezünk egy olyan munkafolyamatot, amelyben a részegységek az előttük álló elemtől kapott részeredményekből a sorban következő számára állítanak elő részeredményeket, akkor kiküszöbölhetjük az ideiglenes tárolás szükségességét. A fenti példán illusztrálva a lépéseket, mindhárom relációt egy pipeline-ba tesszük. A szelekció eredményét azonnal átadjuk a join-nak, nem számítjuk ki előre az egész relációt. További előnye, hogy kicsi a memóriakövetelmény, mert az eredményeket nem tároljuk sokáig, hanem továbbadjuk. Hátránya, ha nincs közbülső reláció, nem tudunk rendezni sem (nem történik materializáció).
A feldolgozás vezérlése alapján kétféle pipeline-t különböztetünk meg: igényirányított és termelő-irányított.

Az igényirányított esetben maga a rendszer fordul a csővezeték tetejéhez és kér rekordokat. Minden alkalommal, ha a csővezeték megkapja ezt a kérést, akkor kiszámítja és átadja a rendszernek.

Termelőirányított pipeline esetén a csővezeték mentén elhelyezkedő műveletek nem várnak kérésre. A csővezeték legalsó szintjén minden művelet folyamatosan generálja a rekordokat, és egy pufferbe teszi, amíg a puffer meg nem telik (ugyanígy tesz minden szint). Minden szinten minden művelet „egymástól függetlenül” dolgozik.

4.2.1. Pipeline kiértékelési algoritmus

Tekintsük a példánkban azt a join műveletet, amelynek baloldala csővezetéken érkezik. Az egész baloldali reláció nem áll rendelkezésre, a rekordok egyenként jönnek, ezért nem használhatjuk az összes JOIN algoritmust (pl. *merge-join* rendezés alapú illesztési algoritmus, nem alkalmazható, ha a baloldali input nincs rendezve az illesztési attribútumok szerint). Az indexelt nested-loop join azonban használható, mert ahogy beérkeznek a baloldali rekordok, az illesztési attribútum-értékek alapján a jobboldali reláció feletti index segítségével kikeressük a jobboldali relációból az illeszkedő rekordokat és összeillesztjük őket egymással.

5. Relációs kifejezések transzformációi

Láttuk az elemi műveletek és egy adott lekérdezés kiértékelésének lépéseit. A bevezetőben azonban említettük, hogy egy adott lekérdezéshez több formális algebrai alakot is konstruálhatunk, amelyek mind más és más költséggel hajthatók végre.

5.1. Ekvivalens kifejezések

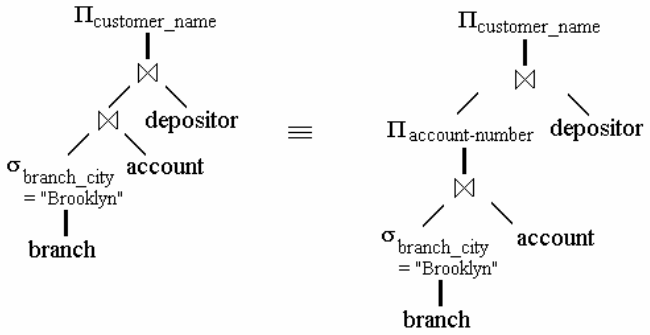
Tekintsük a következő természetes nyelven megfogalmazott kérdést. „Add meg azoknak a fogyasztóknak a nevét, akiknek van számlájuk Brooklyn-ban!”.

$$\Pi_{customer\_name}(\sigma_{branch-city="Brooklyn"}(branch \triangleright\triangleleft (account \triangleright\triangleleft depositor))$$

A fenti relációs algebrai alak végrehajtása rengeteg erőforrást pazarolna, hiszen három reláció összekapcsolása után végezné csak el a szelekciót. Egy sokkal hatékonyabb alak a következő:

$$\Pi_{customer\_name}((\sigma_{branch-city="Brooklyn"}(branch)) \triangleright\triangleleft (account \triangleright\triangleleft depositor))$$

A második forma takarékosabban bánik az erőforrásainkkal, először kiválasztja a *branch* relációból a Brooklyn-ban lévő fiókokat, majd csak ezt illeszti a maradék kifejezéssel. A kezdeti és az átalakított kifejezés műveleti fáját mutatja az 5.1-es ábra.



A legkisebb költségű végrehajtási terv megtalálása a lekérdezés optimalizáló feladata. Az optimalizáló első teendője adott lekérdezéshez ekvivalens algebrai alakok megtalálása. Majd az algebrai alakokhoz alternatív végrehajtási tervet kell készítenie.

5.2. Ekvivalencia szabályok

Az ekvivalens algebrai alakok legenerálásához az optimalizálónak szüksége van olyan szabályokra, amelyek mentén a feladat algoritmikusan elvégezhető, most ezeket a szabályokat vesszük sorra. Jelölések: θ, θ_1, θ_2 predikátumok, L_1, L_2, L_3 attribútumok, E, E_1, E_2 relációs algebrai kifejezés.

1. Szelekció kaszkádosítása:
$$\sigma_{\theta 1 \wedge \theta 2}(E)=\sigma_{\theta 1}(\sigma_{\theta 2}(E))$$
2. A szelekció kommutativitása:
$$\sigma_{\theta 1}(\sigma_{\theta 2}(E))=\sigma_{\theta 2}(\sigma_{\theta 1}(E))$$
3. Projekció kaszkádosítása:
$$\prod_{L1}(\prod_{L2}(\ldots(\prod_{Ln}(E))\ldots)) =(\prod_{L1}(E))$$
4. Az illesztés és a Descartes szorzat kapcsolata:
$$\sigma_{\theta}(E_1 \times E_2) = E_1 \rhd\lhd_{\theta} E_2$$
$$\sigma_{\theta 1}(E_1 \rhd\lhd_{\theta 2} E_2) = E_1 \rhd\lhd_{\theta 1 \wedge \theta 2} E_2$$
5. A theta-join kommutativitása:
$$E_1 \rhd\lhd_{\theta} E_2 = E_2 \rhd\lhd_{\theta} E_1$$
6. A természetes illesztés asszociativitása:
$$(E_1 \rhd\lhd E_2) \rhd\lhd E_3 = E_1 \rhd\lhd (E_2 \rhd\lhd E_3)$$

7. A szelekció művelet disztributivitása a join felett
-Ha a θ_0 csak E1-beli attribútumokat tartalmaz:
$$\sigma_{\theta 0}(E_1 \rhd\lhd_{\theta} E_2) = \sigma_{\theta 0}(E_1) \rhd\lhd_{\theta} E_2)$$

8. A projekció disztributív a theta-join felett
-Ha L_1, L_2 E_1 illetve E_2-beli attribútumokat tartalmaz és a join feltételében csak $L_1 \cup L_2$-beli attribútumok vannak.
$$\Pi_{L1 \cup L2}(E_1 \rhd\lhd_{\theta} E_2) = (\Pi_{L1}(E_1)) \rhd\lhd_{\theta} (\Pi_{L2}(E2))$$

-Az L_1 és L_2 az E_1 illetve E_2-beli attribútumok halmaza. Az L_3 olyan E_1-beli és az L_4 pedig olyan E_2-beli attribútumok halmaza, amely nincs benne L_1 és L_2 uniójában. A join feltételében csak L_3 és L_4-beli attribútumok lehetnek.

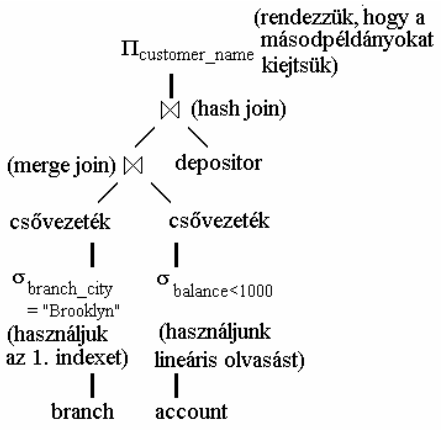
$$\Pi_{L1\cup L2}(E_1 \triangleright\triangleleft_\theta E_2) = \Pi_{L1\cup L2}((\Pi_{L1\cup L3}(E_1)) \triangleright\triangleleft_\theta (\Pi_{L2\cup L4}(E_2)))$$

9. A metszet és az unió kommutativitása
$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$
10. Az unió és a metszet asszociativitása
$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$
11. A szelekció disztributív az unió, a metszet és a különbség műveletek felett
$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2) = \sigma_P(E_1) - E_2$$
12. A projekció disztributív az unió művelet felett
$$\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$$

A felsorolt szabályok nem tartalmazzák az összes ekvivalenciaszabályt, hanem csak ízelítőt adnak belőlük. Számos egyéb, nem csak alapműveletekre kihegyezett átalakítási lehetőség is létezik.

6. A kiértékelési terv kiválasztása

Az ekvivalens kifejezések generálása csak az első lépése az optimalizálásnak. A második lépésben minden egyes kifejezéshez konkrét algoritmusokat kell rendelnünk. Meg kell mondanunk, hogy a műveleteket milyen sorrendben, milyen algoritmus szerint, milyen munkafolyamatba szervezve hajtjuk végre. Ennek egy grafikus alakban megadott példa reprezentációját mutatja a 6.1-es ábra.



6.1 ábra

6.1. Költségalapú optimalizálás

A fordító az előző fejezetben látott azonosságok alkalmazásával először felsorol minden, az eredeti kifejezéssel ekvivalens kifejezést (véges sokat). Ezután minden kifejezéshez hozzá tudunk rendelni kiértékelési tervet. A megfelelő kiértékelési terv kiválasztásának folyamata tulajdonképpen a költség-optimalizálás. Minden kiértékelési tervre kiszámítjuk a költséget és kiválasztjuk a legolcsóbbat (becslések, statisztikák alapján). Hátránya, hogy túl sokféle kiértékelési terv lehet, amely rengeteg munkát ró a rendszerre. Tekintsünk például három reláció illesztését. Három relációt hatféleképpen állíthatunk sorrendbe, és ezt még be kell szorozni 2-vel attól függően, hogy a zárójelet az első kettő relációhoz vagy a második kettőhöz tesszük (ne

feledjük, hogy a végeredmény ugyan minden esetben azonos lesz, azonban a join elvégzésének módja/erőforrásigénye még két reláció esetén sem azonos akkor, ha a sorrendjüket felcseréljük). Általános esetben n reláció join-ja (2*(n-1))!/(n-1)! különböző ekvivalens alakot jelent (és ebben nincs is benne az algoritmus hozzárendelés). Ez már kis n esetén is rengeteg ekvivalenst produkál, pl. n=7 esetén 665280-at, n=10 esetén már több, mint 176 billiót!

Szerencsére azonban nincs is szükség az összes ekvivalens kiértékelésére. Némi heurisztikával kezelhetőbb méretűvé csökkenthetjük a problémateret. Tekintsük a következő kifejezést:

$$(r_1 \triangleright\triangleleft r_2 \triangleright\triangleleft r_3) \triangleright\triangleleft r_4 \triangleright\triangleleft r_5$$

Keressük meg először azt az optimális alakot, amely csak az első három relációt illeszti, majd utána az eredményt a maradék kettő relációval illeszti. Az első három reláció illesztésére 12 lehetőség adódik, majd az eredmény és a maradék két reláció illesztése ismét 12 lehetőséget kínál. Ha értelmetlenül minden verziót kipróbálunk, akkor összesen 144 lehetőséget nézünk végig. Ha azonban először megkeressük az első három reláció optimális kiértékelését és a 4. és az 5. relációval már ezt az optimális eredményt illesztjük, akkor a kiértékelés csupán 12+12 lépést próbálgat végig.

A bemutatott algoritmus egyik legnagyobb problémája, hogy elméletileg is rossz, mert nem minden esetben képes megtalálni az optimális megoldást. Ennek oka, hogy az algoritmus mohó lévén mindig a lokálisan legjobb megoldást választja, és nem mérlegeli azt, hogy egy kicsit rosszabb lokális megoldás globálisan esetleg jobb eredményre vezetne. Sajnos, nem létezik olyan algoritmus, amely kis komplexitás mellett képes az optimális megoldást legenerálni, ezért be kell érnünk szuboptimális megoldásokkal.

6.2. Heurisztikus optimalizálás

A költségalapú optimalizálás legnagyobb hátránya magának – mint láttuk - az optimalizációs algoritmusnak a költsége. A legtöbb kereskedelmi rendszer ezért valamilyen heurisztikát használ a megfelelő kifejezés kiválasztásához. Egy heurisztika alapú optimalizációs stratégia szabályai lehetnek például a következők:

1) Végezzük el a szelekciót olyan korán, ahogy csak lehet, mert ezzel csökkentjük a sorok számát (bizonyos esetekben ez ugyan növeli a költséget, de általában csökkenti). A szabály alapján az optimalizáló olyan ekvivalens átalakításokra fog törekedni, amelyben legbelül lesznek a szelekciós műveletek (7-es szabály).
2) Hamar végezzük el a projekciót, mert ezzel csökkenthető a sorok mérete. A szelekciót általában érdemesebb előbb elvégezni, mert a reláció mérete jobban csökken, mint a projekció alkalmazásával, de persze előfordulhat, hogy a projekció redukálja jobban a reláció méretét.
3) Bontsuk szét a szelekciók konjunkcióját szelekciók szorzatára, hogy minden szelekciónak csak egy tényezője legyen. Ez lehetővé teszi, hogy a fában a szelekciókat lefelé vándoroltassuk, ezáltal a közbenső műveletek jóval kevesebb rekordot adjanak végeredményként.
4) A kiértékelési fában vándoroltassuk lefelé a szelekciókat.
5) Határozzuk meg, hogy mely szelekció és join eredményezi a legkisebb (=legkevesebb rekordot tartalmazó) relációkat. Használjuk fel join asszociativitását, alakítsuk át a fát úgy, hogy ezek a szelekciós és join műveletek hajtódjanak végre először.
6) Előfordulhat, hogy a join megegyezik a Descartes-szorzattal. Ha ezt egy szelekció követi, akkor vonjuk össze a kettőt egy join művelettét, így kevesebb rekordot kell generálni.

7) Törjük szét a projekció-listákat. Az egyes vetítéseket lökjük lefelé a fán, amennyire tudjuk. Ehhez új vetítéseket is létrehozhatunk, ha szükséges.
8) Keressük meg azokat a részfákat, ahol a csővezetéket lehet alkalmazni.

A szabályok alkalmazásával több kiértékelési fát kapunk. Ezeknek meghatározzuk a költségét és vesszük közülük a legolcsóbbat.

Mint láttuk a költségalapú és heurisztikus optimalizálás is jelentős terhelést jelent a rendszer számára. Nem elég tehát a legjobb megoldást megtalálni, hanem a keresés költségét is optimalizálni kell. Az az érdekes helyzet adódik, hogy az optimális optimalizáló a saját működését (tehát a kiértékelési terv keresését) és magának a megtalált megoldásnak a végrehajtását kell, hogy optimalizálja.

7. Irodalomjegyzék

[1] Silberschatz, Korth, Sudarshan: *Database System Concepts*, McGraw-Hill 2005.

[2] Joe Hellerstein, Eric Brewer: *Query Processing*, Advanced Topics in Computer Systems, Online

http://www.cs.berkeley.edu/~brewer/cs262/QP.html

[3] George Koch-Kevin Loney: *Oracle 8*, Panem Könyvkiadó, 1999.

[4] Simon Zoltán: *Lekérdezés feldolgozás és optimalizálás*, Oktatási segédanyag, BME TMIT 2002.

Ld. még http://www.informatik.uni-trier.de/~ley/db/dbimpl/qo.html