SQL-KYSELYJEN OPTIMOINNISTA

Niko Jalkanen

17.4.2007

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu –tutkielma

TIIVISTELMÄ

SQL tarjoaa korkean tason deklaratiivisen rajapinnan relaatiotietokannan tietoihin. SQL-kyselyiden käsittelyn vaiheet ovat jäsentäminen, optimointi ja suoritus. Tietokannan optimoija on väline, jonka tehtävänä on löytää optimaalisin suoritussuunnitelma kyselylle. Optimoijalla on yleensä kaksi eri vaihtoehtoa toimia: heuristinen ja tilastollinen. Heuristinen kyselynoptimointi käyttää muunnossääntöjä relaatioalgebran lausekkeen muuttamiseksi tehokkaampaan muotoon uudelleen järjestämällä kyselyn. Kyselypuuna esitetyn relaatioalgebran lausekkeen suoritussuunnitelma sisältää tiedot saantimenetelmistä kyselyn relaatioille sekä algoritmit, joilla relaatioiden operaatiot suoritetaan. Kustannusperustainen kyselynoptimointi perustuu optimoijan arvioon kyselysuunnitelman operaattoreiden tuottamien rivien määristä (ts. kardinaliteeteista). Kustannusperustaisessa optimoinnissa esimerkiksi Oracle 10g -tietokannanhallintajärjestelmä ottaa huomioon I/O- kustannuksen ja CPU-kustannuksen. Optimoija evaluoi eri kustannukset vertaamalla kyselyn suorittamiseen tarvittavaa I/O-operaatioiden kokonaisaikaa sekä tarvittavaa CPU-aikaa. Tehokkuuteen on mahdollista vaikuttaa myös tietokantavastaavan ja ohjelmoijan toimenpiteillä.

Avainsanat: SQL, optimointi, suoritussuunnitelma, vihjeet, relaatioalgebra, tilastotieto

SISÄLLYSLUETTELO

1 Johdanto 1
2 Relaatioalgebran perusoperaatiot ja SQL 3
2.1 Valinta 4
2.2 Projektio 4
2.3 Karteesinen tulo 5
2.4 Yhdiste 6
2.5 Erotus 7
2.6 Liitos 7
2.7 Leikkaus 8
2.8 Alikysely 9
2.9 Ryhmittely 9
3 Kyselynkäsittely 11
3.1 Esimerkki 11
3.2 Kyselynkäsittelyn vaiheet 13
3.3 Kyselyn jäsentäminen 17
3.4 Heuristinen kyselynoptimointi 20
3.5 Kustannusten estimointi relaatioalgebran perusoperaatioille 22
3.5.1 Tietokannan tilastotiedot 23
3.5.2 Valintaoperaatio 24
3.5.3 Liitosoperaatio 28
3.5.4 Projektio-operaatio 33
3.5.5 Joukko-operaatiot 35
3.6 Muita optimointiperiaatteita 36
3.7 Suoritussuunnitelman generointi 38
4 Oraclen suorittama kyselyjen optimointi 40
4.1 Kustannusperustainen optimointi 40
4.2 Tehokkaan SQL:n kirjoittaminen 42
4.2.1 Tehokas where 42
4.2.2 Vihjeiden käyttö 45
4.2.3 Parhaan liitostavan ja järjestyksen valinta 46
4.2.4 Indeksointi 47
4.3 Tietokantavastaavan keinot 48
4.4 Viritysvälineet 49
4.4.1 EXPLAIN PLAN 49
4.4.2 Autotrace 50
4.4.3 SQL Trace ja TKProf 50
5 Yhteenveto 52
VIITELUETTELO 53

1 JOHDANTO

Mitä SQL-kyselyiden optimointi on? Tähän kysymykseen yritetään vastata lyhyesti tässä tutkielmassa, menemättä kuitenkaan liian syvälle optimoinnin maailmaan, koska jo esimerkiksi indeksointi olisi oman tutkielman arvoinen.

Tutkielman luvussa 2 käydään läpi relaatioalgebraa ja sen eri operaatioita. Relaatioalgebra on optimoinnin kannalta tärkeää, koska relaatiotietokannat perustuvat pohjimmiltaan relaatioalgebraan ja sen operaatioihin. Relaatiotietokannoissa on kyselykielenä SQL (Structured Query Language), jonka historia alkaa 1970-luvulta amerikkalaisesta IBM:n tutkimuslaboratoriosta, ja sen ensimmäinen versio SEQUEL on vuodelta 1974. Virallinen SQL-standardi hyväksyttiin vuonna 1986, jota kuitenkin on päivitetty useampaan otteeseen. SQL:stä onkin tullut lähes standardi tietokantojen ylläpidossa.

Seuraavana tulee tarkasteltavaksi kyselyiden käsittely luvussa 3, jossa käydään läpi kyselyn jäsentämisen eri vaiheita, heuristista optimointia ja kustannusten estimointia eri operaatioille. Kysely jakautuu kolmeen vaiheeseen, jotka ovat jäsentäminen, optimointi ja suoritus [CB05]. Optimoinnin tehtävänä on valita mahdollisimman tehokas suoritussuunnitelma. Jäsentämisen päätehtävänä on tarkastaa kyselyn syntaksi ja semantiikka sekä tuottaa lopuksi jäsennyspuu. Heuristisessa kyselyn optimoinnissa käytetään muunnossääntöjä relaatioalgebran lauseen muuttamiseksi tehokkaampaan muotoon. Muunnossääntöjen avulla optimoija muuttaa kyselypuun tehokkaampaan muotoon uudelleen järjestämällä kyselyn [CB05].

Luvussa 4 käsitellään kyselyn optimointia Oraclen tietokannanhallintajärjestelmän kannalta: Kuinka Oraclessa voidaan vaikuttaa kyselyjen tehokkuuteen, optimoida niitä ja vaikuttaa optimoijaan ja mitä apuvälineitä näihin toimintoihin on? Yleensä kyselyn suoritustehokkuutta voidaan parantaa helposti muuttamalla kyselyn rakennetta tehokkaammaksi mm. käyttämällä indeksejä, virkistämällä tilastotiedot ja huolehtimalla resurssien riittävyydestä. Kustannusperustainen kyselynoptimointi perustuu optimoijan arvioon kyselysuunnitelman operaattoreiden tuottamien rivien määristä, ja koska se perustuu arvioihin, se ei aina onnistu halutulla tavalla. Luvussa katsotaan mitä tapoja on Oraclessa näiden oletuksien muuttamiseen ja parantamiseen [IC91].

Lopuksi luvussa 5 tehdään yhteenveto kyselyiden optimointiin liittyvistä asioista ja esitetään joitakin johtopäätöksiä.

2 RELAATIOALGEBRAN PERUSOPERAATIOT JA SQL

Relaatiotaulu, jossa on *k* attribuuttia, on matemaattiselta kannalta *k*-paikkainen *relaatio*, missä taulun rivit listaavat relaatioon kuuluvat *k*-monikot. Tämän vastaavuuden mukaisesti voidaan relaatiotauluja yhdistellä ja niistä voidaan poimia tietoja tavanomaisin joukko-opin operaatioin (relaatioiden yhdiste, leikkaus, karteesinen tulo jne.) Relaatiotaulujen manipuloinnissa käytetyille joukko-opin operaatioille, tai kuten tässä yhteydessä sanotaan 'relaatioalgebralle', on tietty standardoitu esitystapa: relaatiotietokantojen *kyselykieli* SQL [PK02].

SQL ei ole proseduraalinen kieli, joten sen avulla tehtävät kyselyt voidaan määritellä kysymyksellä "mitä" ennemmin kuin "miten". SQL ei myöskään ole pelkästään kyselykieli, vaan se määrittelee ominaisuudet tietokannan rakenteen määrittelyyn ja muuttamiseen, kyselyjen tekemiseen, tietojen lisäämiseen/poistamiseen/muuttamiseen, valtuuksien/turvallisuuden hoitamiseen ja tapahtumankäsittelyn ohjaamiseen. Seuraavaksi tarkastellaan SQL:n perusoperaatioita käyttäen kuvan 2.1 tauluja apuna.

EMP: Query(localhost) | **DEPT: Query(localhost)** | Start Page

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
NULL	NULL	NULL

EMP: Query(localhost) | DEPT: Query(localhost) | Start Page

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17.12.1980 0:0...	800	NULL	20
7499	ALLEN	SALESMAN	7698	20.2.1981 0:00:00	1600	300	30
7521	WARD	SALESMAN	7698	22.2.1981 0:00:00	1250	500	30
7566	JONES	MANAGER	7839	2.4.1981 0:00:00	2975	NULL	20
7654	MARTIN	SALESMAN	7698	28.9.1981 0:00:00	1250	1400	30
7698	BLAKE	MANAGER	7839	1.5.1981 0:00:00	2850	NULL	30
7782	CLARK	MANAGER	7839	9.6.1981 0:00:00	2450	NULL	10
7788	SCOTT	ANALYST	7566	9.12.1982 0:00:00	3000	NULL	20
7839	KING	PRESIDENT	NULL	17.11.1981 0:0...	5000	NULL	10
7844	TURNER	SALESMAN	7698	8.9.1981 0:00:00	1500	0	30
7876	ADAMS	CLERK	7788	12.1.1983 0:00:00	1100	NULL	20
7900	JAMES	CLERK	7698	3.12.1981 0:00:00	950	NULL	30
7902	FORD	ANALYST	7566	3.12.1981 0:00:00	3000	NULL	20
7934	MILLER	CLERK	7782	23.1.1982 0:00:00	1300	NULL	10
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Kuva 2.1 Esimerkeissä käytettävien taulujen sarakkeet ja data

2.1 Valinta

Yksi SQL-kyselykielen perusoperaatio on tietyn ehdon *E* täyttävien rivien *valinta* annetusta relaatiosta (taulusta) *R*. Matemaattisesti kyse on relaatio-operaatiosta σ_E, joka on määritelty seuraavasti:

> $\sigma_E(R)$ = relaatio *R'*, joka sisältää ne *R*:n monikot, joiden kohdalla ehto *E* on voimassa.

SQL:n mukainen operaatio valintalausekkeelle on:

```
SELECT * FROM R WHERE E
```

Haetaan esimerkiksi kaikki kuvan 2.1 Dept –taulun sarakkeet, joilla deptno on suurempi kuin nolla:

```
SELECT * FROM Dept WHERE deptno>0;
```

Tällöin tulosjoukkona palautuu:

```
10 ACCOUNTING     NEW YORK
20 RESEARCH       DALLAS
30 SALES          CHICAGO
40 OPERATIONS     BOSTON
```

2.2 Projektio

Toinen perusoperaatio on tietyn sarakejoukon *S* valinta relaatiosta *R*. Matemaattisesti kyse on relaation *R* *projektiosta* attribuuttijoukolle *S*:

$\pi_S(R)$ = relaatio *R'*, joka sisältää vain joukkoon *S* sisältyvät *R*:n attribuutit.

SQL-kielinen merkintä tälle operaatiolle on (projektiossa siis tuloksesta karsitan toistuvat arvot eli duplikaatit):

```
SELECT DISTINCT S FROM R
```

Valinta- ja projektio-operaatioita voi myös yhdistellä, tuottamalla vaikkapa SQL-kysely:

```
SELECT DISTINCT S FROM R WHERE E
```

joka on relaatioalgebran lausekkeena esitettynä muotoa:

$$\pi_S(\sigma_E(R))$$

Haetaan esimerkiksi kuvan 2.1 Emp-taulun job-sarakkeen arvo kertaalleen:

```
SELECT DISTINCT job FROM Emp;
```

jolloin tulosjoukkona palautuu jokainen kyseisen kentän arvo duplikaatit poistettuna:

```
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

2.3 Karteesinen tulo

Karteesisessa tulossa R x *S* muodostetaan tulosrelaatioon monikoita, kokoamalla yhdeksi monikoksi arvot monikkopareista, joissa parin monikoista ensimmäinen kuuluu relaatioon *R* ja toinen relaatioon *S*. Yhdistetty monikko muodostetaan jokaisesta monikkoparista relaatio-algebran ja SQL:n avulla ilmaistuna:

R x S = Relaatio, jonka rivit ovat kaikki mahdolliset *R*:n ja *S*:n riviparit.

```
SELECT * FROM R,S
```

Esimerkiksi karteesinen tulo, joka tuottaisi 56 riviä kuvan 2.1 tauluille Emp ja Dept, saadaan kyselyllä:

```
SELECT * FROM Dept,Emp;
```

2.4 Yhdiste

Yhdisteen avulla muodostetaan relaatio, joka sisältää kummankin yhdistettävän relaation monikot:

$R \cup S = \{ t \mid t \in R \vee t \in S \}$, missä *R* ja *S* ovat relaatioita ja *t* on relaation *R* tai *S* monikko.

Yhdisteellä voidaan liittää kaksi tai useampia tauluja siten, että lopputulokseen tulee rivejä useasta taulusta allekkain. SELECT-lauseissa on oltava sama määrä haettavia sarakkeita vastaavassa järjestyksessä ja vastaavien sarakkeiden on oltava samaa tietotyyppiä. ORDER BY-käskyjä voi olla vain yksi ja sekin on oltava viimeisenä. UNION estää saman rivin toistumisen tuloksessa (vrt. DISTINCT), jos halutaan säilyttää duplikaatit on käytettävä muotoa UNION ALL. Esimerkiksi jos kuvan 2.1 taulussa Emp olevalle job-sarakkeelle haetaan UNION:lla sama uudelleen:

```
SELECT job FROM Emp UNION  SELECT job FROM Emp;
```

palautuu seuraavat arvot:

```
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

Jos käytetään määrittelyä UNION ALL:

```
SELECT job FROM Emp UNION ALL SELECT job FROM Emp;
```

palautuu kuvan 2.1 job-sarakkeen sisältö kaksi kertaa eli yhteensä 28 riviä.

2.5 Erotus

Erotuksessa tulosrelaatioon otetaan ne relaation *R* monikot, jotka eivat sisälly erotettavaan relaatioon *S*:

$$R - S = \{ t \mid t \in R \wedge t \notin S \}.$$

Esimerkiksi kuvan 2.1 tauluja käyttäen kysely

```
SELECT deptno FROM Dept EXCEPT SELECT deptno FROM Emp;
```

palauttaa arvon 40.

2.6 Liitos

Relaatiotietokannan taulujen tärkein yhdistämistapa on erirakenteisten taulujen liittäminen yhteen attribuuttivertailujen määräämällä tavalla. Matemaattiselta kannalta kyse on operaatiosta, jossa ensin muodostetaan lähtötaulujen R_1 ja R_2 karteesinen tulo, josta sitten

valitaan monikot, jotka toteuttavat halutun ehdon *E*. Usein halutaan vielä valitut rivit projisoida tietylle attribuuttijoukolle *S*:

$\pi_S(\sigma_E(R_1 \times R_2)) = \pi_S(R_1 \rhd\lhd_E R_2)$ = valituista riveistä/ripipareista ainoastaan joukkoon *S* kuuluvat sarakkeet.

SQL-kielessä vastaava liitosoperaatio kirjoitetaan esimerkiksi muodoissa:

```
SELECT S FROM R1, R2 WHERE E
SELECT S FROM R1 JOIN R2 ON E
```

Liitos voi olla theta-, yhtäsuuruus-, ulko- tai luonnollinen liitos. Ulkoliitoksille (vasen ja oikea) on omat symbolit relaatioalgebrassa [CB05].

2.7 Leikkaus

Leikkaus on joukko-opin operaatio, jolla saadaan tulokseksi kahden joukon yhteiset alkiot. Se voidaan esittää erotus-operaation avulla:

$$R \cap S = R - (R - S) = S - (S - R)$$

Esimerkiksi seuraava kysely kuvassa 2.1 oleville tauluille käyttäen saraketta deptno:

```
SELECT deptno FROM Dept INTERSECT SELECT deptno FROM Emp;
```

palauttaa arvot 10, 20 ja 30.

2.8 Alikysely

Alikyselyillä voidaan rajata varsinaista kyselyä käyttämällä toista pääkyselyn sisään kirjoitettua SELECT-lausetta. Sisäkkäisiä alikyselyjä voi olla monta tasoa ja niiden suoritus alkaa alimmalta tasolta. Alikyselyistä voi tulla tuloksena joko vain täsmälleen yksi rivi tai sitten useampia rivejä. Haetaan esimerkiksi kuvan 2.1 Emp-taulusta ne rivit, joiden deptno-sarakkeen arvo on Dept-taulun pienin deptno-sarakkeen arvo:

```
SELECT deptno, ename FROM Emp WHERE deptno=(SELECT MIN(deptno)
FROM Dept);
```

jolloin palautuu tulosjoukko:

```
10 CLARK
10 KING
10 MILLER
```

Saman lauseen voisi esittää myös seuraavalla tavalla:

```
SELECT deptno, ename FROM Emp WHERE deptno IN (SELECT MIN(deptno)
FROM Dept);
```

tai myöskin seuraavalla tavalla:

```
SELECT e.deptno, e.ename FROM emp e, (SELECT MIN(deptno) as
deptno FROM dept) d  WHERE e.deptno=d.deptno;
```

2.9 Ryhmittely

Ryhmittelyllä voidaan nimensä mukaisesti ryhmitellä tulosjoukkoa hakuehtojen mukaan. Esimerkiksi jos halutaan hakea eri ammatissa olevat ryhmiteltynä ammatin mukaan käyttäen kuvan 2.1 taulua Emp:

```
SELECT job, count(*) FROM Emp GROUP BY job;
```

jolloin palautuu tulosjoukko:

```
CLERK              4
SALESMAN           4
PRESIDENT          1
MANAGER            3
ANALYST            2
```

Nykyisin myöskin ryhmittelylle on määritelty relaatioalgebran symbolit [CB05].

3 KYSELYNKÄSITTELY

On olemassa monta vaihtoehtoista tapaa optimoida sama SQL-lause käyttämään mahdollisimman vähän resursseja. Yleensä yritämme pienentää mahdollisimman pieneksi kyselyn kokonaissuoritusajan, joka muodostuu kaikkien kyselyssä olevien yksittäisten operaatioiden suoritusajoista. Kyselyä voidaan optimoida myös siten, että yritetään päästä mahdollisimman suureen rinnakkaisten operaatioiden määrään. Kun kaikkien suoritusvaihtoehtojen laskenta käy mahdottomaksi, pyritään pääsemään mahdollisimman lähelle optimaalista ratkaisua. Tietokannan kyselynoptimoija on väline, joka suorittaa optimoinnin tietokannan sisällä ja sillä on yleensä kaksi eri vaihtoehtoa toimia: heuristinen ja tilastollinen. Luku perustuu pääosin lähteeseen [CB05].

3.1 Esimerkki

SQL tarjoaa korkean tason deklaratiivisen rajapinnan relaatiotietokannan tietoihin, jolloin vaaditaan ainoastaan kyselyn tulokselta vaadittavien ehtojen ilmaiseminen kyselyssä. Yksinkertaisenkin SQL-kyselyn laskemiseksi on yleensä olemassa useita mahdollisia tapoja hakea tarvittava tieto kyselyn relaatioista, liittää relaatiot toisiinsa ja järjestää ne tehokkaimpaan järjestykseen sekä suorittaa muut tarvittavat operaatiot kyselyn laskemiseksi.

Seuraavassa esimerkissä näytetään erilaisten prosessointistrategioiden vaikutusta resurssien käyttöön.

```
SELECT * from Emp e, Dept t WHERE e.Deptno=t.Deptno AND
(e.job='Manager' AND t.loc='BOSTON');
```

Seuraavat kolme vastaavaa relaatioalgebran lauseketta vastaavat edellistä kyselyä:

(1) $\sigma_{(job='Manager') \times (loc='BOSTON') \times (Emp.deptno=Dept.deptno)}(Emp \times Dept)$

(2) $\sigma_{(job='Manager') \times (loc='BOSTON')}(Emp \triangleright\triangleleft_{Emp.deptno=Dept.deptno} Dept)$

(3) $(\sigma_{job='Manager'}(Emp)) \triangleright\triangleleft_{Emp.deptno=Dept.deptno}(\sigma_{loc='BOSTON'}(Dept))$

Oletamme, että tässä esimerkissä on Emp-taulussa 1000 tietuetta ja Dept-taulussa 50 tietuetta. Emp-taulussa on 50 johtajaa, jokainen eri toimistossa ja 5 toimistoista on Bostonissa. Seuraavaksi vertailemme näiden kolmen kyselyn tarvitsemia levyoperaatioita. Yksinkertaisuuden vuoksi oletamme, että indeksejä ei ole, eikä viiteavaimia relaatioiden välillä ja jokainen välitulos tallennetaan levylle. Lopputuloksen kirjoitusta levylle ei ole huomioitu, koska se on kaikissa sama. Oletamme myös, että voimme lukea yhden tietueen kerrallaan ja keskusmuistia on tarpeeksi prosessoimaan kaikki relaatiot.

Ensimmäisessä kyselyssä on karteesinen tulo Emp- ja Dept -taulujen välillä, joka tarvitsee (1000+50) levylukua lukeakseen tiedot ja tekee relaatiot (1000*50) tietueille. Seuraavaksi täytyy lukea jokainen tietue uudestaan ja testata ne kyselyn predikaatteja vasten, joka vaatii toiset (1000*50) levylukua, josta saamme yhteensä:

$$(1000 + 50) + 2 * (1000 * 50) = 101050 \text{ levylukua}$$

Toinen kysely liittää Emp- ja Dept -taulut deptno-sarakkeen perusteella, joka vaatii tietueiden lukuun (1000+50) levylukua. Tiedämme että näiden kahden taulun liitoksen relaatiossa on 1000 tietuetta, yksi jokaiselle Emp-taulun jäsenelle eli valintaoperaatio tarvitsee 1000 levyhakua lukeakseen lopputuloksen, josta saamme yhteensä:

$$2 * 1000 + 1000 + 50 = 3050 \text{ levylukua}$$

Viimeinen kysely lukee ensimmäiseksi Emp-taulun, joka tarvitsee 1000 levylukua löytääkseen johtaja-tietueet ja luo relaation 50 tietueelle. Seuraava valintaoperaatio lukee Dept-taulun löytääkseen Bostonin toimistot, tämä tarvitsee 50 levylukua ja tekee relaation viidelle tietueelle. Viimeisenä operaationa liitetään syntyneet relaatiot (50+5) levylukua, josta saamme yhteensä:

$$1000 + 2 * 50 + 5 + (50 + 5) = 1160 \text{ levylukua}$$

Viimeinen vaihtoehto oli selvästi paras, suhteella 87:1 verrattuna ensimmäiseen kyselyyn. Jos kasvattaisimme Emp-taulun koon 10000 tietueeseen ja Dept-taulun 500 tietueeseen, niin suhde olisi jo noin 870:1. Tästä voimmekin päätellä, että karteesinen tulo ja liitosoperaatio ovat paljon tehottomampia kuin valintaoperaatio, ja kolmas vaihtoehto pienentää relaatioiden kokoa, jotka on liitetty yhteen. Näemme tästä nopeasti, että yksi perusstrategioista kyselyjen prosessoinnissa on suorittaa yksioperandisia operaatioita, kuten valinta ja projektio, mahdollisimman aikaisessa vaiheessa.

3.2 Kyselynkäsittelyn vaiheet

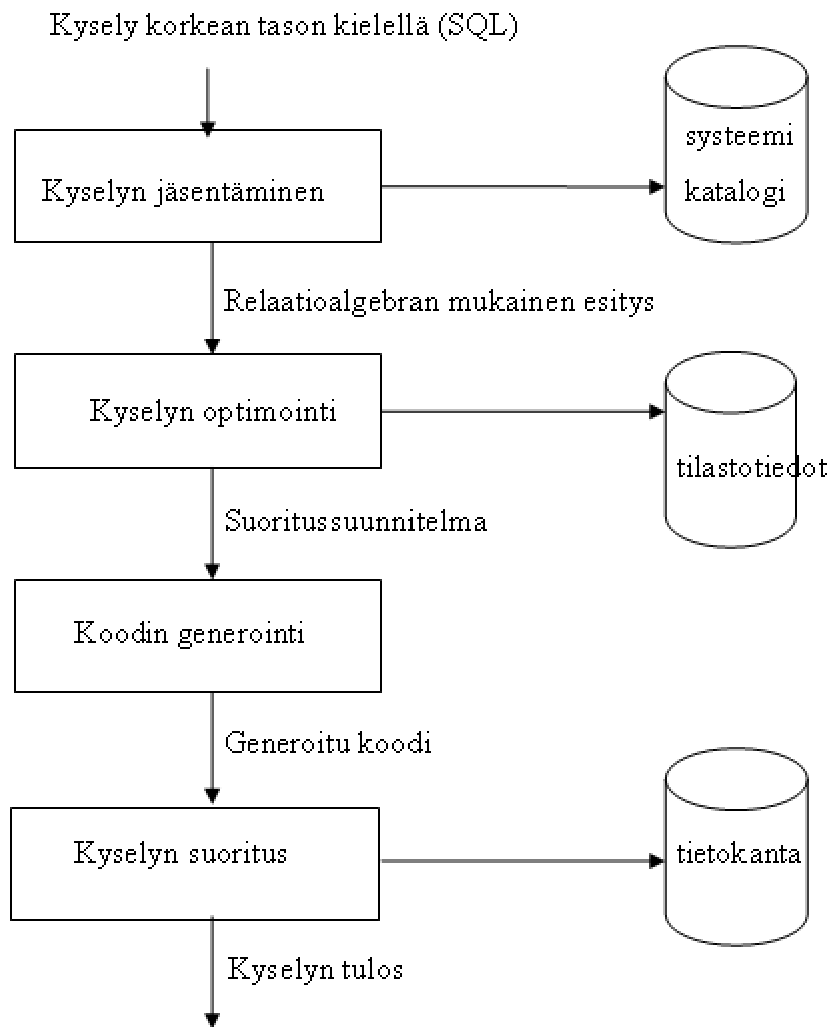
Kyselynkäsittelyn tarkoituksena on muuttaa kirjoitettu kysely oikeaksi ja tehokkaaksi suoritusstrategiaksi, josta tuloksena saadaan tarvittu tietojoukko. Käsittelyn vaiheet ovat jäsentäminen, optimointi, koodin generointi ja suoritus (kuva 3.1).

Kyselynoptimoijan tehtävänä on päättää millä menetelmällä kyselyn tulos saadaan tehokkaimmin lasketuksi. Jäsentäminen muuntaa kyselyn järjestelmän ymmärtämään muotoon. Optimoinnin tehtävänä on valita mahdollisimman tehokas suoritussuunnitelma. Kyselyn relaatioiden määrän kasvaessa suoritussuunnitelmien määrä kasvaa nopeasti suureksi ja siksi tehokkaimman suoritussuunnitelman löytyminen on laskennallisesti vaikea tehtävä.

Kyselynoptimoinnissa käytetyt kustannusfunktiot ovat arvioita, eivät siis tarkkoja ja niinpä optimoija voi virheellisesti valita väärän strategian [EN00].

Optimointi jakautuu kahteen eri osa-alueeseen: yritetään löytää relaatioalgebran keinoin kustannuksiltaan pienin suoritussuunnitelma sekä mahdollisimman tehokkaat algoritmit suorittamaan operaatiot, kuten esimerkiksi indeksi- tai lomitusliitos ja sopivat indeksit. Relaatiotietokannassa voi olla tilastollisia tietoja, jotka helpottavat suoritussuunnitelman valintaa, kuten esimerkiksi tilastotietoja monikoiden lukumäärästä, arvojen jakaumasta ja erilaisten arvojen määrästä. Tilastotietoja käytettäessä on kuitenkin hyvä huolehtia siitä, että ne pysyvät jotakuinkin ajan tasalla, koska vanhentuneet tiedot voivat johtaa virheellisiin optimointiratkaisuihin. Optimoija ei laske kyselylle varsinkaan monimutkaisemmissa

kyselyissä kaikkia eri vaihtoehtoja, vaan käy läpi todennäköisimmät vaihtoehdot, jotka perustuvat esimerkiksi tilastotietoihin, koska kaikkien eri variaatioiden laskeminen olisi liian hidas toimenpide.



Kuva 3.1 Kyselynkäsittelyn vaiheet [CB05]

Tarkastellaan kyselyä:

```
SELECT e.ename,e.sal FROM Emp e WHERE e.sal>1000 and e.sal<4000;
```

Tässä kyselyssä ei ole liitosten järjestämisongelmaa, koska kysely viittaa vain yhteen relaatioon. Optimoija voi kuitenkin harkita useampia eri tapoja laskea kysely:

- Kysely voidaan laskea tauluselauksella, jolloin relaation Emp kaikki tietosivut käydään läpi, luetaan sivun kaikki monikot *t* ja tulostetaan kyselyn vastaukseen ehdot e.sal>1000 ja e.sal<4000 täyttävistä monikoista *t* attribuuttien ename ja sal arvo.
- Jos relaatioon Emp on indeksi jollakin tai joillakin attribuuteista ename ja sal, kysely voidaan laskea indeksiselauksella.
- Muita tapoja laskea kysely on B-puuselaus.

Kustannusperustaisessa optimoinnissa kunkin suoritussuunnitelman laskentakustannus arvioidaan matemaattisesti varsinaista tietokantaa tutkimatta ja suoritettavaksi valitaan suunnitelma, jonka kustannus on pienin. Laskentakustannus riippuu pitkälti niiden monikoiden lukumäärästä, jotka suoritussuunnitelman eri operaattorit käsittelevät. Optimoija arvio jokaisen operaattorin tuottaman kardinaliteetin. Kardinaliteettien arviointi alkaa tietokannan tilastotiedoista, jotka on tallennettu ennalta tietokannan tietohakemistoon (data dictionary). Tyypillisiä tilastotietoja ovat relaatioiden kardinaliteetti ja relaation attribuuttien kardinaliteetti tai arvojakauman approksimaatio. Valitsevuudelle johdetaan arvio valinta-attribuutin ja relaation tilastotiedoista.

Heuristinen optimointi puolestaan tapahtuu sääntöjen pohjalta eli käytössä on joukko erilaisia sääntöjä, joiden pohjalta operaatiot järjestetään ja suoritustavat valitaan. Heuristisessa optimoinnissa ei käytetä apuna taulu tai tilastotietoja. Sääntöjen pohjalta generoidaan useita vaihtoehtoisia suoritussuunnitelmia ja jokaiselle vaihtoehdolle annetaan kustannusarvio ja pienimmän kustannuksen saanut suunnitelma valitaan. Säännöissä ROWID-haku on paras ja indeksiselaus huonoin. Heuristinen optimoija on poistumassa Oraclesta uusien versioiden myötä [Bur01].

Kustannusperusteisessa optimoinnissa tilastotietojen ajantasaisuus on yksi optimoijan tärkeimmistä apuvälineistä optimoinnin teossa. Jos tietohakemistossa olevat tiedot ovat vanhentuneita, niin optimoija voi tehdä optimoinnin väärin. Tietohakemistossa olevat

tilastotiedot kuvaavat tietokannan nykytilaa. Tilastotietoja ei aina kuitenkaan voida pitää tarkasti ajan tasalla, sillä esimerkiksi relaatioiden kardinaliteetin kasvattaminen jokaisen monikon lisäämisen jälkeen aiheuttaisi kahden eri monikon konfliktoinnin asianomaisessa tietohakemistomonikossa. Tilastotietojen jatkuva ylläpito rasittaisi myös palvelimen prosessointikykyä ja aiheuttaisi täten viiveitä muihin kyselyihin sekä myös tilastotietojen laskemiseen. Näistä syistä tietokantavastaava voi ajaa aika ajoin tilastotiedot.

Arvojakaumien tasaisuus olettaa, että attribuutin jokainen eri arvo on yhtä todennäköinen, ts. jokaista eri arvoa esiintyy relaatiossa yhtä monta. Yksittäisen relaation yksittäisen attribuutin epätasaista arvojakaumaa voidaan approksimoida tietohakemistoon tallennettavalla histogrammilla. Histogrammi jaottelee arvot pylväisiin arvonsa mukaan, jolloin samat arvot kasvattavat arvoalueensa pylvästä. Histogrammi tarjoaa parannetun valittavuuden tiedolle, joka johtaa parempaan optimoinnin lopputulokseen. Histogrammin apu tulee parhaiten esille, kun arvojoukko on suuri ja arvot jakautuvat isolle arvoalueelle. Yksilöivälle (unique) tiedolle histogrammista ei ole apua.

Predikaattien riippumattomuus olettaa, että kyselyn eri valintaehtojen valitsevuudet arvioidaan toisistaan riippumattomasti ja valitsevuudet kerrotaan keskenään. Eri valinta-attribuutit voivat kuitenkin riippua toisistaan, esim. funktionaalisen riippuvuuden kautta. Yksittäisen relaation attribuuttikombinaatioille voidaan laskea moniulotteinen histogrammi, mikä taaskaan ei ole mahdollista liitospredikaateille. Sitä paitsi monissa tietokannoissa on nykyään satoja attribuutteja käsittäviä relaatioita, mikä tekee mahdottomaksi moniulotteisen historiatiedon tallentamisen kaikille attribuuttikombinaatioille.

Sisältyvyysriippuvuusoletus määritellään [MLR03]:

> liitospredikaatin $r.A = s.B$ valitsevuudeksi arvioidaan tyypillisesti $1/\max\{V(A,r),V(B,s)\}$, missä $V(A,r)$ tarkoittaa attribuutin A kardinaliteettia relaatiossa r.

Tämä arvio sisältää implisiittisesti oletuksen, että jokainen pienemmän arvojoukon arvo sisältyy suurempaan arvojoukkoon. Oletus on kyllä usein perusteltu, esimerkiksi silloin, kun liitos on pääavaimen ja viiteavaimen välinen. Kun edellä mainitut oletukset eivät ole voimassa, tuloksena voi olla huomattavia virheitä kyselyn välitulosten kardinaalisuusarvioissa ja siis myös kyselysuunnitelmien kustannusarvioissa, mikä taas voi johtaa epäoptimaalisen suunnitelman valintaan.

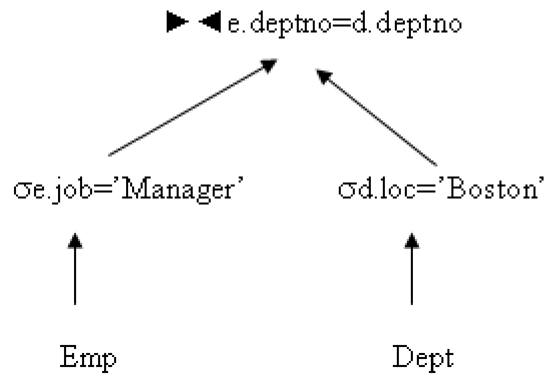
3.3 Kyselyn jäsentäminen

Kyselyn jäsentämisen ensimmäinen vaihe on kyselyn prosessointi. Jäsentämisen tavoitteena on muuttaa korkean tason kielellä, kuten SQL:llä kirjoitettu kysely matalan tason kielelle, relaatioalgebran mukaiseksi lausekkeeksi. Usein kyselyn jäsentäminen jaetaan vaiheisiin, jotka ovat: analysointi, normalisointi, semanttinen analyysi, yksinkertaistaminen, uudelleen järjestäminen. Jäsentämisen päätehtävänä on tarkastaa kyselyn syntaksi ja semantiikka sekä tuottaa lopuksi jäsentämispuu, jossa on kyselyn rakenne [Ala05].

Analysointivaihe tarkastaa kyselyn semanttisuuden ja syntaksisuuden oikeellisuuden ohjelmointikääntäjissä käytettyjen menetelmien avulla. Tässä vaiheessa tarkastetaan, että kyselyssä olevat attribuutit ja relaatiot ovat olemassa ja objektityypit ovat sopivia kyselyssä oleville operaatioille. Tarkastellaan seuraavaksi kyselyä kuvan 2.1 Emp-tauluun kohdistuvaa kyselyä:

```
SELECT empNumber FROM Emp WHERE ename>34;
```

Yllä oleva kysely ei tule onnistumaan kahdesta syystä: ensinnäkään Emp-taulussa ei ole empNumber nimistä saraketta ja ename ei ole numeerista tietotyyppiä. Tämän vaiheen lopuksi kysely on muutettu esitysmuotoon, joka on sopivampi prosessille, tämän sisäiseksi esitystavaksi on yleensä valittu puu (kuva 3.2).



Kuva 3.2 Relaatioalgebra-puu [CB05]

Normalisointivaihe konvertoi kyselyn normalisoituun muotoon, jota on helpompi käsitellä jäsentämisen edetessä. Kysely voidaan konvertoida yhteen kahdesta eri normaalimuodosta käyttäen muutamaa muutossääntöä, ja nämä normaalimuodot ovat konjunktiivinen normaalimuoto ja disjunktiivinen normaalimuoto.

Semanttinen analyysi hylkää normalisoidut kyselyt, jotka on muotoiltu väärin tai ovat ristiriitaisia. Kysely on väärin muotoiltu, jos se ei tuota generoinnille lopputulosta. Näin voi käydä, jos kaikkia liitoksia ei ole määritelty. Kysely on ristiriitainen, jos sen ehdot ovat sellaisia, etteivät ne voi koskaan täyttyä, esimerkiksi:

$$e.job = \text{'Manager'} \wedge e.job = \text{'Secretary'}$$

Nämä ehdot kuvan 2.1 Emp-taulun kyselyssä ovat ristiriitaiset, koska Emp-taulun jäsen ei voi olla sekä Manager että Secretary. Ehto:

$$((e.job = \text{'Manager'} \wedge e.job = \text{'Secretary'}) \vee e.salary > 20000)$$

voidaan yksinkertaistaa muotoon e.salary > 20000, hylkäämällä ristiriitaiset ehdot, jotka palauttavat arvon epätosi.

Yksinkertaistamisen tehtävänä on tunnistaa turhat vaatimukset, eliminoida yleiset ilmaisut ja muuttaa kysely semanttisesti yhtenevään, mutta paljon yksinkertaisempaan ja tehokkaampaan muotoon. Käyttöoikeudet, näkymien määrittelyt ja eheysrajoitteet tarkastetaan tyypillisesti tässä vaiheessa. Jos käyttäjällä ei ole pääsyä kaikkiin kyselyn tietoihin, niin siinä tapauksessa kysely hylätään. Olettaen, että käyttäjällä on oikeus tietoihin, niin ensimmäinen optimointi on muuttaa ehdot Boolen algebran mukaan:

$p \wedge (p) \equiv p$	$p \vee (p) \equiv p$
$p \wedge \text{false} \equiv \text{false}$	$p \vee \text{false} \equiv p$
$p \wedge \text{true} \equiv p$	$p \vee \text{true} \equiv \text{true}$
$p \wedge (\sim p) \equiv \text{false}$	$p \vee (\sim p) \equiv \text{true}$
$p \wedge (p \vee q) \equiv p$	$p \vee (p \wedge q) \equiv p$

Seuraavassa esimerkissä tarkastelemme näkymän määrittelyä ja kyselyä kyseiseen näkymään:

```
CREATE VIEW emp2 AS SELECT empno,ename, sal, deptno FROM Emp
WHERE deptno=20;

SELECT * FROM Emp2 WHERE (Deptno=20 and Sal>20000);
```

Näkymän johdosta kyselystä tulee seuraavanlainen:

```
SELECT empno,ename, sal, deptno FROM Emp WHERE (Deptno=20 and
Sal>20000) AND deptno=20;
```

Lauseen WHERE-osa kutistuu kuitenkin muotoon (Deptno=20 and Sal>20000).

Kyselyn jäsentämisen viimeisenä vaiheena on kyselyn uudelleen järjestäminen. Kyselyn uudelleen järjestäminen tehdään, jotta kysely saataisiin tehokkaampaan muotoon ja tähän palaamme seuraavassa kohdassa.

3.4 Heuristinen kyselynoptimointi

Tässä kohdassa tarkastelemme lähemmin heuristista kyselynoptimointia, jossa käytetään muunnossääntöjä relaatioalgebran lauseen muuttamiseksi tehokkaampaan muotoon.

Muunnossääntöjen avulla optimoija muuttaa kyselypuun tehokkaampaan muotoon uudelleen järjestämällä kyselyn. Sääntöjen esittelyssä (alla) käytetään kolmea relaatiota *R*, *S* ja *T*, joista *R* ylimäärittelee attribuutit $A = \{A_1, A_2, \ldots, A_n\}$, S määrittelee attribuutit $B = \{B_1, B_2, \ldots, B_n\}$; *p*, *q* ja *r* tarkoittavat predikaatteja ja L, L_1, L_2, M, M_1, M_2 *ja* N tarkoittavat attribuuttijoukkoja [CB05].

1. Konjunktiivinen valintaoperaatio erillisiksi operaatioiksi

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

2. Valintaoperaatioiden kommutatiivisuus

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

3. Peräkkäisistä projektioista ainoastaan viimeinen tarvitaan

$$\Pi_L \Pi_M \ldots \Pi_N(R) = (\sigma_q(R)) = \Pi_L(R)$$

4. Valinnan ja projektion kommutatiivisuus

$$\Pi_{A1\ldots Am}(\sigma_p(R)) = \sigma_q(\Pi_{A1\ldots Am}(R)) \text{ , missä } p \in \{A_1, A_2, \ldots, A_m\}$$

5. Liitoksen (ja karteesisen tulon) kommutatiivisuus

$$R \rhd\lhd_p S = S \rhd\lhd_p R$$

$$R \times S = S \times R$$

6. Valinnan ja liitoksen (tai karteesisen tulon) kommutatiivisuus

$\sigma_p(R \rhd\lhd_r S) = (\sigma_p(R)) \rhd\lhd_r S$

$\sigma_p(R \times S) = (\sigma_p(R)) \times S$, missä $p \in \{A_1, A_2, \ldots, A_n\}$

7. Projektion ja liitoksen (tai karteesisen tulon) kommutatiivisuus

$\Pi_{L1 \cup L2}(R \rhd\lhd_r S) = (\Pi_{L1}(R)) \rhd\lhd_r (\Pi_{L2}(S))$

8. Yhdisteen ja leikkauksen kommutatiivisuus (ei erotuksen)

$R \cup S = S \cup R$

$R \cap S = S \cap R$

9. Valinnan ja joukko-operaatioiden kommutatiivisuus

$\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$

$\sigma_p(R \cap S) = \sigma_p(S) \cap \sigma_p(R)$

$\sigma_p(R - S) = \sigma_p(S) - \sigma_p(R)$

10. Projektion ja yhdisteen kommutatiivisuus

$\Pi_L(R \cup S) = \Pi_L(S) \cup \Pi_L(R)$

11. Liitoksen (ja karteesisen tulon) assosiatiivisuus

$(R \rhd\lhd S) \rhd\lhd T = R \rhd\lhd (S \rhd\lhd T)$

$(R \times S) \times T = R \times (S \times T)$

12. Yhdisteen ja leikkauksen assosiatiivisuus (ei erotus)

$$(R \cup S) \cup T = S \cup (R \cup T)$$

$$(R \cap S) \cap T = S \cap (R \cap T)$$

Heuristisessa optimoinnissa käytetään seuraavanlaista algoritmia [EN00]:

1. Suoritetaan valintaoperaatiot mahdollisimman aikaisin; käytetään sääntöä 1 muuttamaan valintaoperaation ehdot erillisiksi. Säännöillä 2, 4, 6 ja 9 painetaan valinnat niin alas kyselypuussa kuin mahdollista: valinnan vaihdannaisuus.
2. Yhdistetään karteesinen tulo ja seuraava valinta liitokseksi.
3. Järjestetään lehtisolmut sääntöihin 11 ja 12 perustuen.
    a. Suoritetaan ensin mahdollisimman rajoittavat valinnat:
        i. tulos pienin (rivejä tai tavuja)
        ii. pienin valitsevuus (selektiivisyys)
    b. Vältetään karteesisen tulon muodostusta (yleensä).
4. Painetaan projektio-operaatiot niin alas puussa kuin mahdollista; Käytetään sääntöä 3 poistamaan ylimääräiset projektio-operaatiot ja sääntöjä 4, 7, 10 painamaan projektio-operaatiot puussa niin alas kuin mahdollista. (Operaatioiden tarvitsemat attribuutit on kuitenkin säilytettävä)
5. Yhdistetään operaatiot (alipuu), jotka voidaan suorittaa yhdellä algoritmilla.

3.5 Kustannusten estimointi relaatioalgebran perusoperaatioille

Tietokannanhallintajärjestelmissä voi olla useita erilaisia tapoja määritellä relaatioalgebran operaatiot ja kyselyn optimoinnin tarkoituksena on valita niistä tehokkain. Löytääkseen tehokkaimman operaation hallintajärjestelmä käyttää kaavaa, joka arvioi kustannuksia eri vaihtoehdoille ja valitsee niistä pienimmän. Tässä kohdassa käsitellään erilaisia vaihtoehtoja, joita on käytettävissä relaatioalgebran perusoperaatioiden toteutukseen. Yleensä hallitsevin kustannus kyselyn prosessoinnissa on levyoperaatiot, koska ne ovat paljon hitaampia kuin

muistioperaatiot. Seuraavissa kohdissa esitellään operaatioiden toteutuksia ja niiden kustannuksia, jotka sisältävät arvioidut levyoperaatioiden määrät, pois lukien lopputuloksen kirjoituksen levylle. Monet kustannusarvioista perustuvat relaation kardinaliteettiin. Tarkemmat algoritmit löytyvät esimerkiksi lähteestä [CB05].

3.5.1 Tietokannan tilastotiedot

Relaatioalgebran välitulosten kustannusarvioiden onnistuminen riippuu tilastotietojen määrästä ja ajantasaisuudesta. Tyypillisesti voimme olettaa, että tietokannanhallintajärjestelmä säilyttää vähintäänkin seuraavia tietoja systeemikatalogissa.

Jokaiselle perusrelaatiolle *R*

- nTuples(R) – Monikkojen määrä relaatiossa *R* (kardinaliteetti)
- bFactor(R) – Relaation *R* lohkokerroin (monikkojen määrä, joka mahtuu yhteen lohkoon)
- nBlocks(R) – Lohkojen määrä, joka tarvitaan tallentamaan relaatio *R*. Jos lohkot on tallennettu peräkkäin: nBlocks(R) = [nTuples(R)/bFactor(R))

Käytetään [x] merkitsemään sitä, että laskennan lopputulos on pyöristetty lähimpään kokonaislukuun, joka on suurempi tai yhtä suuri kuin x.

Jokaiselle perusrelaation *R* attribuutille *A*

- $nDistinct_A(R)$ – Attribuutti *A*:n erilaisten arvojen määrä relaatiossa *R*
- min_A (R), $max_A(R)$ – Suurin ja pienin mahdollinen arvo attribuutille *A* relaatiossa *R*
- $SC_A(R)$ – Valinnan kardinaliteetti (selection cardinality) attribuutille *A* relaatiossa *R*. Keskimääräinen arvo monikoille, jotka täyttävät yhtäsuuruusvertailun attribuutille *A*. Jos oletamme, että attribuutin *A* arvot jakautuvat tasan relaatiossa *R*, ja että on vähintään yksi arvo, joka täyttää ehdon, niin:

$$SC_A(R) = \begin{cases} 1 & \text{jos } A \text{ on avainattribuutti relaatiossa } R \\ [nTuples(R)/nDistinct_A(R)] & \text{muissa tapauksissa} \end{cases}$$

Voimme arvioida valinnan kardinaliteettia myös muille ehdoille:

$$SC_A(R) = \begin{cases} [nTuples(R)*((max_A(R) - c)/(max_A(R) - min_A(R))] & \text{erisuuruus } (A > c) \\ [nTuples(R)*((c - max_A(R))/(max_A(R) - min_A(R))] & \text{erisuuruus } (A < c) \\ [(nTuples(R)/nDistinct_A(R))*n] & A \text{ joukossa } \{c_1, c_2, \ldots, c_N\} \\ SC_A(R)*SC_B(R) & \text{jos } A \wedge B \\ SC_A(R) + SC_B(R) - SC_A(R)*SC_B(R) & \text{jos } A \vee B \end{cases}$$

Jokaiselle monitasoiselle indeksille *I* attribuuttijoukossa *A*

- $nLevels_A(I)$ — Tasojen määrä *I*:lle
- $nLfBlocks_A(I)$ — Lehtien määrä *I*:lle

Tilastotietojen ylläpito on yleensä ongelmallista, koska jokaisen lisäyksen, poiston tai päivityksen jälkeen ei kannata päivittää tilastotietoja; tämä johtaisi huonoon tehokkuuteen. Tilastotiedot kannattaakin ajaa tämän vuoksi esimerkiksi öisin, viikonloppuisin tai silloin, kun tietokannan kuormitus on vähäinen.

3.5.2 Valintaoperaatio

Valintaoperaatio relaatioalgebrassa toimii relaatiolle *R* siten, että se kertoo ja määrittelee relaation *S,* jossa on vain ne relaation *R* monikot, jotka täyttävät määritellyn predikaatin.

Tarkasteltavat päästrategiat ovat:

- Lineaarihaku (järjestämätön tiedosto, ei indeksiä)
- Binäärihaku (järjestetty tiedosto, ei indeksiä)
- Hajautusavaimen yhtäsuuruus
- Pääavaimen yhtäsuuruusehto

- Pääavaimen erisuuruusehto
- Klusteroidun indeksin yhtäsuuruusehto
- Klusteroimattoman indeksin yhtäsuuruusehto
- Sekundäärisen B+ -puu indeksin ehdon erisuuruus

Lineaarihaulla voi olla tarpeellista käydä läpi jokainen tietue jokaisesta lohkosta, jotta tiedettäisiin täyttääkö se predikaatin. Lineaarihakua kutsutaan myös kokotaulun läpikäynniksi (*full table scan*). Esimerkiksi yhtäsuuruusehto avainattribuutille olettaen, että monikot ovat jakautuneet tasaisesti tiedostoon, silloin on keskimäärin tarpeen käydä puolet läpi lohkoista, jotta löydämme oikean monikon ja tällöin keskimääräinen kustannus on:

[nBlocks(R)/2]

Kaikilla muilla ehdoilla koko tiedosto täytyy käydä läpi, jolloin kustannus on:

nBlocks(R)

Binäärihaussa, jos predikaatti on muotoa (A = x) ja tiedosto on järjestetty relaation *R* avainattribuutilla *A*, arvioitu kustannus haulle on:

[$\log_2$(nBlocks(R))]

Hajautusavaimen A yhtäsuuruusehdolla haettaessa laskemme hajautusalgoritmilla osoitteen monikolle. Jos ylivuotoja (overflow) ei tule, kustannus on 1, mutta jos ylivuoto tulee, ylimääräinen haku voi olla tarpeen, riippuen ylivuotojen määrästä ja niiden käsittelystä.

Jos predikaatti tarkastelee pääavaimen (A = x) yhtäsuuruutta, niin silloin voimme käyttää pääavainindeksiä etsiessämme ehdon täyttävää monikkoa. Tällöin täytyy lukea yksi lohko enemmän kuin mikä on indeksihakujen määrä, saman verran kuin indeksissä on tasoja. Tällöin arvioitu kustannus on:

$nLevels_A(1) + 1$

Jos predikaatti tarkastelee pääavaimen (A < x, A <= x, A > x, A >= x) erisuuruutta, niin silloin voimme käyttää pääavainindeksiä etsiessämme monikkoa, joka täyttää yhtäsuuruusehdon $A = x$. Koska indeksi on lajiteltu, niin tämän jälkeen haemme vain kaikki monikot, jotka löytyvät ennen ja jälkeen tämän monikon. Olettaen että arvot jakautuvat tasaisesti, voimme olettaa, että puolet monikoista täyttää erisuuruusehdon ja silloin arvioitu kustannus on:

$nLevels_A(1) + [nBlocks(R)/2]$

Jos predikaatti tarkastelee yhtäsuuruutta attribuutille *A*, joka ei ole pääavain, mutta jolle on klusteroitu toissijainen indeksi, voimme käyttää indeksiä hakemaan monikot. Tällöin arvioitu kustannus on:

$nLevelsA(I) + [SC_A(R)/bFactor(R)]$

Toinen termi on arvio lohkojen määrälle, joka tarvitaan tallentamaan yhtäsuuruusehdon täyttävät monikot, joiden lukumäärä on ilmaistu lausekkeella $SC_A(R)$.

Jos predikaatti tarkastelee yhtäsuuruutta attribuutille *A*, joka ei ole pääavain ja jolle on klusteroimaton toissijainen indeksi, voimme käyttää indeksiä hakemaan monikot, mutta täytyy olettaa, että monikot ovat eri lohkoissa. Tällöin arvioitu kustannus on:

$nLevelsA(I) + [SC_A(R)]$

Jos predikaatti tarkastelee erisuuruusehtoa attribuutille *A* (A < x, A <= x, A > x, A >= x), jolle on sekundäärinen B+ -puu-indeksi, voimme käydä läpi puuta lehtisolmuista ylöspäin; pienimmästä avaimen arvosta x:ään asti (ehto < tai <=) tai x:stä ylöspäin suurimpaan arvoon asti (ehto > tai >=). Olettaen, että arvot jakautuvat tasaisesti, voimme olettaa, että puolet lehtisolmuista täytyy käydä läpi ja indeksin kautta käydään puolessa monikoista. Tällöin arvioitu kustannus on:

$nLevels_A(I) + [nLfBlocks_A(I)/2 + nTuples(R)/2]$

Seuraavaksi esitetään esimerkki kustannusten estimoinnista valintaoperaatiolle, käyttäen annettuja oletuksia kuvan 2.1 Emp-taululle:

- pääavainattribuutilla *empno* on hajautusindeksi, ilman ylivuotoja
- viiteavainattribuutilla *deptno* on klusteroitu indeksi
- *sal*-attribuutilla on B^+ puu indeksi
- systeemikatalogissa on seuraavat tilastotiedot Emp-taululle
  - $nTuples(Emp) = 3000$
  - $bFactor(Emp) = 30 \Rightarrow nBlocks(Emp) = 100$
  - $nDistinct_{empno}(Emp) = 500 \Rightarrow SC_{empno}(Emp) = 6$
  - $nDistinct_{job}(Emp) = 10 \Rightarrow SC_{job}(Emp) = 300$
  - $nDistinct_{sal}(Emp) = 500 \Rightarrow SC_{sal}(Emp) = 6$
  - $min_{sal}(Emp) = 10000$
  - $max_{sal}(Emp) = 50000$
  - $nLevels_{empno}(I) = 2$
  - $nLevels_{sal}(I) = 2$
  - $nLfBlocks_{sal}(I) = 50$

Arvioitu kustannus lineaarihaulle avainattribuutilla *empno* on 50 lohkoa ja kustannus lineaarihaulle muille kuin avainattribuuteille on 100 lohkoa. Tarkastellaan seuraavia ehtoja:

1. $\sigma_{empno=100}(Emp)$
2. $\sigma_{job='IT_PROG'}(Emp)$
3. $\sigma_{deptno=80}(Emp)$
4. $\sigma_{sal>20000}(Emp)$
5. $\sigma_{job='IT_PROG' \wedge deptno=80}(Emp)$

1: Tämä ehto sisältää ainoastaan yhtäsuuruusehdon pääavaimelle. Attribuutti empno on hajautettu, joten käytetään strategiaa *hajautusavaimen yhtäsuuruus*, joka on määritelty aikaisemmin tässä luvussa ja sen perusteella saamme kustannukseksi yhden lohkon. Arvioitu kardinaliteetti tulosjoukolle on $SC_{empno}(Emp) = 1$.

2: Predikaatin attribuutti ei ole avain eikä indeksoitu, joten lineaarihakua ei voida tehostaa, joten arvioitu kustannus on 100 lohkoa. Arvioitu tulosjoukon kardinaliteetti on $SC_{job}(Emp) = 300$.

3: Predikaatin attribuutti on viiteavain klusteroidulla indeksillä, joten voimme käyttää strategiana *klusteroidun indeksin yhtäsuuruusehtoa* kustannusten arvioinnissa, joka on 2 + [6/30] = 3 lohkoa. Tulosjoukon arvioitu kardinaliteetti on $SC_{deptno}(Emp) = 6$.

4: Predikaatti hakee arvoaluetta sal-attribuutille joka on B^+-puu indeksoitu, joten tälle voidaan käyttää strategiaa *klusteroimattoman indeksin yhtäsuuruusehto,* jolloin arvioitu kustannus on 2 + [50/2] + [3000/2] = 1572 lohkoa. Arvioitu kardinaliteetti tulosjoukolle on $SC_{sal}(Emp) = [3000*(50000-20000)/(50000-10000)] = 2250$.

5: Predikaatilla on kaksi ehtoa, joista jälkimmäisellä on klusteroitu indeksi (sama kuin kohta 3.), jolle tiedämme jo kustannukseksi 3 lohkoa. Kun haemme jokaista monikkoa käyttäen klusteroitua indeksiä, voidaan samalla tarkastaa täyttääkö se myös ensimmäisen ehdon (job='IT_PROG'). Jälkimmäisen ehdon kardinaliteetiksi tiedämme jo kuusi, jos käytämme välirelaatiota T, silloin voimme arvioida erilaisten arvojen määrän job-attribuutille relaatiossa *T*, $nDistinct_{job}(T)$, joka on [(6 + 10)/3] = 6. Kun yhdistämme tähän vielä toisen ehdon, niin silloin saamme arvioiduksi kardinaliteetiksi $SC_{job}(T) = 6/6 = 1$, mikä on siinä tapauksessa oikein, että jokaisessa toimipaikassa on yksi johtaja.

3.5.3 Liitosoperaatio

Liitosoperaatio on eniten aikaa vievä operaatio, kun se on osana karteesista tuloa. Tästä syystä onkin tärkeää, että suoritamme liitosoperaation mahdollisimman tehokkaasti. Theta-liitos operaatio määrittelee predikaatin, joka sisältää monikot, jotka täyttävät määritellyn predikaatin *F* karteesisen tulon kahdesta relaatiosta *R* ja *S*. Predikaatti *F* on muotoa $R.a\ \theta\ S.b$, missä θ voi olla yksi loogisista vertailuoperaattoreista. Mikäli predikaatti sisältää ainoastaan yhtäsuuruusehtoja, niin silloin tämä liitos on yhtäsuuruusliitos (equijoin). Jos liitos taas

tarkastelee kaikkia yhteisiä attribuutteja *R*:lle ja *S*:lle, niin silloin liitosta kutsutaan luonnolliseksi liitokseksi (natural join).

Tarkasteltavat päästrategiat ovat:

- jaksottaiset sisäkkäiset silmukat
- indeksiliitos
- lomitusliitos
- hajautusliitos.

Kardinaliteetti karteesiselle tulolle *R* ja *S*, $R \times S$ on:

$$nTuples(R) * nTuples(S)$$

Kardinaliteetin arvioiminen liitoksille, jotka ovat riippuvaisia arvojen jakaumasta, on jo paljon vaikeampaa. Huonoimmassa tapauksessa tiedämme ainoastaan, että liitoksen kardinaliteetti ei voi olla suurempi kuin karteesisen tulon kardinaliteetti:

$$nTuples(T) \leq nTuples(R) * nTuples(S)$$

Jotkut järjestelmät käyttävät tätä ylärajana, mutta yleensä arvio on liian pessimistinen. Jos oletamme, että arvot jakautuvat tasaisesti molemmissa relaatioissa, voimme parantaa arviota yhtäsuuruusliitokselle predikaatilla $R.A = S.B$:

Jos *A* on avainattribuutti *R*:lle, niin silloin *S*:n monikko voi johtaa vain yhteen monikkoon *R*:ssä. Tässä tapauksessa yhtäsuuruusliitoksen kardinaliteetti ei voi olla suurempi kuin *S*:n kardinaliteetti:

$$nTuples(T) \leq nTuples(S)$$

Jos *B* on *S*:n avain, on voimassa:

$$nTuples(T) \leq nTuples(R)$$

Jos kumpikaan *A* ja *B* eivät ole avaimia, silloin voimme arvioida liitokselle kardinaliteetiksi:

$$nTuples(T) = SC_A(R)*nTuples(S) \text{ tai}$$
$$nTuples(T) = SC_A(S)*nTuples(R)$$

Ylemmässä arviossa käytämme hyväksemme sitä tosiasiaa, että jokaiselle monikolle *s* relaatiossa *S* voimme olettaa keskiarvoksi $SC_A(R)$ monikkoa annetulla arvolla attribuutille *A* ja kertomalla tämä *S*:n monikkojen määrällä, saamme edellä mainitun arvion.

Yksinkertaisimmassa liitosalgoritmissa sisäkkäinen silmukka liittää kaksi relaatiota monikko kerrallaan. Ulompi silmukka käy jokaisen *R*:n monikon läpi ja sisempi silmukka käy jokaisen *S*:n monikon läpi. Voimme parantaa tätä algoritmia lisäämällä kaksi ylimääräistä silmukkaa, jotka prosessoivat lohkoja, jolloin luvut ja läpikäynnit tapahtuvat lohko kerrallaan (nopeampaa kuin yksittäiset luvut). Kunkin *R*:n lohkon lukemisen jälkeen täytyy jokaiselle *R*:n lohkolle lukea jokainen *S*:n lohko ja käydä vastinmonikot läpi, mistä saamme kustannusarvioksi:

$$nBlocks(R) + (nBlocks(S) * nBlocks(R))$$

Tällä arviolla toinen termi on pysyvä, mutta ensimmäinen termi voi olla hyvin riippuvainen relaatiosta, joka on valittu ulommassa silmukassa. Toisin sanoen meidän pitäisi valita sellainen relaatio, joka tarvitsee pienimmän määrän lohkoja ulommassa silmukassa. Toinen parannus tähän strategiaan on lukea mahdollisimman monta lohkoa kerrallaan pienimmästä relaatiosta *R* tietokantapuskuriin, jonne jätetään yksi lohko sisemmälle relaatiolle ja toinen tulosrelaatiolle. Jos puskuri voi tallentaa nBuffer lohkoa, niin siinä tapauksessa voimme lukea nBuffer - 2 lohkoa relaatiosta *R* yhdellä kertaa ja yhden lohkon relaatiosta *S*. Kuitenkin *R*:n lohkoissa täytyy käydä nBlocks(R) kertaa, mutta *S*:n lukukerrat ovat arviolta:

$$[nBlocks(S)*(nBlocks(R)/(nBuffer-2))]$$

Tällöin keskimääräiseksi kustannukseksi tulee:

$$nBlocks(R) + [nBlocks(S)* (nBlocks(R)/(nBuffer-2))]$$

Jos voimme lukea kerralla kaikki *R*:n lohkot puskuriin, niin silloin kustannus on:

$$nBlocks(R) + nBlocks(S)$$

Jos liitosattribuutilla sisemmässä relaatiossa on indeksi *I* (tai hajautusavain), niin silloin voimme vaihtaa tehottoman tiedostoläpikäynnin indeksihakuun käyttäen indeksiliitosta. Jokaiselle *R*:n monikolle, käytämme tällöin indeksiä hakemaan vastaavat *S*:n monikot. Tämä on erittäin tehokas algoritmi liitokselle, koska se estää karteesisen tulon *R*:lle ja *S*:lle. Jos liitosattribuutti *A* on pääavain relaatiossa *S*, silloin arvioitu kustannus on:

$$nBlocks(R) + nTuples(R) * (nLevels_A(I) + 1)$$

Käytettäessä klusteroitua indeksiä *I* attribuutille *A* saadaan arvioiduksi kustannukseksi:

$$nBlocks(R) + nTuples(R) * (nLevelsA(I) + [SCA(R) / bFactor(R)])$$

Lomitusliitos on yhtäsuuruusliitokselle tehokkain algoritmi, kun molemmat relaatiot on lajiteltu liitosattribuuttien mukaan. Tässä tapauksessa etsimme sopivia monikkoja *R*:lle ja *S*:lle lomittamalla kaksi relaatiota. Mikäli relaatiot eivät ole lajiteltuja, niille täytyy suorittaa esiprosessointi, joka lajittelee ne. Kun relaatiot ovat järjestyksessä, saman liitosattribuutin omaavat monikot ovat peräkkäisessä järjestyksessä. Jos oletamme, että liitos on monen suhde moneen liitos, niin silloin useampi *R*:n ja *S*:n monikko voi saada saman liitosarvon ja jos oletamme, että jokainen samat liitosarvot omaava joukko voidaan säilyttää tietokantapuskurissa samaan aikaan, niin tällöin jokainen lohko jokaisesta relaatiosta tarvitsee lukea vain kerran. Tällöin arvioitu kustannus on:

$$nBlocks(R) + nBlocks(S)$$

Jos relaatio, esimerkiksi *R*, pitää lajitella ensin, niin silloin lisätään lajittelun kustannus, joka on esimerkiksi lomituslajittelua (merge sort) käytettäessä on arviolta:

$$nBlocks(R)*[\log_2(nBlocks(R))]$$

Hajautusliitosta voidaan käyttää luonnolliselle- ja yhtäsuuruusliitokselle. Hajautusliitos-algoritmia voidaan käyttää laskemaan liitokset relaatioiden *R* ja *S* liitosattribuuttijoukolle *A*. Algoritmi toimii siten, että se osittaa relaatiot *R* ja *S* käyttäen hajautusfunktiota. Jokaisella *R*:n ja *S*:n ekvivalentilla osalla pitäisi olla sama arvo liitosattribuuteilla, joita se voi sisältää useampia. Esimerkiksi jos relaatio *R* on ositettu $R_1, R_2, \ldots, R_M$ ja relaatio *S* on ositettu $S_1, S_2, \ldots, S_M$ käyttäen hajautusfunktiota *h*(), ja vastaavasti jos *B* ja *C* ovat *R*:n ja *S*:n attribuutteja ja $h(R.B) \neq h(S.C)$, silloin $R.B \neq S.C$. Toisaalta yhtäsuuruus $h(R.B) = h(S.C)$ ei välttämättä tarkoita, että $R.B = S.C$, koska eri arvot voivat saada saman hajautusarvon. Seuraavana vaiheena on lukea jokainen *R*:n osio vuorollaan ja jokaiselle niiden monikoille liittää vastaavan *S*:n osion monikot. Jos toisessa vaiheessa käytetään sisäkkäisiä silmukoita, niin silloin pienempää osiota käytetään ulommassa silmukassa. Kun jokainen $R_{i:n}$ osio on luettu muistiin ja jokainen S_i:n vastaava osio on luettu ja jokaista monikkoa on kokeiltu R_i:lle, ja tämän lopputuloksena löydetään vastaavuudet. Tehokkuuden parantamiseksi on yleistä, että muistiin tehdään hajautustaulu jokaiselle osiolle R_i käyttäen toista hajautusfunktiota. Hajautusliitoksen arvioitu kustannus on:

$$3(nBlocks(R) + nBlocks(S))$$

Tähän päästään siten, että luetaan *R* ja *S* osiointia varten, kirjoitetaan osiot levylle ja sen jälkeen luetaan *R*:n ja *S*:n osiot vastaavuuksien löytämiseksi. Tämä arvio ei kuitenkaan ota huomioon tilanteita, joissa osioinnissa tulee ylivuotoja ja se olettaa, että hajautusindeksi voidaan säilyttää muistissa. Mikäli tämä ei kuitenkaan pidä paikkaansa, niin silloin osiointia ei voida tehdä kerralla ja joudutaan käyttämään rekursiivista osiointialgoritmia. Tällöin arvioitu kustannus on:

$$2(nBlocks(R) + nBlocks(S))*[\log_{nBuffer-1}(nBlocks(S)) - 1] + nBlocks(R) + nBlocks(S)$$

3.5.4 Projektio-operaatio

Projektio on myös yksioperandinen operaatio, joka määrittelee sen että relaatio *S* sisältää vertikaalisen osajoukon relaatiosta *R* poimien arvot valituille attribuuteille ja poistaen duplikaatit. Projektion toteutuksessa tarvitaan kaksi vaihetta: ensiksi poistetaan attribuutit, joita ei tarvita ja sen jälkeen poistetaan duplikaattimonikot, jotka edellinen vaihe aiheutti. Vaiheista jälkimmäinen on ongelmallisempi, koska sitä ei tarvita jos attribuutit sisältävät relaation avaimen. Duplikaattien eliminoinnissa käytetään kahta lähestymistapaa: hajautusta ja lajittelua.

Silloin kun projektio sisältää avainattribuutin, ei duplikaattien poistoa tarvita. Tällöin arvioitu kardinaliteetti on:

$$nTuples(S) = nTuples(R)$$

Jos taas projektio sisältää yhden attribuutin, joka ei ole avain, niin silloin arvioitu kardinaliteetti on:

$$nTuples(S) = SC_A(R)$$

Muissa tapauksissa oletamme, että projektiot tuottaa karteesisen tulon attribuuteilleen, jolloin arvioitu kardinaliteetti on:

$$nTuples(S) \leq \min(nTuples(R), \Pi_{i}^{m}{}_{=1}(nDistinct_A(R)) ,$$

missä *m* on attribuuttien määrä.

Duplikaattien poistossa lajittelun avulla käytetään lajitteluavaimena kaikkia jäljellä olevia attribuutteja. Lajittelun jälkeen duplikaatit on helppo poistaa, koska ne ovat peräkkäin. Poistaaksemme relaatiosta attribuutit, joita ei tarvita, täytyy kaikki *R*:n monikot lukea kerran ja kopioida tarvitut attribuutit väliaikaiseen relaatioon, tämän kustannus on nBlocks(R). Lajittelun arvioitu kustannus on:

$$nBlocks(R)*[\log_2(nBlocks(R))]$$

ja yhdistetty arvioitu kustannus on:

$$nBlocks(R) + nBlocks(R)*[\log_2(nBlocks(R))]$$

Duplikaatit voidaan poistaa myös hajautuksen avulla. Hajautus on käyttökelpoinen, kun tietokantapuskurissa on paljon *R*:n lohkoja. Hajautuksessa on kaksi vaihetta: ositus ja duplikaattien poisto. Ositusvaiheessa varataan yksi tietokantapuskurin lohko *R* relaation lukemista varten ja (nBuffer-1) lohkoa lopputulosta varten. Jokaiselta *R*:n monikolta poistetaan tarpeettomat attribuutit. Sitten käytetään hajautusfunktiota *h* jäljelle jääneille attribuuteille, minkä jälkeen kirjataan monikko hajautusarvolle. Hajautusfunktio *h* tulisi valita siten, että monikot ovat tasaisesti jakautuneet (nBuffer-1) osaan. Kaksi monikkoa, joilla on eri hajautusarvo, eivät voi olla duplikaatteja, tämän vuoksi voimme rajoittaa duplikaattien poiston ainoastaan niihin hajautusarvoihin, joissa monikkoja on enemmän kuin yksi. Toisessa vaiheessa luetaan jokainen (nBuffer-1) osio kerrallaan, kutsutaan jokaiselle monikolle toista hajautusfunktiota ja lisätään saatu hajautusarvo muistissa olevaan hajautus tauluun. Jos monikko saa saman hajautusarvon kuin joku toinen, silloin tarkistetaan ovatko ne duplikaatteja. Jos ne ovat duplikaatteja, niin silloin uudempi jätetään pois. Kun osiot on käyty läpi, niin kirjoitetaan hajautustaulussa olevat monikot tulosjoukkoon. Jos tarvittavien lohkojen määrä tarvittavalle väliaikaiselle taululle, joka saadaan projektiosta *R* ennen duplikaattien poistoa, on *nb*, niin silloin arvioitu kustannus on:

$$nBlocks(R) + nb$$

Kustannuksessa ei ole mukana tulosjoukon kirjoitusta levylle ja se olettaa ettei hajautuksessa tule ylivuotoja.

3.5.5 Joukko-operaatiot

Binääriset joukko-operaatiot yhdiste, leikkaus ja erotus voidaan suorittaa ainoastaan sellaisille joukoille, jotka molemmat sisältävät saman määrän ja samanmuotoisia attribuutteja [EN00]. Nämä operaatiot voidaan toteuttaa siten, että ensimmäiseksi molemmat relaatiot lajitellaan saman attribuutin mukaan ja tämän jälkeen käydään läpi lajitellut relaatiot kertaalleen, jotta saadaan haluttu tulos. Yhdisteen tapauksessa liitetään tulosjoukkoon molempien relaatioiden monikot, poistaen tarvittaessa duplikaatit. Leikkauksen tapauksessa tulosjoukkoon laitetaan ainoastaan ne monikot, jotka löytyvät molemmista relaatioista. Erotuksen tapauksessa tulosjoukkoon laitetaan ne *R*:n monikot, joita ei löydy relaatiosta *S*. Jokaiselle näistä operaatioista voimme tehdä algoritmin, joka pohjautuu lomitusliitokseen. Kaikille operaatioille arvioitu kustannus on:

$$\mathrm{nBlocks(R)} + \mathrm{nBlocks(S)} + \mathrm{nBlocks(R)}*[\log_2(\mathrm{nBlocks(R)})] + \mathrm{nBlocks(S)}*[\log_2(\mathrm{nBlocks(S)})]$$

Operaatioiden toteuttamiseen voidaan käyttää myös hajautusta, jolloin yksi taulu ositetaan ja muita käytetään apuna löytämään oikea osio. Esimerkiksi yhdisteellä ensiksi relaatio *R* ositetaan, ja tämän jälkeen relaation *S* monikoille lasketaan samalla algoritmilla hajautusarvo ja lisätään monikot tulosjoukkoon, ei kuitenkaan duplikaatteja. Leikkauksessa ensimmäiseksi ositetaan relaatio *R* hajautustauluun ja sen jälkeen jokaiselle *S*:n monikolle tarkastetaan löytyykö vastaavaa hajautustaulusta, jos löytyi, lisätään se tulosjoukkoon [EN00]. Koska yhdisteestä täytyy poistaa duplikaatit, sen kardinaliteetin arviointi on vaikeaa, mutta voimme kuitenkin antaa korkeimman ja alimman raja-arvon:

$$\max(\mathrm{nTuples(R)}, \mathrm{nTuples(S)}) \leq \mathrm{nTuples(T)} \leq \mathrm{nTuples(R)} + \mathrm{nTuples(S)}$$

Myös erotukselle voidaan antaa korkein ja alin raja-arvo:

$0 \leq nTuples(T) \leq nTuples(R)$

3.6 Muita optimointiperiaatteita

Yksi SQL:n parhaimmista ominaisuuksista on kyky esittää monimutkaisia ehtoja käyttäen sisäkkäisiä kyselyjä, jotka voivat olla korreloituja tai korreloimattomia. Perinteinen suoritustapa tällaisille kyselyille on monikkojen iterointiparadigma, joka on tehoton erityisesti korreloiduilla alikyselyillä. Vuonna 1982 Won Kim kehitti parannuksia tähän evaluointitapaan ja osoitti, että usein on mahdollista muuttaa alikyselyllinen kysely kyselyksi, jossa ei ole alikyselyä. Kim jakoi alikyselyt neljään eri luokkaan: korreloimattomat, koosteiset alikyselyt (type-A); korreloimattomat, ei-koosteiset alikyselyt (type-N); korreloidut, ei-koosteiset alikyselyt (type-J); ja korreloidut koosteiset alikyselyt (type-JA). Tyyppi-A:n kyselyille ei ole tehtävissä mitään. Tyyppi-N:n kyselyt vastaavat niitä käyttäen predikaatteja IN, NOT IN, EXISTS ja NOT EXISTS. Kim huomasi, että kyselyt jotka käyttävät IN predikaattia voidaan muuttaa liitoksiksi. Tyyppi-J:n kyselyjä kohdellaan yleensä tyyppi-N:nä. Tyyppi-JA on näistä kiinnostavin, koska se on yleinen tyyppi SQL-kyselyissä ja lisäksi muut kyselyt voidaan kirjoittaa tyyppi-JA:na. Esimerkiksi kysely, jossa on EXISTS, voidaan uudelleen kirjoittaa seuraavalla tavalla: ehto EXISTS (SELECT attr…) (jossa attr on attribuutti) muutetaan muotoon 0 > (SELECT COUNT(*) (muutos '*' :ksi tarvitaan null-arvojen vuoksi). Tyyppi-JA:n kyselyjen käsittelyssä ensimmäiseksi alikyselyllinen kysely muutetaan koosteiseksi kyselyksi; tulos viedään väliaikaiseen tauluun, mikä liitetään pääkyselyyn. Esimerkiksi kysely:

```
SELECT c_custkey FROM Customer WHERE c_acctbal > 10000 AND
c_acctbal > (SELECT sum(o_totalprice) FROM Order WHERE o_custkey
= c_custkey);
```

suoritetaan:

```
CREATE Temp AS
SELECT     o_custkey
sum(o_totalprice) as sumprice
```

```
FROM       Order
GROUP BY   o_custkey;

SELECT     o_custkey
FROM       Customer, Temp
WHERE      o_custkey =  c_custkey
AND        c_acctbal > 10000
AND        c_acctbal > sumprice;
```

Ylläoleva tapaus ei kuitenkaan toimi useammasta syystä: asiakkaat, joilla ei ole tilauksia puuttuvat uudelleen kirjoitetusta kyselystä, mutta alkuperäisessä ne tulevat mukaan. Toinen ongelma on, että käsittely on virheellinen, kun korrelaatio käyttää muuta kuin yhtäsuuruusehtoa. Nämä ongelmat voidaan kuitenkin ratkaista kerralla käyttämällä seuraavanlaista strategiaa: ensiksi relaatiot liitetään ulkoliitoksella ja sitten suoritetaan koostaminen. Esimerkiksi:

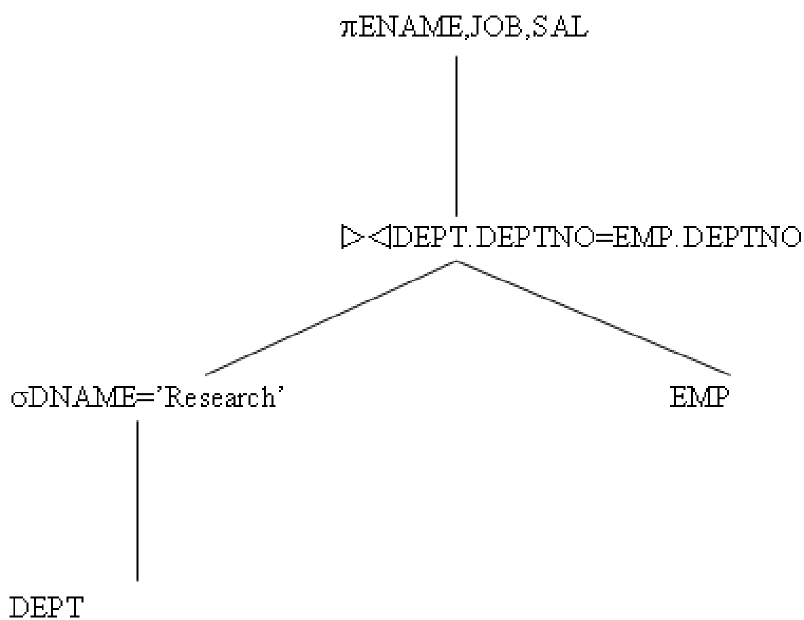
```
CREATE Temp(ckey, sumprice) AS
(SELECT    c_custkey
 sum(o_totalprice) as sumprice
FROM       Customer Left Outer Join Order on o_custkey =
c_custkey
GROUP BY   c_custkey;

SELECT     c_custkey
FROM       Customer, Temp
WHERE      c_custkey = ckey
AND        c_acctbal > 10000
AND        c_acctbal > sumprice;
```

Tässä tavassakin on kuitenkin kaksi epäkohtaa liittyen tehokkuuteen. Ensimmäiseksi olemme kiinnostuneet vain tietyistä asiakkaista (luottoraja (c_acctbal) -ehto) ja ulkoliitos tuo kaikki asiakkaat [RDF06].

3.7 Suoritussuunnitelman generointi

Kyselypuuna esitetyn relaatioalgebran lausekkeen suoritussuunnitelma (ks. kuva 3.1) sisältää tiedot saantimenetelmistä (access methods) kyselyn relaatioille sekä algoritmit, joilla relaatioiden operaatiot suoritetaan. Ennen kuin suoritussuunnitelma voidaan generoida, optimoijan täytyy valita optimaalisin strategia haun toteuttamiseksi [EN00]. Otetaan esimerkiksi kyselypuu (kuva 3.3):



Kuva 3.3. Esimerkki kyselypuusta

Kun puusta generoidaan suoritussuunnitelma, optimoija saattaisi valita SELECT-operaatiolle indeksihaun (jos sellainen on olemassa), kokotaulun läpikäynnin Emp-taululle, liitokselle sisäkkäiset silmukat ja liitosoperaation tulokselle projektio-operaation. Toisaalta kyselyn suoritus voi määritellä materialisoidun tai putkitetun evaluoinnin. Materialisoidussa evaluoinnissa operaation tulos tallennetaan väliaikaiseen relaatioon. Esimerkiksi liitos-operaatio voidaan suorittaa ja sen tulos tallettaa väliaikaiseen relaatioon, jonka jälkeen se luetaan toisen algoritmin syötteeksi joka suorittaa projektio-operaation ja tämän tuloksena voi syntyä kyselyn tulostaulu. Putkitetussa evaluoinnissa ei taas muodostu välitaulua, vaan välitulos ohjataan suoraan seuraavalle operaatiolle. Esimerkiksi Dept-taulun monikot, jotka on

saatu SELECT-operaatiolla viedään puskuriin, tämän jälkeen liitosoperaatio hakee monikot puskurista ja sen tulos putkitetaan projektio-operaatiolle. Putkituksella saadaan kustannus-säästöä, koska välitulosta ei tarvitse kirjoittaa ja lukea levyltä [EN00].

4 ORACLEN SUORITTAMA KYSELYJEN OPTIMOINTI

Tässä luvussa perehdytään siihen kuinka Oraclen tietokannassa voidaan vaikuttaa kyselyjen tehokkuuteen, optimoida niitä/vaikuttaa optimoijaan ja mitä apuvälineitä näihin toimintoihin on. Yleensä kyselyn suoritustehokkuutta voidaan parantaa helposti muuttamalla kyselyn rakennetta tehokkaammaksi, lisäämällä/käyttämällä indeksejä, virkistämällä tilastotiedot, huolehtimalla resurssien riittävyydestä ja analysoimalla odotustiloja. Tämä luku käsittelee Oraclen kyselyjen optimointia pääosin lähteen [Ala05] mukaan.

4.1 Kustannusperustainen optimointi

Kustannusperustainen kyselynoptimointi perustuu optimoijan arvioon kyselysuunnitelman operaattoreiden tuottamien rivien määristä (ts. kardinaliteeteista). Kustannusperustaisuus sisältää useita hyvin tunnettuja potentiaalisia virheiden lähteitä. Tietohakemiston tilastovirheiden lisäksi kustannusperustainen kyselyoptimointi voi tehdä virheellisiä oletuksia sarakkeen arvojen tasaisesta jakaumasta, sarakkeiden keskinäisestä riippumattomuudesta sekä liitossarakkeiden keskinäisistä suhteista [IC91].

Kustannusperustaisessa optimoinnissa Oracle ottaa huomioon I/O-kustannuksen ja CPU-kustannuksen. Optimoija evaluoi eri kustannukset vertaamalla I/O-operaatioiden kokonais-aikaa sekä tarvittua CPU-aikaa, joka tarvitaan kyselyn suorittamiseksi. Optimoija ottaa I/O:n ja CPU:n kokonaisajan, sekä arvioi ja konvertoi sen kokonaissuoritusajaksi. Tämän jälkeen se vertailee eri suorituspolkujen kustannuksia ja valitsee niistä kustannuksiltaan pienimmän.

Laskeakseen kustannukset tarkasti optimoijalla täytyy olla ajantasaiset tilastotiedot, jotka sisältävät I/O-etsintäajan, I/O-siirtoajan ja CPU-nopeuden. Nämä kertovat optimoijalle, kuinka nopea systeemin I/O on, sekä CPU:n tehokkuuden. Kustannusperusteinen optimointi antaa Oraclessa lähes aina paremman tuloksen kuin sääntöperustainen optimointi. Versiosta 10g lähtien tietokanta on voinut itse huolehtia tilastotietojen ajantasaisuudesta, aiemmissa versioissa tilastotiedot piti virkistää erikseen.

Kustannusperusteisella optimoijalla on kolme eri toimintatapaa, jotka sille voidaan asettaa: ALL_ROWS, FIRST_ROWS_n ja FIRST_ROWS. ALL_ROWS on oletus optimointitapa, joka ohjaa optimoijan käyttämään kustannusperusteista tapaa olipa kyselyn tauluille tilastotietoja tai ei, päämääränä maksimoida tuotos. FIRS_ROWS_n käyttää kustannusperusteista optimointia riippumatta tilastotietojen saatavuudesta, päämääränä vastausaika ensimmäisille n riveille, missä n voi olla 10, 100 tai 1000. FIRST_ROWS käyttää myös kustannusperusteista optimointia ja tiettyä heuristiikkaa (rules of thumb – 'peukalosääntöjä'), riippumatta siitä onko tilastotietoja saatavilla tai ei. Päämääränä on saada ensimmäiset *n* riviä mahdollisimman nopeasti. Tila voidaan asettaa tietokantainstanssille, sessiolle tai lauseelle. Session optimointitilaa voidaan muuttaa lauseella:

```
ALTER SESSION SET optimizer_goal = first_rows_10;
```

Lause asetta optimointitilaksi FIRST_ROWS_10.

Optimoija suorittaa useita vaikeita vaiheita löytääkseen optimaalisen suoritussuunnitelman kyselylle. Alkuperäinen SQL-kysely muuttaa yleensä muotoaan ja optimoija evaluoi eri suorituspolut. Jos liitokset ovat tarpeen, niin silloin optimoija evaluoi kaikki mahdolliset liitostavat ja -järjestykset. Optimoija evaluoi kaikki mahdollisuudet ja palauttaa suoritussuunnitelman, jota se pitää kokonaiskustannuksiltaan pienimpänä, joka sisältää I/O- ja CPU-kustannukset.

Oracle ei yleensä koskaan suorita kyselyä sellaisena kuin sen on käyttäjä kirjoittanut. Optimoija tarkastaa, jos erilainen SQL-muoto tuottaa saman lopputuloksen tehokkaammin, niin silloin optimoija muuttaa SQL:n ennen sen suorittamista. Hyvä esimerkki muunnoksesta on lause, jossa on OR-ehto, jolloin optimoija muuttaa ehdon UNION:ksi tai UNION ALL:ksi. Myöskin esimerkiksi, jos kysely sisältää indeksin, optimoija voi muuttaa lauseen sellaiseksi, että se suorittaa kokotaulun läpikäynnin, mikä on joissain tilanteissa paljon tehokkaampi. Joka tapauksessa on aina hyvä muistaa, että kyselyä ei aina suoriteta käyttäjän haluamassa muodossa.

Kustannusperusteinen optimoija on systemaattinen, mutta ei ole kuitenkaan taattua, että optimoija noudattaa samaa suunnitelmaa vastaavissa tapauksissa, eikä optimoija muutenkaan ole täydellinen. Seuraavia asioita tulisi erityisesti huomioida:

- Optimoijaa ei ole muutettu Oraclen versioiden välillä. Suoritussuunnitelmat kuitenkin saattavat muuttua version mukana.
- Ohjelmoija voi tietää enemmän kuin optimoija valitessaan parasta suorituspolkua. Ohjelmoija tietää käyttäjän tarpeet, joista optimoija on taas täysin tietämätön ja tämä voi johtaa tilanteeseen, että optimoija optimoi tuotosta, vaikka käyttäjä haluaisi tuloksen nopeasti näytölle. Käyttämällä vihjettä FIRST_ROWS_n, voidaan tämä haittapuoli voittaa.
- Optimointi on täysin riippuvainen oikeista tilastotiedoista. Jos tilastotiedot puuttuvat tai ne ovat vanhoja, niin silloin optimoija tekee vääriä päätöksiä.

4.2 Tehokkaan SQL:n kirjoittaminen

Yksi tietokantavastaavan tärkeimmistä tehtävistä on auttaa kirjoittamaan tehokasta SQL:ää sovelluksiin. Tehokas koodi tarkoittaa hyvää suorituskykyä ja helppoa tapaa vähentää kyselyjen I/O:ta pienentämällä rivien määrää, jotka optimoija joutuu läpikäymään. Optimoijan tehtävänä on löytää optimaalisin suoritussuunnitelma kyselylle, joka tarkoittaa sitä, että optimoija ei uudelleen kirjoita tehotonta kyselyä, vaan ainoastaan tuottaa suoritussuunnitelman kyselylle. Vaikka kysely olisi kirjoitettu tehokkaaksi, optimoija ei aina tuota parasta suoritussuunnitelmaa. Kirjoittajalla on aina parempi tuntemus ohjelmasta ja datasta kuin optimoijalla. Tästä syystä voidaankin käyttää apuna vihjeitä pakottamaan optimoija käyttämään sitä tietoa.

4.2.1 Tehokas where

Valitsevuus (selectivity) -kriteeri WHERE-lauseessa voi vähentää dramaattisesti datan määrää, joka Oraclen täytyy käsitellä suorittaessaan kyselyä. On olemassa muutamia periaatteita,

joiden avulla voidaan varmistua siitä, että SQL-kyselyjen rakenne ei ole luonnostaan tehoton. Liitostavat voivat olla hyviä, mutta katsottaessa joitakin näistä periaatteista ne tuomitsevat sen tehokkuuden näkökulmasta. Huolellinen WHERE-ehtojen määrittely voi saada aikaan huomattavan merkityksen, kun optimoija valitsee olemassa olevan indeksin. Valitsevuuden periaate – kyselyn palauttamien rivien määrä prosentteina taulun kokonaisrivien määrästä – on siinä ideana. Pienempi prosentti tarkoittaa korkeaa valitsevuutta ja päinvastoin. Koska valitsevampi ehto tarkoittaa vähemmän I/O:ta, optimoijalla on tapana pitää parempana valita sen tapaisia WHERE-ehtoja toisten sijaan. Esimerkiksi:

```
SELECT * FROM Emp WHERE empno=12 AND job='MANAGER';
```

Esimerkissä on kaksi WHERE-ehtoa, mutta kuten näemme, ensimmäinen ehto empno käyttää vähemmän I/O:ta, koska se on pääavain ja sillä on parempi valitsevuus – ainoastaan yksi rivi palautuu. Optimoija selvittää molempien sarakkeiden valitsevuuden katsomalla sen indeksitilastotiedoista, joka kertoo kuinka monta riviä taulu sisältää molemmille sarakkeiden arvoille. Jos kummallakaan sarakkeella ei ole indeksiä, niin silloin Oracle käyttää kokotaulun läpikäyntiä (full table scan) saadakseen vastauksen kyselylle. Jos taas molemmat sarakkeet on indeksoitu, niin silloin käytetään valitsevampaa. Kun optimoija valitsee kokotaulun läpikäynnin, mutta tietokantavastaava on sitä mieltä, että sen tulisi käyttää indeksiä, tulee suorittaa seuraavat askeleet:

- Näkymien käyttö kyselyssä estää joskus indeksien käytön. Tarkasta että suoritussuunnitelma näyttää oikeat indeksit kyselylle.
- Jos luulet, että suuri datan määrä vääristää taulua, käytä silloin histogrammia apuna tarjotaksesi Oraclelle täsmällisen esityksen datan jakaumasta taulussa. Optimoija olettaa, että sarakkeen data jakautuu tasaisesti. Optimoija saattaa luopua indeksin käytöstä, vaikka sarakkeen arvo on valitseva, koska sarake itsessään on luonteeltaan valitsematon. Histogrammi auttaa optimoijaa saamaan oikean kuvan sarakkeen datan jakaumasta.
- Jos Oracle edelleen kieltäytyy käyttämästä indeksiä, voidaan se pakottaa käyttämään sitä vihjeiden avulla.

Käytettäessä WHERE-ehtoa kuten job LIKE '%AN%', optimoija voi päättää olla käyttämättä indeksiä ja tehdä kokotaulun läpikäynnin, koska sen täytyy suorittaa merkkijonon sovitus-operaatio koko sarakkeelle löytääkseen oikeat rivit. Optimoija oikeellisesti selvittää, että sen täytyy mennä eteenpäin ja katsoa vain taulua sen sijaan, että se lukisi sekä indeksiä ja taulun arvot.

Esimerkiksi jos taulussa on 1000 riviä, jotka ovat 200:ssa lohkossa ja sille suoritetaan kokotaulun läpikäynti, olettaen että DB_FILE_MULTIBLOCK_READ_COUNT on kahdeksan, silloin I/O arvoksi saadaan 25 koko taulun lukemiselle. Jos indeksillä on pieni valitsevuus, silloin suurin osa indeksistä täytyy lukea ensin. Jos indeksillä on 40 lehteä ja niistä täytyy lukea 90 prosenttia indeksoidun datan hakuun ensin, niin silloin I/O on jo arvossa 32, sen lisäksi täytyy suorittaa lisää I/O-operaatioita arvojen lukemiseksi taulusta. Kokotaulun läpikäynti käyttää kuitenkin vain 25 I/O:ta, mikä tarkoittaa, että se on tässä tapauksessa tehokkaampi valinta kuin indeksin käyttö. Täytyy muistaa, että pelkkä indeksin olemassaolo ei takaa sitä, että sitä aina käytetään.

Kun WHERE-ehdossa käytetään SQL-funktiota, niin silloin Oraclen optimoija estää sarakkeen indeksin käytön. Pitää aina muistaa varmistaa, että käytetään funktioon perustuvaa indeksiä, jos on pakko käyttää funktiota WHERE-ehdossa.

Suurin osa SQL lauseista sisältää usean taulujen liitoksia. Usein väärä taulujen liitosstrategia tuhoaa koko kyselyn. Seuraavana muutama vihje taulujen liittämiseksi:

- Käytä yhtäsuuruusliitosta, koska se on tehokkaampi kuin muut liitokset.
- Filtteröinti-operaatioiden suoritus aikaisessa vaiheessa vähentää liitettävien rivien määrää myöhäisemmissä vaiheissa. Tarkoituksena on käyttää kyselyn päätaululle mahdollisimman valitsevaa ehtoa, koska se tarkoittaa, että pienempi määrä rivejä siirtyy seuraavaan vaiheeseen.
- Tee liitokset siinä järjestyksessä, että ensimmäinen vaihe palauttaa pienimmän määrän rivejä.

Kun on tarpeellista laskea useita kokonaisuuksia samalle taululle, voidaan useiden SQL-lauseiden kirjoitus estää Case-rakenteen avulla. Erillisille kyselyille Oraclen täytyy lukea koko taulu ja siksi onkin tehokkaampaa käyttää tässä tapauksessa Case-lausetta, joka mahdollistaa usean kokonaisuuden käsittelyn yhdellä taulun lukemisella.

Alikyselyiden suorituskyky on parempi silloin, kun käytetään IN-ehtoa EXISTS-ehdon sijaan. Oracle suosittelee käytettäväksi IN-ehtoa, jos alikyselyllä on valitsevampi WHERE-osa ja jos taas isäkyselyllä on valitsevampi WHERE-osa, niin silloin on parempi käyttää EXISTS-rakennetta.

Aina kun on mahdollista, tulee käyttää WHERE-ehtoa HAVING-ehdon sijaan. WHERE-ehto rajoittaa rivien määrää. HAVING-ehto aiheuttaa sen, että rivejä haetaan paljon enemmän kuin on tarpeen, mikä johtaa turhaan lajitteluun ja summaukseen.

4.2.2 Vihjeiden käyttö

Olettamuksena kustannusperusteisessa optimoinnissa on, että optimoija tietää parhaiten mitä tehdä. Olettamus pitää paikkansa, jos on kysymys optimaalisen suoritussuunnitelman valinnasta tilastotietojen perusteella. Täytyy kuitenkin muistaa, että optimoija perustuu sääntöihin ja ohjelmoijalla on sovellusalueesta ja datasta sellaista tietoa, jota optimoijalla ei koskaan voi olla. Niinpä ohjelmoija voikin antaa kyselyyn vihjeitä, jotka pakottavat optimoijan annetunlaiseen suoritukseen. Esimerkiksi, jos tiedetään, että toinen indeksi on valitsevampi kuin toinen, niin silloin Oracle voidaan pakottaa käyttämään valitsevampaa indeksiä vihjeen avulla. Vihjeet voivat muuttaa liitostapaa, liitosjärjestystä tai hakupolkua. Vihjeillä voidaan myös rinnakkaistaa SQL-operaatioita. Seuraavana esitetään muutama perusvihje, joita kyselylle voidaan antaa:

- ALL_ROWS: Ohjaa Oraclen optimoimaan tuotosta.
- FIRST_ROWS(n): Käskee Oraclea hakemaan ensimmäiset *n* riviä mahdollisimman nopeasti, tavoitteena nopea vasteaika.

- FULL: Ohjaa tekemään taululle koko taulun läpikäynnin, ohittaen indeksit.
- ORDERED: Vihje pakottaa tiettyyn suoritusjärjestykseen kyselyn tauluille.
- INDEX: Käskee käyttämään indeksi hakua, vaikka optimoija olisi tehnyt kokotaulun läpikäynnin.
- INDEX_FFS: Indeksin nopea läpikäynti, vihje pakottaa lukemaan indeksin kokonaan, lukien monta lohkoa kerrallaan.

4.2.3 Parhaan liitostavan ja järjestyksen valinta

Liitostavan valinta perustuu siihen, kuinka monta riviä halutaan kyselyn palauttavan. Optimoija yrittää valita ideaalisen liitostavan, mutta ei kuitenkaan aina tee sitä useastakaan syystä. Tästä syystä tulisikin aina tarkastaa mitä tapaa optimoija ehdottaa, ja vaihtaa sitä tarvittaessa.

Karteesista tuloa pitää välttää. Karteesinen tulo tulee, kun liitosten välillä ei ole ehtoa. Vaikka taulut olisivat pieniä, niin datamäärä kasvaa karteesisessa tulossa nopeasti suureksi ja johtaa huonoon suorituskykyyn. Esimerkiksi, jos kyselyssä on kaksi taulua ja molemmissa 1000 riviä, niin silloin karteesinen tulo palauttaa 1000*1000 = 1000000 riviä.

Jos liitetään pieniä osajoukkoja dataa, niin silloin sisäkkäiset silmukat (nested loop) on ideaalinen tapa. Olettaen, että palautettava joukko on maksimissaan 10000 riviä, niin siinä tapauksessa sisäkkäiset silmukat on todennäköisesti paras liitostapa. Vihje sisäkkäisen silmukan käytölle annetaan seuraavasti:

```
SELECT  /*+ USE_NL (Taulu1 Taulu2) */
```

Jos taas liitos tuottaa suuren määrän rivejä lopputulokseksi tai olennainen osa taulujen datasta liitetään, niin silloin paras liitostapa on todennäköisesti hajautusliitos (hash join). Vihje hajautusliitoksen käyttämiseen annetaan seuraavasti:

```
SELECT /*+ USE_HASH (Taulu1 Taulu2) */
```

Lomitusliitos on ideaalisin, kun liitos tehdään erisuuruusehdolla. Vihje annetaan seuraavalla tavalla:

```
SELECT /*+ USE_MERGE (Taulu1 Taulu2) */
```

Taulujen liitosjärjestys on erittäin tärkeä silloin, kun SQL-lauseessa liitetään yhteen kaksi tai useampia tauluja. Päätaulu on liitoksen ensimmäinen taulu, joka tulee WHERE:n jälkeen. Päätaulun ehtojen pitäisi sisältää sellaisia rajoitteita, että se palauttaa vähän rivejä tai ainakin pienimmän määrän rivejä verrattuna muiden taulujen ehtoihin. Jos olisimme liittämässä kolmea taulua, ensimmäisenä tulisi liittää taulu, jolla on niistä tiukin rajoite.

4.2.4 Indeksointi

Indeksi on struktuuri, jossa on yhden tai useamman taulun sarakkeen arvo ja joka palauttaa arvon kyseisille sarakkeille nopeasti. Indeksin tehokkuus tulee esille siinä, että siitä löytää nopeasti tarpeelliset rivit tarvitsematta käydä läpi kaikkia taulun rivejä. Yleensä indeksit ovat paljon tehokkaampia kuin taulun läpikäynnit, koska ne tarvitsevat paljon vähemmän I/O:ta. EXPLAIN PLAN -komennon avulla voidaan selvittää käyttääkö tietty kysely indeksejä. Jos indeksejä ei käytetä oikein, ne voivat hidastaa kyselyjä ja tietokantaa merkittävästi.

Indeksejä tulisi käyttää ainoastaan silloin, kun kysely hakee pienen osan taulusta. Jos kysely hakee taulun riveistä enemmän kuin 10-15 prosenttia, niin silloin ei kannata indeksoida. Täytyy muistaa, että indeksin käyttö estää kokotaulun läpikäynnin. Joka tapauksessa aina kun halutaan päästä tietylle riville indeksoidussa taulussa, täytyy ensimmäiseksi Oraclen katsoa onko kyselyssä oleville sarakkeille indeksiä. Indeksistä Oracle hakee ROWID:n, joka on looginen osoite sen sijaintiin levyllä. Jos taulun rivit ovat yksilöiviä, niin silloin voidaan käyttää taulun pääavainta. Pääavaimena olevan sarakkeen täytyy olla yksilöivä ja sillä täytyy olla arvo (not null). Pääavaimen lisäksi taululla voi olla useita sekundääriavaimia.

Päämääränä tulisi olla, että indeksejä käytetään mahdollisimman vähän. Jokainen lisäys-, päivitys- ja poisto-operaatio aiheuttavat muutoksen taulun indeksiin ja tämä voi johtaa joissain tapauksissa sovellusten hidastumiseen. Seuraavaksi esitetään muutama ohje, joita seuraamalla voi varmistaa, että indeksit auttavat sovellusta sen sijaan, että vaikeuttaisivat sitä:

- Indeksoidun sarakkeen täytyy olla erittäin valitseva.
- Indeksoi kaikki tärkeät vierasavaimet.
- Indeksoi kaikki predikaattisarakkeet.
- Indeksoi sarakkeet, joita käytetään liitoksissa.

4.3 Tietokantavastaavan keinot

Suorituskyvyn viritys sisältää SQL:n ja tietokannan resurssien säätöä. Ohjelmoijat hoitavat perinteisesti SQL:n ja tietokantavastaava virittää tietokannan parametreja. Usein tietokanta-vastaavat valittavat sitä, että vasteajat ovat hitaita, koska SQL on huonosti kirjoitettua.

Partitioidut taulut johtavat usein valtaviin suorituskyvyn paranemisiin ja niitä on myös helppo ylläpitää. Taulun partitiointi useampaan osaan vähentää tosiasiassa datan määrää, joka kyselyn täytyy läpikäydä. Jos taas taulu sisältää kymmeniä miljoonia rivejä, silloin se tulisi partitioida. Oraclen 10g -versiossa on viisi eri tapaa partitioinnille.

Materialisoituja näkymiä olisi hyvä käyttää silloin kun dataa on paljon, koska ne auttavat parantamaan vasteaikoja. Materialisoidut näkymät ovat objekteja, joissa on dataa – normaalisti yhteenvetotietoja toisista tauluista. Raskaat liitosoperaatiot voidaan myös tehdä etukäteen ja tallentaa ne materialisoituina näkyminä. Materialisoidut näkymät vähentävät tarvetta monimutkaisille kyselyille, koska niissä voidaan etukäteen laskea kokonaisuudet. Liitos kahden suuren taulun välillä ja datan kokoaminen ovat kalliita operaatioita resurssien käytölle. Muita tapoja, joilla tietokantavastaava voi auttaa parantamaan kyselyiden suorituskykyä, ovat esimerkiksi: pakkaus, kyselysuunnitelmien lukitseminen (outline), rinnakkainen suoritus, tilastotietojen keräys/päivitys ja histogrammit.

4.4 Viritysvälineet

SQL:n viritysvälineet ovat tärkeitä työvälineitä virittäjälle, niiden avulla voidaan etsiä hyviä suoritusstrategioita ja tuotantokannoissa ne ovat hyviä reagoivaan viritykseen. Välineet voivat antaa hyvän arvion arvioiduista resurssien käytöistä.

4.4.1 EXPLAIN PLAN

EXPLAIN PLAN -väline auttaa virittämään SQL-kyselyjä näyttämällä suoritussuunnitelman, jonka Oraclen optimoija on kyselylle valinnut. Virityksen aikana kyselylle voidaan kokeilla erilaisia muotoja ja vihjeitä. EXPLAIN PLAN on erinomainen työkalu silloin, kun viritys tehdään kokeilemalla erilaisia vaihtoehtoja, koska se antaa heti näkyviin sen, kuinka muutos vaikutti kyselyn suoritussuunnitelmaan. Tämä väline on siinäkin suhteessa hyvä, että se antaa ainoastaan suoritussuunnitelman, sen sijaan että se ajaisi kyselyn, joka nopeuttaa optimointiprosessia huomattavasti. Välineen käyttämiseksi vaaditaan, että käyttäjä ymmärtää suoritussuunnitelman ja sen kuinka erilaiset liitos- ja hakutavat vaikuttavat kyselyn nopeuteen. Suorituksen tulos menee tauluun plan_table, josta se voidaan hakea normaalilla kyselyllä. Myös jotkut välineet, kuten esimerkiksi Oracle Enterprise Manager, osaa hakea sen suoraan. Lopputulos on suoraan sama kuin kustannusperusteinen optimoija olisi suorittanut kyselyn. EXPLAIN PLAN suoritetaan seuraavalla tavalla:

```
EXPLAIN PLAN SET statement_id='testi' INTO plan_table FOR SELECT
* FROM Emp;
```

Tulos voidaan hakea esimerkiksi seuraavasti:

```
SQL> set lines 130
SQL> set pages 0
SQL> SELECT * FROM table(DBMS_XPLAN.DISPLAY);
Plan hash value: 1445457117
```

```
-----------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows   | Bytes | Cost (%CPU)| Time   |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |   107 |  6634 |      3 (0)|00:00:01 |
|   1 |  TABLE ACCESS FULL| EMPLOYEES |   107 |  6634 |      3 (0)|00:00:01 |
-----------------------------------------------------------------------------
```

8 rows selected.

4.4.2 Autotrace

Autotrace-väline automatisoi EXPLAIN PLAN:in tuottamisen automaattisesti aina kun kysely suoritetaan SQL*Plus:lla. Oletuksena autotrace-oikeudet on tunnuksilla SYSTEM ja SYS. Mikäli muilla tunnuksilla halutaan käyttää Autotrace-välinettä, niin niille täytyy antaa oikeudet siihen. Autotracen käyttöä suunniteltaessa pitää ensin luoda taulu suunnitelmaa varten ellei sitä ole jo luotu. Autotrace voidaan laittaa päälle eri optioin:

- SET AUTOTRACE ON EXPLAIN: tämä tuottaa suoritussuunnitelman, mutta ei aja itse kyselyä.
- SET AUTOTRACE ON STATISTICS: näyttää ainoastaan suorituksen tilastotiedot kyselylle.
- SET AUTOTRACE ON: tämä näyttää molemmat yllä olevista.

Tilastotiedot ovat erityisen hyödyllisiä silloin, kun kyselyä halutaan optimoida esimerkiksi levyhakujen ja lajittelujen suhteen.

4.4.3 SQL Trace ja TKProf

SQL Trace on Oraclen työkalu, joka auttaa seuraamaan SQL:n suoritusta ja TKProf taas auttaa formatoimaan luettavaan muotoon SQL Tracen tuotoksen. SQL Trace -työkalu kerää varsinaisen suorituksen lopputuloksen tiedostoon. Joskus esimerkiksi ei tiedetä tarkkoja SQL-

lauseita, joita suoritetaan, koska lauseet muodostetaan dynaamisesti; tällöin SQL Trace voi kerätä lauseet. SQL Trace voi seurata seuraavia asioita:

- CPU- ja kuluneen ajan.
- Jäsentämisten ja ajokertojen määrän kyselyille.
- Fyysisten ja loogisten lukujen määrän.
- Suoritussuunnitelman kaikille SQL-lauseille.
- Library cache:sta löytökertojen suhteen.

SQL Trace antaa paljon tärkeää tietoa sekä resurssien käytöstä että kyselyjen tehokkuudesta. Välineellä saadaan myös hyvää tietoa siitä, jos jokin lause suoritetaan esimerkiksi liian usein. Väline auttaa muodostamaan käsityksen siitä, kuinka paljon CPU-aikaa kyselyt vievät ja kuinka paljon I/O:ta käytetään. EXPLAIN PLAN, joka on optiona SQL Trace:ssa kertoo jokaiselle suunnitelman riville, kuinka monta riviä se palauttaa, mikä auttaa löytämään ne kohdat, jotka ovat raskaimpia, ja joihin täytyy kiinnittää eniten huomiota kyselyssä. SQL Trace asetetaan päälle siten, että suoritetaan komento:

```
ALTER SESSION SET sql_trace = true;
```

Komento asettaa SQL Trace:n päälle käynnissä olevan session ajaksi.

5 YHTEENVETO

Tärkeintä SQL-kyselyiden optimoinnissa on se, että ohjelmoija tarkastaa tekemänsä lauseet ja niiden suorituskyvyn jo suunnitteluvaiheessa. Tällä toimenpiteellä estetään suurin osa tehottomuusongelmista sovelluksissa jo heti alkuvaiheessa. Oracle tarjoaa kehittäjälle hyvät välineet SQL-lauseiden suorituskyvyn analysointiin ja näitä apuna käyttämällä pääsee jo pitkälle. Toisena tärkeänä asiana näkisin indeksoinnin merkityksen tehokkuudelle. Indeksit voivat tuoda merkittäviä parannuksia suorituskykyyn silloin, kun ne on suunniteltu oikein ja niitä ei ole liikaa. Täytyy myös muistaa, että ohjelmoija ei ainoastaan vastaa kyselyjen tehokkuudesta ja optimoinnista, vaan tässä työssä on hyvä käyttää apuna tietokantavastaavaa.

Vaikka kustannusperusteinen optimoija Oraclessa toimiikin useimmissa tapauksissa erittäin hyvin – jos tilastotiedot ovat ajan tasalla – on hyvä muistaa optimoijan oletukset datan tasaisesta jakautumisesta sekä muista tunnetuista optimoijan puutteista. Itse optimoinnin tarve näyttäisi kuitenkin vuosi vuodelta vähenevän, koska optimoijien älykkyys kasvaa. Ne pystyvät itse automatisoidusti keräämään statistiikkaa ja todennäköisesti tulevaisuudessa myös histogrammitietoja, jolloin itse optimoinnin tarve vähenee ja kenties katoaa lähes kokonaan. Histogrammitietojen automaattinen kerääminen auttaisi usein huomattavastikin kustannusperusteista optimoijaa tilanteissa, joissa data ei jakaudu tasaisesti ja dataa on paljon. Optimoija ei voi kuitenkaan korjata kyselyssä olevia loogisia virheitä, vaan kyselyn tekijän täytyy tietää mitä hän tekee ja tämäntyyppisiä virheitä tuskin koskaan voidaan korjata automatisoidusti, koska kone ei voi tietää tällaisessa tilanteessa mitä käyttäjä tarkoitti.

Koodaajan on hyvä tuntea tulevaisuudessakin relaatioalgebran ja optimoinnin periaatteet, jotta hän ymmärtää kuinka kyselyitä tehdään tehokkaasti ja miksi optimoija tekee sellaisia päätöksiä kuin se tekee.

VIITELUETTELO

Ala05 Alapati, S., R.: *Expert Oracle Database 10g Administration*. Apress, 2005.

Bur01 Burleson, D., K.: *Oracle High-Performance SQL Tuning*. McGraw-Hill, 2001.

CB05 Connolly, T., Begg, C.: *Database Systems, A Practical Approach to Design, Implementation, and Management*. Addison-Wesley, Harlow, England, 2005.

EN00 Elmasri, R., Navathe, S., B.: *Fundamentals of Database Systems*. Addison-Wesley, 2000.

RDF06 Rivero, L., C., Doorn, J., H., Ferraggine, V., E.: *Advanced query optimization*, Idea group publishing, 2006.

IC91 Ioannidis, Y.,E., Christodoulakis, S., On the propagation of errors in the size of join results, *Proceedings of the 1991 ACM SIGMOD Conference*, 268-277.

PK02 Koikkalainen, P.: *Tietotekniikan perusteet,* 2002 http://erin.mit.jyu.fi/pako/kurssit/perusteet/kirja/node2.html (16.4.2007).

MLR03 Markl, V., Lohman, G., M., Raman, V.: LEO: an autonomic query optimizer for DB2. *IBM Systems Journal*, 42:1, 2003, 98–106.