Optimizacija upita kod Oracle baze podataka

Seminarski rad

Sistemi za upravljanje bazama podataka

Student: Uroš Pešić 1465 Profesor: Dr Aleksandar Stanimirović

Elektronski fakultet, Niš
April 2024.

Sadržaj

UVOD

SQL (*Structured Query Language*) je struktuirani upitni jezik namenjen za manipulaciju podacima i strukturama u sistemima za upravljanje relacionim bazama podataka (*Relational Database Management Systems – RDBMS*). Bez obzira na to što je nastao još sedamdesetih godina prošlog veka, SQL je i danas nezamenljiv i predstavlja standard u ovom domenu.

Glavni razlog za njegovu veliku popularnost leži u činjenici da je SQL bio prvi upitni jezik **deklarativnog** tipa – upitima korisnik definiše *šta* želi kao rezultat, tj. *šta* želi da uradi, za razliku od imperative paradigme koja je do tada bila zastupljena, kod koje je bilo potrebno definisati *kako* dobiti rezultat. Na ovaj način je programerima, analitičarima i ostalim korisnicima relacionih baza, pisanje upita značajno pojednostavljeno – čitava unutrašnja organizacija DBMS-a je apstrahovana i moguće je manipulisati podacima korišćenjem intuitivnih naredbi, koje su slične prirodnom jeziku. Zahvaljujući navedenim prednostima i jednostavnoj sintaksi SQL se danas koristi i kao osnova za mnoge upitne jezike za rad sa NoSQL bazama podataka (kao najočigledniji primer imamo CQL – Cassandra Query Language, i malo manje sličan Cypher – za pisanje upita nad Neo4J bazom podataka).

Sa druge strane, deklarativna priroda SQL-a uvodi komplikacije u procesu izvršenja korisničkih upita. Kako se dobijanje željenih rezultata postiže pisanjem upita višeg nivoa, odgovornost za definisanje načina na koji će oni da se izvrše se prebacuje sa programera na DBMS. Korisnički upit je potrebno skenirati (izdvojiti ključne reči, tzv. tokene), parsirati - proveriti da li su ispoštovana sva sintaksna pravila, a zatim validirati – proveriti da svi navedeni atributi i relacije semantički imaju smisla. Nakon ovih koraka, upit je potrebno prevesti u izraz proširene relacione algebre, koja predstavlja matematičku osnovu za rad sa relacionim modelima. Svi upiti su interno predstavljeni kao izrazi proširene relacione algebre, koje je dalje potrebno obraditi i optimizovati kako bi se omogućilo njihovo optimalno izvršenje, kako u pogledu vremena, tako i u pogledu računarskih resursa. Danas, kada su relacione baze podataka neizostavni deo gotovo svih informacionih sistema, u kojima se često čuva na stotine miliona slogova, mehanizam optimizacije upita predstavlja ključni deo svakog DBMS-a.

U narednim poglavljima u ovom seminarskom radu je akcenat upravo na različitim tehnikama za **optimizaciju upita** kod relacionih DBMS-ova (u prvom delu), i dat je pregled tehnika koje Oracle DBMS koristi u ove svrhe (u drugom delu). Kao preduslov za naredna poglavlja, potrebno je poznavati osnovne operacije proširene relacione algebre, kao i neke algoritme i tehnike za obradu SQL upita, mada su neke od njih pomenute i u nastavku.

1 - OPTIMIZACIJA UPITA

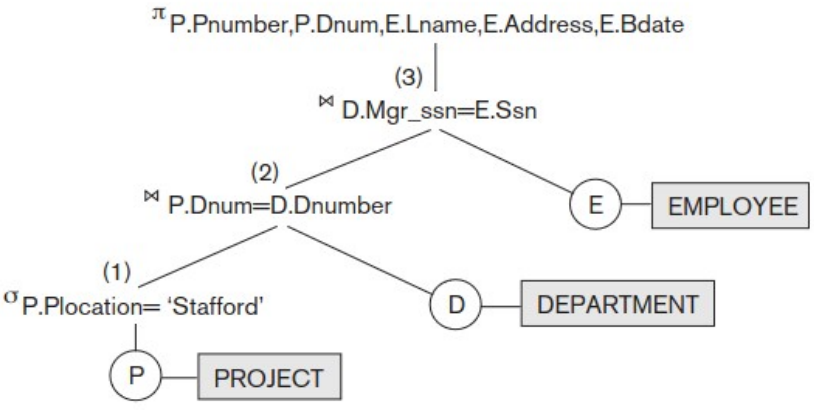
Optimizacija upita predstavlja proces određivanja *najbolje dostupne strategije* za izvršenje korisničkog upita[1]. Najbolja dostupna strategija za izvršenje često nije i najoptimalnija moguća – primarni cilj optimizacije je da se odredi dovoljno efikasan plan izvršenja, na osnovu dostupih informacija o relacijama u bazi (kao što su npr. broj različitih vrednosti za određene atribute u relaciji, selektivnosti različitih relacija za definisane uslove selekcije, itd.) u prihvatljivom vremenskom periodu. Optimizacija upita je zadatak komponente DBMS-a koja se zove **optimizator**.

U narednih nekoliko poglavlja tad je pregled tehnika za optimizaciju upita u relacionim DBMS-ovima – počev od različitih heuristika i definisanih opštih pravila za optimizaciju, do metoda zasnovanih na kaznenim (eng. *Cost*) funkcijama, koje služe za procenu plana izvršenja, uzimajući u obzir konkretno stanje sistema.

1.1 – Stablo upita

Prvenstveno je potrebno definisati internu reprezentaciju upita u DBMS-u. Svaki SQL upit se u procesima skeniranja i parsiranja prevodi u izraz proširene relacione algebre. Ovaj izraz se predstavlja pomoću strukture podataka koja ima formu stabla i zove se **stablo upita**. Listovi ovog stabla predstavljaju početne relacije koje su deo FROM naredbe, dok interni čvorovi predstavljaju operacije proširene relacione algebre koje se nad njima izvršavaju. Svaka operacija definisana internim čvorovima se izvršava kada njeni operandi postanu dostupni. Tok izvršenja upita predstavljenog stablom se kreće od listova, ka korenskom čvoru. Kada se izvrši korenski čvor dobijena relacija predstavlja rezultat postavljenog upita. Na sledećoj slici je ilustrovan primer stabla upita za sledeći upit [1]:

Q1: SELECT *P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate*
FROM *PROJECT P, DEPARTMENT D, EMPLOYEE E*
WHERE *P.Dnum=D.Dnumber* ***AND*** *D.Mgr_ssn=E.ssn* ***AND*** *P.Location=`Stanfford`;*



Slika 1: Primer stabla upita za Q1

Ovo je jedna od mogućih *inicijalnih* reprezentacija upita Q1.

Treba napomenuti da, osim stabala, postoji i reprezentacija upita korišćenjem drugih struktura – kao što su grafovi upita. Kod grafova upita, definišu se čvorovi relacija i čvorovi sa konstantnim vrednostima (npr. string iz nekog od uslova selekcije), a potezi između čvorova predstavljaju relacione operatore. Međutim, kako kog grafova upita ne postoji jasna definicija redosleda izvršenja ovih operatora, za potrebe optimizacije je pogodnija reprezentacija upita u obliku stabla. Zbog toga neće biti naveden primer graf reprezentacija upita Q1.

1.2 – Heuristička optimizacija stabala upita

Ovakav pristup optimizacije zasniva se na primeni definisanih pravila transformacije, čijom primenom prevodimo inicijalno stablo upita u oblik koji je najčešće[1] efikasniji za izvršenje. Primenom svake transformacije, dobija se novo stablo, koje je *semantički ekvivalentno* inicijalnom – predstavlja isti upit i daje iste rezultate, jedino se razlikuje redosled izvršenja relacionih operacija. Transformacije koje se koriste za optimizaciju inicijalnog upita su sledeće:

1. **Nadovezivanje selekcija:** Uslov selekcije, koji predstavlja konjunkciju više manjih uslova možemo zameniti predstaviti uzastopnim primenama operacije selekcije :

   $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) = \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R)))$

   Ukoliko imamo disjunkciju uslova, možemo je transformisati u konjunkciju korišćenjem DeMorganovih pravila Bulove algebre.

2. **Komutativnost operacije selekcije σ –** $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$

3. **Nadovezivanje projekcija:** Sekvencu operacija projekcije možemo zameniti samo prvom projekcijom iz tog niza:

   $\pi_{list1}(\pi_{list2}\ldots(\pi_{listn}(R))) = \pi_{list1}(R)$

4. **Komutativnost operacija selekcije i projekcije:** Ukoliko operacija selekcije uključuje samo atribute iz liste atributa projekcije, nije bitno koju od operacija prvu primenjujemo.

5. **Komutativnost operacija prirodnog spoja (⋈) i Dekartovog proizvoda (x)**.

6. **Distributivnost operacije selekcije prema operaciji ⋈ (ili x):** Ukoliko se operacija selekcije odnosi na jednu od relacija koje učestvuju u JOIN operaciji, moguće je prvo primeniti operaciju selekcije, a zatim operaciju prirodnog spoja (ili Dekartovog proizvoda):

   $\sigma_{c1}(R \bowtie S) = \sigma_{c1}(R) \bowtie S$

   Operaciju prirodnog spoja moguće je zameniti Dekartovim proizvodom (X), osobina i dalje važi.

7. **Distributivnost operacije projekcije prema operaciji ⋈ (ili x):** Označimo je lista atributa koje zadržavamo posle projekcije $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$, pri čemu su atributi A_i iz

1 Kako su ova pravila zasnovana na heuristikama, a ne uzimaju u obzir informacije o konkretnim relacijama i atributima, može se desiti, mada retko, da primenom nekog od ovih pravila dobijemo novi upit koji ima nešto lošije performanse, nego pre primene tog pravila.

relacije R, a B_i iz relacije S, tada možemo primeniti distributivnost projekcije na sledeći način:

$\pi_{c1}(R \bowtie S) = \pi_{A1, A2, \ldots, An}(R) \bowtie \pi_{B1, B2, \ldots, Bm}(S)$

8. **Komutativnost operacija unije (U) i preseka (∩)**
9. **Asocijativnost operacija ⋈, X, U,** ∩
10. **Distributivnost operacije selekcije u odnosu na skupovne operacije:** Filtriranje skupova u skladu sa uslovom operacije selekcije, možemo izvršiti i pre i posle primene skupovne operacije:

    $\sigma(R \text{ op } S) = \sigma(R) \text{ op } \sigma(S)$, gde op označava bilo koju skupovnu operaciju (presek, unija, razlika)

11. **Konverzija sekvence (σ, X) u prirodni spoj:** Ovo je jedno od važnijih pravila. Operacija prirodnog spoja u suštini predstavlja operaciju dekartovog proizvoda između dve relacije nakon koje se vrši selekcija sa uslovom jednakosti između para (ili više parova) atributa. Iako su obe ove operacije vremenski dosta zahtevne, uvek je bolje koristiti operaciju prirodnog spoja – zahvaljujući različitim mogućnostima implementacije je moguće postići značajno bolje performanse prilikom izvršenja, naročito u slučaju da postoji primary index ili clustering index za atribut jedne od relacija koji se javlja u uslovu spoja. Takođe i algoritmi kao što su sort-merge join ili partition-hash join nam nam omogućavaju da operaciju spoja izvršimo u vremenskoj složenosti koja je manja od O(m * n), gde m i n predstavljaju broj slogova dve relacije koje spajamo. Sa druge strane, kod Dekartovog proizvoda je potrebno upariti svaki slog prve relacije sa svakim slogom druge relacije (vremenska složenost je O(m * n)), da bi na kraju dobar deo kombinacija bio filtriran uslovom selekcije: uzmimo ekstremni primer kada su relacije povezane vezom 1:1 – tada će maksimalan broj slogova u spoju biti min(m, n), što je mnogo manje od m * n, naročito ako se uzme u obzir da brojevi m, n mogu biti jako veliki.
12. **Distributivnost projekcije u odnosu na uniju:**

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

13. **Primena selekcije samo na jedan argument operacije preseka:** Ukoliko su svi atributi koji su prisutni u uslovima selekcije iz jedne relacije, lakše je prvo filtrirati samo tu relaciju, a zatim tražiti presek (zato što proveravamo uslov za manji broj slogova, a zatim prilikom traženja preseka imamo manji broj kandidata za rezultujući skup, jer smo smanjili kardinalnost jednog od skupova).

Navedene transformacije (i još neke trivijalne transformacije koje su izostavljene) koristimo u **algoritmu za heurističku algebarsku optimizaciju upita**:

1. Sve uslove selekcije koji predstavljaju konjunkciju uslova razdvojiti na niz operacija selekcije korišćenjem pravila (disjunkciju je moguće transformisati u konjunkciju korišćenjem DeMorganovih pravila). Na ovaj način je moguće lakše manipulisati zasebnim uslovima i nezavisno ih premeštati u okviru stabla upita.

2. Koristeći pravila 2, 4, 6, 10, 13 potreno je pomeriti sve operacije selekcije, što je moguće niže u stablu, u zavisnosti od atributa koji se javljaju u uslovu selekcije (ukoliko se selekcija vrši nad atributima jedne relacije, onda je selekciju moguće pomeriti do samog lista stabla koji predstavlja ovu relaciju).

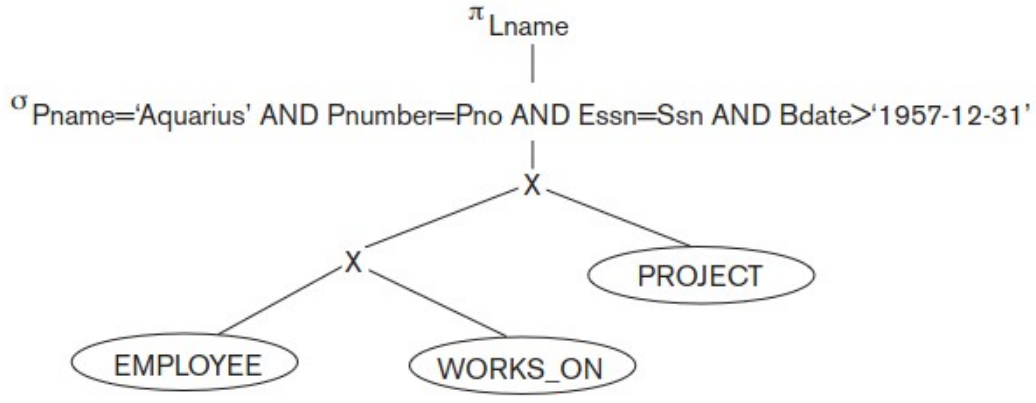
3. Korišćenjem asocijativnosti binarnih operacija (pravila 5 i 9), preurediti listove stabla upita tako da redosled operacija selekcije bude takav da se prvo izvrše one selekcije koje su *najrestriktivnije* (na osnovu procene o selektivnosti, ili na osnovu broja slogova u relaciji nakon filtriranja). Pritom je potrebno voditi računa da promenom redosleda relacija ne stvorimo potrebu za korišćenjem operacije Dekartovog proizvoda – koja je vremenski najzahtevnija.

4. Korišćenjem pravila 11 pretvoriti svaku operaciju Dekartovog proizvoda, koja je praćena uslovom selekcije u prirodni spoj (naravno ukoliko je uslov selekcije u stvari uslov spoja, a ne uslov za filtriranje).

5. Korišćenjem pravila 3, 4, 7, 12 premestiti sve operacije projekcije, što je moguće niže u stablu. Ukoliko je potrebno moguće je definisati nove operacije projekcije po potrebi, u zavisnosti od potrebe za određenim atributima. Nakon svake projekcije potrebno je zadržati samo one atribute koji su neophodni za tekuću, i sve naknadne operacije.

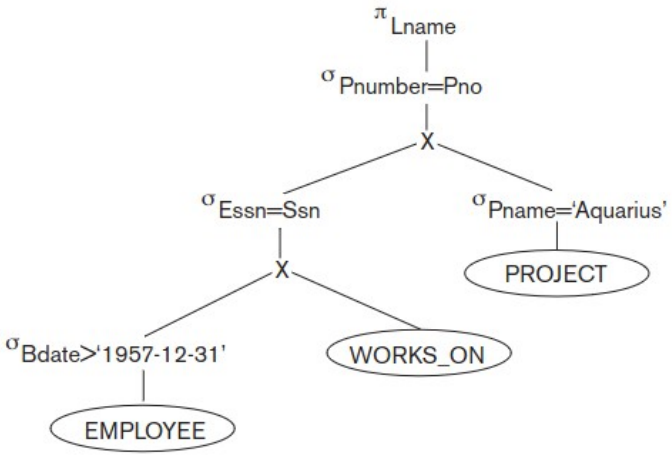
Generalno, ideja je da se sve operacije koje smanjuju veličinu privremenih rezultata izvrše što ranije – selekcije smanjuju broj vrsta u međurezultatima, dok projekcije smanjuju broj atributa. Takođe, najrestriktivnije JOIN i select operacije je potrebno izvršiti što ranije iz istog razloga. Primeri *inicijalnog stabla* i *finalnog stabla* koje se dobija primenom navedenog algoritma optimizacije nad upitom Q2, dati su na slikama 2 i 5.

***Q2: SELECT** E.Lname*
***FROM** EMPLOYEE E, WORKS_ON W, PROJECT P*
***WHERE** P.Pname=“Aquarius“**AND** P.Pnumber=W.Pno **AND** E.Essn=W.ssn **AND** E.Bdate > „1957-12-31“;*



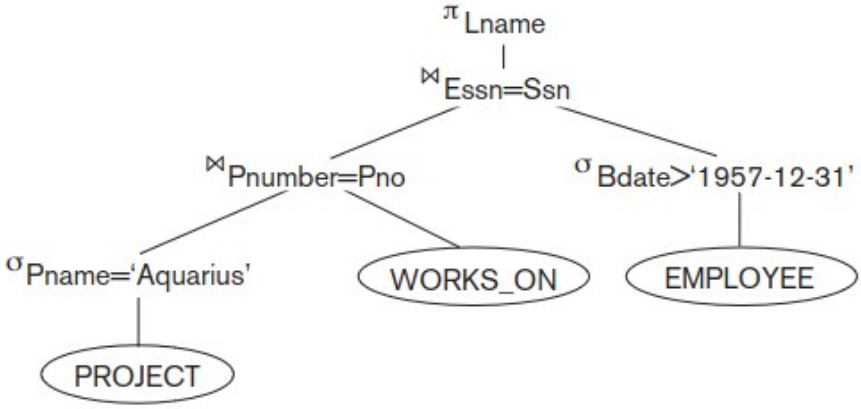
Slika 2: Inicijalno stablo upita za upit Q2

Primenom 1. koraka i premeštamo sve operacije selekcija što bliže listovima (selekcije koje vrše filtriranje stoje uz same relacije i prve se izvršavaju, selekcije koje predstavljaju uslov spoja stoje odmah nakon operacije x) – slika 3.



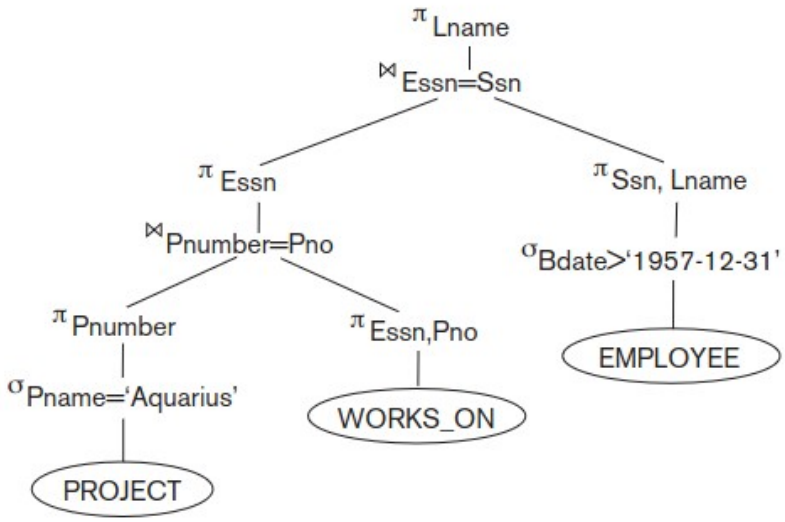
Slika 3: Primena 1. koraka algoritma heurističke optimizacije

Primenom 2. i 3. koraka dobijamo stablo sa slike 4. Treba primetiti da je promenjen redosled listova stabla (početnih relacija), kako bi se prvo izvršila operacija selekcija nad relacijom PROJECT, koja će kao rezultat generisati samo jedan slog (ili par njih pod uslovom da više projekata, npr. u različitim departmanima, može nositi isto ime), za razliku od selekcije nad relacijom EMPLOYEE, koja će sigurno vratiti veći broj vrsta kao rezultat. Zbog ovakvog preuređenja će prvo biti izvršena operacija spoja između relacija PROJECT i WORKS_ON, a zatim se ovaj privremeni rezultat spaja sa relacijom EMPLOYEE.



Slika 4: Primena koraka 2 i 3

Konačno, primenom koraka 5. dobijamo finalno optimizovano stablo za upit Q2:



Slika 5: Finalno optimizovano stablo

Heurističkom optimizacijom, kao što je već napomenuto, primenjuje pravila koja u *opštem* slučaju dovode do optimalnijih stabala, bez uvida u to kakve bi bile performanse transformisanog upita u konkretnoj situaciji. Za bolji uvid u stvarne performanse različitih stabala, sa različitim implementacijama relacionih operacija, heurističke metode se koriste u kombinaciji sa metodama optimizacije *na osnovu troškova izvršenja (eng.* ***Cost-based****).*

1.3 - Cost-based optimizacija upita

Da bi izgenerisao najefikasniji dostupni plan izvršenja SQL upita, optimizator mora da uporedi efikasnost različitih mogućih strategija. Zbog toga je potrebno definisati kvantitativnu meru „kvaliteta“ plana izvršenja, koja će da se koristi za poređenje različitih planova. Ova mera predstavlja cenu (cost) izvršenja plana i obrnuto je srazmerna vremenu performansama izvršenja. Vrednost cost funkcije može da zavisi od više komponenti:

1. **Cena pristupa sekundarnom skladištu (disku):** Predstavlja cenu transfera blokova podataka sa fajla na disku u glavnu memoriju. Vrednost ove komponente zavisi od broja blokova koje je potrebno preneti, od postojanja struktura pristupa za konkretni fajl (indeksi, tipovi indeksa, itd.), od toga da li je fajl sortiran po vrednosti nekog atributa i slično.
2. **Cena skladištenja privremenih rezultata na disku.**
3. **Cena izvršenja (CPU cena):** Odnosi se na vreme izvršenja konkretnih operacija nad podacima koji su u glavnoj memoriji. U te operacije spadaju, pretrage slogova, sortiranje, spajanje, izračunavanje funkcija agregacija, itd.
4. **Cena zauzeća memorije:** Predstavlja broj blokova u glavnoj memoriji koji je potreban za izvršenje upita.

5. **Cena komunikacije:** Ima smisla kod distribuiranih baza podataka, gde predstavlja cenu troškova koji se javljaju usled problema preraspoređivanja i prikupljanja podataka između distribuiranih particija.

U zavisnosti od tipa baze podataka, moguće je koristiti samo neke od ovih komponenti. U velikim bazama, najbolje poboljšanje performansi se dobija smanjivanjem cene pristupa sekundarnom skladištu. Za manje baze, kod kojih često svi blokovi mogu biti prebačeni sa diska u glavnu memoriju, primarni cilj je smanjiti cenu izvršenja operacija (CPU cenu). Sa druge strane, kod distribuiranih baza najveći gubitak performansi se javlja usled dodatnih troškova komunikacije, pa je najveći akcenat na minimizaciji ove komponente.

Zahvaljujući ovakvoj postavci, problem optimizacije upita možemo posmatrati kao klasični problem minimizacije cost funkcije u prostoru stanja (stanja su predstavljena različitim planovima izvršenja). Kako ovaj problem nije trivijalan, često je moguće proceniti samo konačan broj stanja, čime se ne garantuje dobijanje *globalno* najoptimalnijeg rešenja.

1.3.1 - DBMS katalog

Za potrebe izračunavanja vrednosti cost funkcije za potencijalni plan izvršenja optimizator koristi različite informacije o stanju sistema. Ove informacije se čuvaju u katalogu informacija DBMS-a. Za svaki fajl je potrebno zapamtiti **broj slogova (r), prosečnu veličinu sloga (R)**, **broj blokova (b)** na koje je fajl podeljen, moguće je pamtiti **broj slogova po bloku (bfr – blocking factor)** itd. Zatim, potrebno je čuvati informacije o samoj organizaciji fajlova - da li su sortirani i po kom atributu, kakvi indeksi postoje, i**nformacije o primarnim, sekundarnim i klastering indeksima**, **broj nivoa indeksa** za slučaj da se koriste indeksi sa više nivoa, itd. Što se tiče samih relacija, za potrebe optimizacije se koriste različite statistike, među kojima su najznačajnije: **broj jedinstvenih vrednosti atributa A u relaciji R (NDV(A, R)), selektivnost (sl) –** procenat slogova koji zadovoljava neki uslov jednakosti, za dati atribut, **kardinalnost selekcije (s = sl * r)** – prosečan broj slogova koji zadovoljava uslov selekcije (za uslov jednakosti nad nekim atributom). DBMS takođe kreira i čuva posebne strukture sa informacijama o statističkoj raspodeli atributa u relacijama, koje se nazivaju **histogrami**. Zahvaljujući histogramima je moguće bolje aproksimirati selektivnost za različite atribute (bez njih bismo morali da pretpostavimo da svi atributi imaju uniformnu raspodelu – što najčešće nije slučaj). Histogrami vrše diskretizaciju atributa na više podintervala, pri čemu se pretpostavlja uniformna raspodela vrednosti u okviru intervala. Zbog toga je za bolju aproksimaciju raspodele, potrebno podeliti opseg vrednosti na atributa veliki broj intervala.
Zadatak DBMS-a je da vodi evidenciju o svim ovim metrikama i da to čini na efikasan način – vrednosti ovih parametara se često menjaju i kada bi se prilikom svake promene u bazi oni ažurirali, brzo bi došlo do preopterećenja resursa. Zbog toga se najčešće ove vrednosti periodično ažuriraju, dok se manja odstupanja metrika od stvarnih vrednosti tolerišu.

1.3.2 - Cost funkcija za operaciju selekcije

U ovom i narednom poglavlju dat je primer izračunavanja cost funkcije koja u obzir uzima samo cenu pristupa disku, prilikom učitavanja blokova sa diska u glavnu memoriju. Navedeni su primeri algoritama za implementaciju operacija selekcije koji su relevantni i koriste se i kod Oracle DBMS-a[2].

Za svaku od navedenih metoda će cena biti izražena u broju blokova koje je potrebno preneti sa diska u glavnu memoriju:

- **Metoda linearne pretrage (brute-force pristup) –** Uslov selekcije se proverava za svaki slog u fajlu, što znači da svi blokovi fajla moraju biti učitani u glavnu memoriju. Cena ovog algoritma je $C_1 = b$, gde *b* predstavlja broj blokova fajla na disku. U slučaju uslova jednakosti sa ključem, cena se može aproksimirati kao $C_1 = b / 2$, jer je u proseku potrebno pretražiti samo polovinu blokova pre nego što naiđemo na odgovarajući slog.
- **Binarna pretraga –** Za binarnu pretragu je potrebno da fajl bude sortiran po atributu koji je deo uslova selekcije. Tada je potrebno pribaviti $log_2 b$ blokova sa diska u glavnu memoriju, pre nego što se pronađu odgovarajući slogovi, i dodatno jos *ceiling(s / bfr) – 1* blokova u slučaju da je broj slogova koji zadovoljavaju uslov veći od broja slogova u jednom bloku (s označava kardinalnost selekcije), pa je ukupna cena: $C_2 = log_2 b + ceiling(s / bfr) - 1$. U slučaju da se vrši selekcija sa uslovom jednakosti nad atributom koji je primarni ključ, imamo s = 1, tj. $C_2 = log_2 b$.
- **Selekcija sloga na osnovu primarnog indeksa –** Ukoliko je primarni indeks implementiran kao indeks sa više nivoa, gde je broj nivoa *x*, za traženje sloga sa datom vrednošću primarnog ključa je potrebno učitati $C_3 = x + 1$ blokova – po jedan indeksni blok za svaki nivo, i jedan dodatni blok za pribavljanje konkretne vrste na osnovu adrese u indeksu.
- **Selekcija sloga na osnovu heširanog ključa –** u slučaju da je fajl u kome se slogovi nalaze heširan po primarnom ključu, za pribavljanje sloga po ključu iz ovakvog fajla je potrebno pribaviti samo 1 blok – onaj koji odgovara heširanoj vrednosti ključa iz uslova selekcije. $C_4 = 1$.
- **Selekcija većeg broja slogova korišćenjem clustering indeksa –** Slično kao kod primarnog indeksa, potrebno je pribaviti *x* indeksnih blokova, nakog čega iz indeksa osnovnog nivoa dobijamo adresu bloka u kome se nalazi prvi slog koji ispunjava uslov jednakosti iz selekcije. Pošto clustering indeks može biti definisan nad atributima koji nemaju unique vrednosti, moguće je dobiti više slogova kao rezultat, pa je potrebno pribaviti ukupno još *ceiling(s / bfr)* blokova. $C_5 = x + ceiling(s / bfr)$.
- **Selekcija korišćenjem sekundarnog indeksa –** Prvi slučaj je da imamo uslov jednakosti nad nekim atributom. Ukoliko je sekundarni indeks definisan nad atributom koji nije ključ i nema unique vrednosti, cena pristupa sekundarnom skladištu je u najgorem slučaju $C_5 = x + 1 + s$: *x* pristupa indeksnim blokovima, 1 blok koji sadrži pokazivače na konkretne

2 Implementacija ovih algoritama ovde nije obrađena, jer spada u domen obrade upita, mada može da se nasluti na osnovu izračunate cost funkcije.

slogove u fajlu, a kako svi ovi slogovi mogu biti u različitim blokovima, potrebno je učitati još maksimalno *s* blokova (za svaki slog rezultata po jedan). Druga mogućnost je da je uslov selekcije predstavlja nejednakost (<, >, <=, >=). Tada možemo grubo pretpostaviti da će polovina svih slogova zadovoljiti uslov, pa je cena $C_5 = x + r / 2 + x_k / 2$, gde x_k predstavlja broj indeksnih blokova u prvom nivou, a *r* ukupan broj slogova. Ova gruba procena se može poboljšati korišćenjem histograma, jer je na osnovu raspodele atributa moguće polje proceniti i selektivnost.

- **Selekcija korišćenjem bitmap indeksa –** Bitmap vektor za vrednost sa kojom se poredi atribut u uslovu selekcije (uslov je tipa jednakosti) ima *r* bitova, tj *r / 8* bajtova. U zavisnosti od veličine bloka u bajtovima, koju ćemo označiti sa v_b, cena izvršenja je $C_6 = r / 8v_b + s$. Član *s* se javlja zbog toga što, u najgorem slučaju, svaki slog koji zadovoljava uslov selekcije može biti u različitom bloku u fajlu.

1.3.3 - Cost funkcija za operaciju JOIN

Da bi aproksimacija cene izvršenja JOIN operacije imala smisla, potrebno je proceniti koliko slogova će imati rezultat ove operacije. Ova metrika se zove **selektivnost spoja** (*eng.* ***Join selectivity - js***) obično definiše kao odnos broja slogova rezultata JOIN operacije, i broja slogova operacije Dekartovog proizvoda, nad istim relacijama, tj:

$$\boldsymbol{js} = |R \bowtie_c S| / |R \times S| = |R \bowtie_c S| / (|R| * |S|), \qquad 0 \leq js \leq 1,$$

U slučaju kada je *c* uslov jednakosti (tada se operacija naziva *EQUIJOIN*), selektivnost spoja se može izračunati se aproksimirati na sledeći način:

$js = 1 / max(NDV(A, R), NDV(B, S))$, gde NDV(X, Y), označava broj različitih vrednosti atributa X u relaciji Y.

Ovaj izraz predstavlja grubu aproksimaciju selektivnosti, jer pretpostavlja uniformnu raspodelu atributa u relaciji. Za tačniju aproksimaciju je potrebno modifikovati formulu shodno informacijama o atributima dobijenih iz histograma.

Primeri cost funkcije (kao i u prethodnom poglavlju, posmatramo samo komponentu koja se odnosi na cenu pristupa sekundarnom skladištu) za različite algoritme spoja (pretpostavljamo da je u pitanju EQUIJOIN, tj. operacija je oblika $R \bowtie_{A=B} S$) su sledeće:

- **Implementacija sa dve ugnježdene petlje (brute-force) –** Pretpostavimo da se iteracija po relaciji R (leva relacija u izrazu), vrši u spoljašnjoj, a iteracija po S (desna relacija) u unutrašnjoj petlji. Ukoliko imamo n_b blokova u glavnoj memoriji, tada će n_b - 2 blokova biti rezervisani za blokove fajla relacije R, 1 blok će biti rezervisan za blok fajla relacije S, dok će poslednji blok služiti za smeštanje delova rezultata. Cena izvršenja ovako implementirane operacije spoja je tada $C_1 = b_r + [ceiling(b_r / n_b – 2) * b_s] + (js * r_r * r_s) / bfr_{res}$, gde je r_i označava broj slogova relacije *i*, dok je bfr_{res} broj slogova po bloku za rezultujuću

relaciju. Poslednji sabirak u cost funkciji predstavlja cenu smeštanja rezultujućih blokova nazad na disk.

- **Ugnježdene petlje uz korišćenje indeksa za jednu relaciju –** Relaciju koja ima indeks za atribut koji se koristi u uslovu spoja možemo staviti u unutrašnju petlju, a zatim obilaziti slogove druge relacije u spoljašnjoj petlji. Za svaki njen slog spoljašnje relacije, korišćenjem indeksa tražimo slog unutrašnje relacije koji se poklapa po vrednosti atributa spoja. Cena izvršenja zavisi od tipa indeksa i definisana je formulom:

  $C_2 = b_r + (r_r * matching\_cost) + (js * r_r * r_s) / bfr_{res}$, gde *matching_cost* predstavlja cenu traženja slogova koji se podudaraju po vrednosti atributa spoja (pogledati deo za cenu selekcija sa različitim vrstama indeksa u prethodnom poglavlju) , npr. za slučaj da imamo sekundarni indeks je $matching\_cost = x + 1 + s_s$.

- **Spoj korišćenjem Sort-merge algoritma –** Prednost ovog algoritma je ta što se kroz blokove fajlova obe relacije prolazi istovremeno, u jednoj petlji, pa je cena izvršenja:

  $C_3 = b_r + b_s + (js * r_r * r_s) / bfr_{res} + sorting\_cost$

  Sorting_cost predstavlja cenu sortiranja fajlova obe relacije (koja je 0 ukoliko su fajlovi već sortirani po atributu iz uslova spoja). Cena sortiranja jednog fajla se može aproksimirati:

  $sorting\_cost = 2 * b + (2 * b * \log_{mf}(n_{runs}))$

  Da bismo objasnili ovakvu formulu cene, dato je kratko objašnjenje sortiranja fajla sort-merge algoritmom. U prvoj fazi – fazi sortiranja, se učitava n_{runs} blokova koji se sortiraju i upisuju nazad na disk. Ovaj postupak se ponavlja sve dok svih *b* blokova početnog fajla ne bude sortirano – odatle potiče prvi sabirak u cost funkciji za sortiranje. Zatim se po *mf* sortiranih pod-blokova istovremeno spaja u jedan veći sortirani blok. Postupak se ponavlja sve dok ceo fajl ne bude sortiran (kako je na kraju svih *b* blokova spojeno, grubo možemo definisati drugi sabirak kao cenu spajanja pod-blokova).

- **Spoj korišćenjem hash-join algoritma –** Kod hash-join algoritma se fajlovi obe relacije dele u jednak broj particija, na osnovu vredosti hash funkcije primenjene na atribut spoja. Zatim se u svakoj iteraciji spajaju odgovarajuće particije svakog fajla (za fajlove podeljene u M particija imaćemo M iteracija). Kako se svi blokovi fajla u fazi particionisanja učitavaju, particionišu a zatim smeštaju nazad na disk, cena ove faze je $2 * b_r + b_s$. Zatim u fazi poklapanja slogova se ponovo svi blokovi ponovo učitavaju u glavnu memoriju za potrebe upoređivanja. Cena ove faze je $b_r + b_s$. Ukupna cena ovog algoritma je:

  $C_4 = 3 * (b_r + b_s) + (js * r_r * r_s) / bfr_{res}$

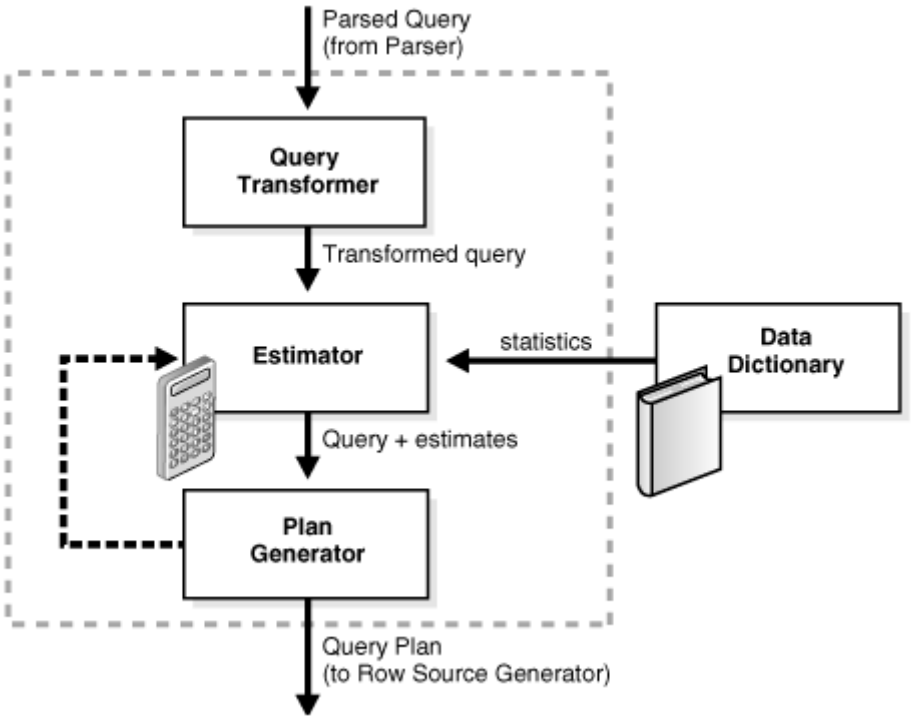
  Glavni izazov kod ovog algorima je kako napraviti hash funkciju, tako da je broj slogova u svakoj pariticiji približno jednak.

Zadatak ovog poglavlja je bio da pruži potrebno razumevanje optimizacije upita korišćenjem metoda zasnovanih na ceni, jer one predstavljaju centralni deo Oracle DBMS optimizatora. Svi algoritmi obrađeni u poglavljima 1.3.2 i 1.3.3 su relevantni i Oracle DBMS razmatra između ovih opcija (uz naravno još par algoritama koji ovde nisu obrađeni) prilikom generisanja najoptimalnijeg plana izvršenja. Upravo je Oracle DBMS optimizator tema narednog poglavlja.

2 – OPTIMIZACIJA UPITA KOD ORACLE DBMS-a

Optimizator upita Oracle DBMS-a je cost-based optimizator. Faktori koji utiču na vrednost cost funkcije su: cena I/O operacija (cena pristupa sekundarnom skladištu iz poglavlja 1.3), CPU vreme i troškovi komunikacije (npr. usled paralelizacije nekih algoritama). Za evaluaciju cost funkcije se koriste statistike o kojima se vodi evidencija u katalogu DBMS-a. Tačnost ovih statističkih parametara bitno utiče na performanse optimizatora i efikasnost generisanih optimalnih planova.

Optimizator na ulazu dobija parsiranu verziju SQL upita za koji je potrebno pronaći optimalnu strategiju izvršenja. Ovakav upit se u prvoj komponenti optimizatora – **transformeru upita (***eng.* ***Query transformer*****)** – inicijalno transformiše u ekvivalentni upit pogodniji za izvršenje. U sledećem koraku, na osnovu transformisanog upita, druga komponenta optimizatora – **generator plana (***eng.* ***Plan generator*****)** - generiše potencijalne planove izvršenja. Generator plana je u sprezi sa trećom komponentom – **komponenta za procenu cene (***eng.* ***Estimator*****)** koja za svaki generisani plan određuje cenu izvršenja. Često se dešava, naročito za složene upite, da je broj potencijalnih strategija za izvršenje jako veliki. Samim tim je često nemoguće izgenerisati sve moguće planove izvršenja. Zadatak estimatora je da u svakoj iteraciji procene eliminiše planove sa velikom vrednošću cost funkcije, dok se bolji planovi zadržavaju da na osnovu njih plan generator eventualno pronađe još bolje strategije za izvršenje. Dijagram ovog procesa je prikazan na slici 6.



Slika 6: Komponente optimizatora Oracle DBMS-a

U sledećem poglavlju je detaljnije objašnjen proces transformacije upita, kao prvi bitan korak u lancu optimizacije.

2.1 – Transformer upita

U ranijim verzijama Oracle-a je inicijalna transformacija upita bila zasnovana na pravilima (tj. heuristikama), od kojih su neke navedene u poglavlju 1.2, a zatim se tako transformisan upit optimizovao cost-based metodama. Danas je heuristički transformer gotovo u potpunosti zamenjen cost-based transformerom. U narednih nekoliko poglavlja objašnjene su najbitnije tehnike transformacije upita Oracle DBMS-a.

2.1.1 – OR ekspanzija

OR ekspanzija je tehnika kojom se disjunkcije u uslovima selekcije ili spoja razdvajaju na više manjih upita, sa po jednim uslovom, čiji se rezultati zatim spajaju UNION ALL operacijom. Ovakvo razbijanje disjunkcije na pojedinačne uslove često poboljšava performanse jer omogućava korišćenje pristupnih struktura, kao što su klastering ili sekundarni indeksi, a u slučajevima spoja može da eliminiše potrebu za operacijom Dekartovog proizvoda, koja je najskuplja za izvršenje. Naravno, nakon ekspanzije, estimator računa cenu izvršenja transformisanog upita, i ukoliko je ona manja od početne, ekspanzija se primenjuje. Primer ove transformacije dat je na slici 7 ispod. Upit koji se transformiše je:

Q: **SELECT** * **FROM** employees e, departments d
WHERE (e.email='SSTILES' OR d.department_name='Sales')
AND e.department_id = d.department_id;

```
3  ------------------------------------------------------------------------------------------------------
4  | Id  | Operation                     | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
5  ------------------------------------------------------------------------------------------------------
6  |   0 | SELECT STATEMENT              |                    |    11 |  2079 |     6   (0)| 00:00:01 |
7  |   1 |  VIEW                         | VW_ORE_19FF4E3E    |    11 |  2079 |     6   (0)| 00:00:01 |
8  |   2 |   UNION-ALL                   |                    |       |       |            |          |
9  |   3 |    NESTED LOOPS               |                    |     1 |    90 |     2   (0)| 00:00:01 |
10 |   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES          |     1 |    69 |     1   (0)| 00:00:01 |
11 |*  5 |      INDEX UNIQUE SCAN        | EMP_EMAIL_UK       |     1 |       |     0   (0)| 00:00:01 |
12 |   6 |     TABLE ACCESS BY INDEX ROWID| DEPARTMENTS        |     1 |    21 |     1   (0)| 00:00:01 |
13 |*  7 |      INDEX UNIQUE SCAN        | DEPT_ID_PK         |     1 |       |     0   (0)| 00:00:01 |
14 |   8 |    NESTED LOOPS               |                    |    10 |   900 |     4   (0)| 00:00:01 |
15 |   9 |     NESTED LOOPS              |                    |    10 |   900 |     4   (0)| 00:00:01 |
16 |* 10 |      TABLE ACCESS FULL        | DEPARTMENTS        |     1 |    21 |     3   (0)| 00:00:01 |
17 |* 11 |      INDEX RANGE SCAN         | EMP_DEPARTMENT_IX  |    10 |       |     0   (0)| 00:00:01 |
18 |* 12 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES          |    10 |   690 |     1   (0)| 00:00:01 |
19 ------------------------------------------------------------------------------------------------------
20
21 Predicate Information (identified by operation id):
22 ---------------------------------------------------
23
24    5 - access("E"."EMAIL"='SSTILES')
25    7 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
26   10 - filter("D"."DEPARTMENT_NAME"='Sales')
27   11 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
28   12 - filter(LNNVL("E"."EMAIL"='SSTILES'))
```

Slika 7: Primer OR ekspanzije primenjene na upit Q

Sa slike možemo da vidimo da je početni upit transformisan u dva manja upita koja su spojena UNION ALL operacijom. Prvi upit pravi spoj između tabela *DEPARTMENT* i *EMPLOYEE,* i filtrira rezultate po uslovu za atribut *email*, dok drugi upit vrši filtriranje po vrednosti atributa *department_name*. Zahvaljujući OR ekspanziji, za traženje radnika sa e-mailom „SSTILES“ se vrši pomoću indeksa. Bez ovakve transformacije bi filtriranje moralo da se izvrši tek nakon operacije spoja, koja bi samim tim imala mnogo više redova nego što je zapravo potrebno i direktno izvršenje početnog upita bi bilo veoma neefikasno.

2.1.2 – Spajanje pogleda (View merging)

Optimizacija SQL upita se u Oracle DBMS-u vrši na nivou SQL blokova. SQL blok može biti blok definisan SELECT naredbom na najvišem nivou, ugnježđeni upit ili pogled. To znači da će u slučaju više blokova optimizator izgenerisati podplanove za svaki zasebni blok – u slučaju ugnježđenih blokova podplanovi se najpre generišu za najugnježđenije blokove. U pojedinim situacijama, u kojima je to moguće, transformator može transformisati upit koji se sastoji iz više blokova u upit sa jednim SELECT-FROM-WHERE blokom. Razlikujemo dva ovakva slučaja: u prvom slučaju se unutrašnji upit nalazi u WHERE delu spoljašnjeg upita, tj. potrebno je evaluirati unutrašnji upit, a zatim se ti rezultati koriste za evaluaciju spoljašnjeg upita. Tehnika za transformisanje ovakvih upita se zove **odmotavanje (***eng.* ***Unnesting*****) –** ova tehnika je objašnjena u narednom poglavlju. Drugi slučaj, koji je tema ovog poglavlja, su upiti kod kojih se unutrašnji upit pojavljuje u FROM delu spoljašnjeg upita. Tehnika kojom se ovakvi upiti transformišu u jedan blok se naziva **spajanje pogleda** (*eng.* ***View merging***) – jer ugnježđeni upit upravo predstavlja pogled (koji može biti eksplicitno prethodno kreiran naredbom CREATE VIEW, ili implicitno definisane kao *inline* SQL blok) koji se spaja sa spoljašnjim blokom. Primer jednog ovakvog upita, sa inline pogledom je sledeći (inline pogled je označen plavom bojom):

```
Q: SELECT e.first_name, e.last_name, dept_locs_v.street_address,
        dept_locs_v.postal_code
   FROM  employees e,
       ( SELECT d.department_id, d.department_name, l.street_address, l.postal_code
         FROM   departments d, locations l
         WHERE  d.location_id = l.location_id ) dept_locs_v
   WHERE  dept_locs_v.department_id = e.department_id
   AND e.last_name = 'Smith';
```

Bez spajanja pogleda, ovaj upit bi se izvršio na sledeći način:
Prvo bi se izvšio unutrašnji blok – prvo se izvršava operacija projekcije, kojom se iz tabele DEPARTMENTS zadržavaju atributi *department_id, department_name* i *location_id*. Slično, iz table LOCATIONS, se projektuju atributi *street_address, postal_code* i *location_id. Z*atim se ovakve dve tabele spajaju po id-u lokacije, čime je konačno evaluiran pogled (logička tabela) *dept_locs_v*. Zatim se ova logička tabela spaja sa tabelom EMPLOYEES, koje je prethodno

filtrirana tako da sadrži samo radnike koji se prezivaju „Smith“. Jedan plan izvršenja prikazan je na slici 8.

```
Plan hash value: 2844867154

-----------------------------------------------------------------------------------------------------
| Id  | Operation                              | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |              |     1 |    50 |     4   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                          |              |     1 |    50 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES    |     1 |    18 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                    | EMP_NAME_IX  |     1 |       |     1   (0)| 00:00:01 |
|   4 |   VIEW PUSHED PREDICATE                |              |     1 |    32 |     2   (0)| 00:00:01 |
|   5 |    NESTED LOOPS                        |              |     1 |    38 |     2   (0)| 00:00:01 |
|   6 |     TABLE ACCESS BY INDEX ROWID        | DEPARTMENTS  |     1 |     7 |     1   (0)| 00:00:01 |
|*  7 |      INDEX UNIQUE SCAN                 | DEPT_ID_PK   |     1 |       |     0   (0)| 00:00:01 |
|   8 |     TABLE ACCESS BY INDEX ROWID        | LOCATIONS    |     1 |    31 |     1   (0)| 00:00:01 |
|*  9 |      INDEX UNIQUE SCAN                 | LOC_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("E"."LAST_NAME"='Smith')
   7 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
   9 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
```

Slika 8: Plan izvršenja upita Q, bez spajanja pogleda

Spajanjem pogleda se Q transformiše u upit koji predstavlja spoj 3 tabele, što se može videti i na osnovu plana izvršenja na slici 9.

```
Plan hash value: 257295346

-----------------------------------------------------------------------------------------------------
| Id  | Operation                              | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |              |     1 |    56 |     4   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                          |              |     1 |    56 |     4   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                         |              |     1 |    56 |     4   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                        |              |     1 |    25 |     3   (0)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES    |     1 |    18 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN                  | EMP_NAME_IX  |     1 |       |     1   (0)| 00:00:01 |
|   6 |     TABLE ACCESS BY INDEX ROWID        | DEPARTMENTS  |     1 |     7 |     1   (0)| 00:00:01 |
|*  7 |      INDEX UNIQUE SCAN                 | DEPT_ID_PK   |     1 |       |     0   (0)| 00:00:01 |
|*  8 |    INDEX UNIQUE SCAN                   | LOC_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
|   9 |   TABLE ACCESS BY INDEX ROWID          | LOCATIONS    |     1 |    31 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("E"."LAST_NAME"='Smith')
   7 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
   8 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
```

Slika 9: Plan izvršenja za upit Q sa primenom spajanja pogleda

Spajanjem pogleda smo dobili plan izvršenja koji odgovara upitu Q‘ koji ovako izgleda:

```
Q‘: SELECT e.first_name, e.last_name, l.street_address, l.postal_code
    FROM   employees e, departments d, locations l
    WHERE  d.location_id = l.location_id
    AND    d.department_id = e.department_id
    AND    e.last_name = 'Smith';
```

Bitno je primetiti da za upit Q imamo ukupno 4 mogućnosti za redosled spojeva: (EMPLOYEES, DEPT_LOCS_V), (DEPT_LOCS_V, EMPLOYEES)[3], gde u okviru pogleda DEPT_LOCS_V imamo 2 mogućnosti (LOCATION, DEPARTMENT) i (DEPARTMENT, LOCATION). Kod upita Q‘ imamo 6 mogućnosti (generalno, za spoj n tabela imamo n! različitih permutacija tabela koje učestvuju u spoju). Ovo je značajna prednost tehnike spajanja pogleda – nudi znatno veću fleksibilnost i raznovrsnost strategija za implementaciju više spojeva. Samim tim je moguće bolje iskoristiti prednost postojećih indeksa i generisati optimalniji plan. Sa druge strane, ovakav pristup dovodi do velikog rasta mogućih kombinacija sa povećanjem broja tabela, pa je potrebno obezbediti efikasan način za pretragu prostora stanja, kako bi se došlo do optimalnog rešenja.

U slučaju pogleda koji sadrže GROUP BY ili DISTINCT naredbe, nije uvek poželjno vršiti spajanje pogleda. Postoje argumenti i za i protiv primene ove tehnike u ovim situacijama. Sa jedne strane, spajanjem bi se operacije DISTINCT i GROUP BY izvršavale kao poslednje u stablu izvršenja, nakon što se izvrše operacije spojeva i filtriranja, sto može smanjiti cenu njihovog izvršenja. Sa druge strane, primena ovih operacija pre ostalih, može da smanji broj slogova koji učestvuju u operacijama spoja, što takođe može pozitivno uticati na cenu izvršenja. Kako najbolji plan u ovakvim situacijama bitno zavisi od trenutnih karakteristika podataka i tabela u sistemu, optimizator mora da proceni cene izvršenja za različite planove u oba slučaja.

2.1.3 – Odmotavanje upita (Query Unnesting)

Kod drugog tipa ugnježdenih upita se unutrašnji upit nalazi u FROM delu spoljašnjeg upita. Tehnika kojom se ovakvi upiti transformišu u složeniji upit sa jednim blokom se naziva **odmotavanje upita**. Ovo se postiže primenom različitih operacija spoja između tabela spoljašnjeg i unutrašnjeg bloka. Neki primeri ovakvih upita su dati u nastavku.

```
Q1: SELECT first_name, last_name, salary
    FROM EMPLOYEES e
    WHERE EXISTS (SELECT *
                  FROM DEPARTMENTS d
                  WHERE e.department_id = d.department_id AND d.location_id  = 1700);
```

3 Iako je JOIN komutativna operacija, iz ugla algoritma koji je implementira nije svejedno koja će se relacija naći u spoljašnjoj a koja u unutrašnjoj petlji, zbog prisustva različitih struktura pristupa.

Za ovakve upite, kod kojih se u ugnježdenom bloku koristi promenljiva iz spoljašnjeg bloke, se kaže da imaju korelisane blokove – za odmotavanje ovakvih upita se tada koristi i izraz **dekorelacija.** Ovaj upit se očigledno može transformisati tako što ćemo umesto unutrašnjeg bloka iskoristiti operaciju unutrašnjeg spoja između tabela EMPLOYEES i DEPARTMENTS. Primer plana izvršenja za ovako transformisan upit je dat na slici 10. Vidimo da je operacija spoja implementirana korišćenjem ranije pomenutog hash-join algoritma.

```
Plan hash value: 4017800164

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                  |   106 |  3074 |     5   (0)| 00:00:01 |
|*  1 |  HASH JOIN                          |                  |   106 |  3074 |     5   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| DEPARTMENTS      |    21 |   147 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | DEPT_LOCATION_IX |    21 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL                 | EMPLOYEES        |   107 |  2354 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
   3 - access("D"."LOCATION_ID"=1700)
```

Slika 10: Plan izvršenja transformisanog upita Q_1

Bez korišćenja tehnike odmotavanja bi za svaki slog iz tabele radnik bilo potrebno izvršiti unutrašnji blok i ukoliko on vrati rezultat, zadržali bismo taj slog. Za 10000 radnika bi bilo potrebno 10000 evaluacija unutrašnjeg upita. Upit sa jednim blokom je evidentno mnogo efikasniji za izvršenje.

Tehnika odmotavanja se takođe koristi za ugnježdene upite ispred kojih se nalazi neki od operatora IN, NOT IN, ANY, ALL. Za odmotavanja ovih upita se koriste specijalne operacije spojeva – **semi-join** (za IN i ANY) i **anti-join** (za NOT IN i ALL). Za demonstraciju operacije semi-join koristićemo sledeći upit:

Q_2: **SELECT** department_name
FROM DEPARTMENTS d
WHERE d.department_id **IN** (**SELECT** e.department_id
FROM EMPLOYEES e
WHERE e.salary > 10000);

Ovaj upit transformacijom postaje:

Q_2': SELECT department_name
FROM DEPARTMENTS d, EMPLOYEES e
WHERE d.department_id S=e.department_id **AND** e.salary > 10000;

Gde je plavom bojom oznančena operacija semi-join. Semi-join operacija funkcioniše tako što će kao rezultat generisati slog tabele DEPARTMENTS čim pronađe prvi odgovarajući slog za spoj, iz tabeli EMPLOYEES, nakon čega prelazi na sledeći slog tabele DEPARTMENTS. Operacija anti-join je slična ovoj operaciji, s tim što se slog prve tabele vraća kao rezultat samo ako se ne pronađe odgovarajući sloj u drugoj tabeli. Primer plana izvršenja za upit Q_2 je prikazan na slici 11.

```
Plan hash value: 2188966913

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |     9 |   207 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN SEMI              |             |     9 |   207 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |    27 |   432 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | DEPT_ID_PK  |    27 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT UNIQUE                 |             |    14 |    98 |     4  (25)| 00:00:01 |
|*  5 |    TABLE ACCESS FULL          | EMPLOYEES   |    14 |    98 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
       filter("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
   5 - filter("E"."SALARY">10000)
```

Slika 11: Primer odmotavanja upita Q_2 korišćenjem semi-join operacije

2.1.4 – Faktorizacija operacija (Join factorization)

Ova transformacija se koristi za optimizaciju izvršavanja SQL upita koji sadrži više grana povezanih UNION ALL operacijom. Tehnikom faktorizacije se prepoznaju izračunavanja koja su zajednička za sve grane ovakvih upita, a zatim se takva izračunavanja izvajaju kao top-level izračunavanja, van UNION ALL operacije. Ovakvim pristupom se sprečavaju višestruka izračunavanja za iste operacije, što obezbeđuje drastično poboljšanje performansi. Pogledajmo sledeći upit:

```
Q: SELECT e.first_name, e.salary, d.location_id
   FROM EMPLOYEES e, DEPARTMENTS d, LOCATIONS l
   WHERE e.department_id = d.department_id AND d.location_id = l.location_id
   AND e.salary > 5000
   AND l.postal_code IS NOT NULL
   UNION ALL
   SELECT e.first_name, e.salary, d.location_id
   FROM EMPLOYEES e, DEPARTMENTS d, EMPLOYEES m
   WHERE e.department_id = d.department_id AND d.manager_id = m.employee_id
```

```
AND e.salary > 5000
AND m.commission_pct IS NOT NULL;
```

U upitu Q su plavom bojom označene operacije koje su zajedničke za obe grane UNION ALL operacije. Faktorizacijom spoja između tabela EMPLOYEES i DEPARTMENTS i operacije selekcije radnika koji imaju platu veću od 5000, obezbeđeno je samo jedno izvršenje ovih operacija. U slučaju da je fajl EMPLOYEE jako veliki, ovakva transformacija bi donela značajno poboljšanje performansi. Plan izvršenja transformisanog upita može se videti na slici 12.

```
| Id  | Operation                     | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT              |                      |    80 |  3200 |    14   (8)| 00:00:01 |
|*  1 |  HASH JOIN                    |                      |    80 |  3200 |    14   (8)| 00:00:01 |
|   2 |   VIEW                        | VW_JF_SET$C68A813A   |    38 |   988 |    11  (10)| 00:00:01 |
|   3 |    UNION-ALL                  |                      |       |       |            |          |
|   4 |     MERGE JOIN                |                      |    27 |   459 |     5  (20)| 00:00:01 |
|*  5 |      TABLE ACCESS BY INDEX ROWID| LOCATIONS          |    22 |   220 |     2   (0)| 00:00:01 |
|   6 |       INDEX FULL SCAN         | LOC_ID_PK            |    23 |       |     1   (0)| 00:00:01 |
|*  7 |      SORT JOIN                |                      |    27 |   189 |     3  (34)| 00:00:01 |
|   8 |       VIEW                    | index$_join$_002     |    27 |   189 |     2   (0)| 00:00:01 |
|*  9 |        HASH JOIN              |                      |       |       |            |          |
|  10 |         INDEX FAST FULL SCAN  | DEPT_ID_PK           |    27 |   189 |     1   (0)| 00:00:01 |
|  11 |         INDEX FAST FULL SCAN  | DEPT_LOCATION_IX     |    27 |   189 |     1   (0)| 00:00:01 |
|* 12 |     HASH JOIN                 |                      |    11 |   176 |     6   (0)| 00:00:01 |
|* 13 |      TABLE ACCESS FULL        | DEPARTMENTS          |    11 |   110 |     3   (0)| 00:00:01 |
|* 14 |      TABLE ACCESS FULL        | EMPLOYEES            |    35 |   210 |     3   (0)| 00:00:01 |
|* 15 |   TABLE ACCESS FULL           | EMPLOYEES            |    57 |   798 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="ITEM_1")
   5 - filter("L"."POSTAL_CODE" IS NOT NULL)
   7 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
       filter("D"."LOCATION_ID"="L"."LOCATION_ID")
   9 - access(ROWID=ROWID)
  12 - access("D"."MANAGER_ID"="M"."EMPLOYEE_ID")
  13 - filter("D"."MANAGER_ID" IS NOT NULL)
  14 - filter("M"."COMMISSION_PCT" IS NOT NULL)
  15 - filter("E"."SALARY">5000)
```

Slika 12: Faktorizacija operacija upita Q

Na osnovu ovog plana možemo da primetimo sledeće: UNION ALL operacija u transformisanom upitu traži uniju slogova koji nastaju spojem tabela DEPARTMENTS i LOCATIONS iz prve grane, i DEPARTMENTS i EMPLOYEES iz druge grane (naravno slogovi se i filtriraju shodno uslovima u WHERE naredbi koji su specifični za grane). Tek nakon nalaženja unije se ovi slogovi spajaju sa tabelom EMPLOYEES, koja je prethodno filtrirana uslovom e.salary > 5000 (obe ove operacije se sada izračunavaju samo jednom, umesto dva puta). Rezultati operacije unije su u planu izvršenja predstavljeni pogledom VW_JF, koji je implicitno kreiran od strane optimizatora.

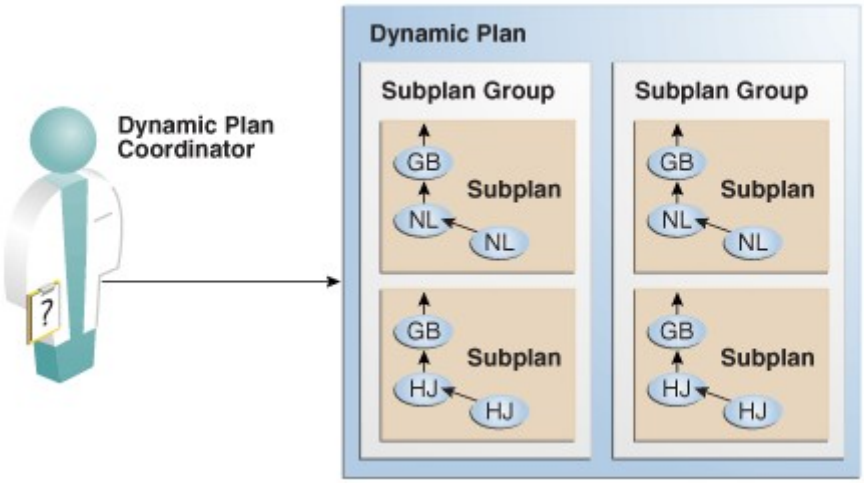
2.1.5 – Ostale transformacije

U prethodnih nekoliko poglavlja navedene su neke od transformacija koje Oracle DBMS koristi za potrebe optimizacije upita. Pored navedenih, Oracle optimizator koristi i brojne druge tehnike kao što su **Star transformacija, umetanje predikata, korišćenje materijalizovanih pogleda**, i mnoge druge. Star transformacija i korišćenje materijalizovanih pogleda imaju veliki značaj u Data Warehouse sistemima. Generalno, veliki deo ovih transformacija implementiraju i drugi DBMS-ovi – prednost Oracle-a je veća raznovrsnost algoritama za implementaciju različitih operacija, koji mogu efikasno da koriste različite strukture za pristup podacima.

2.2 – Adaptivna optimizacija upita

Bitna prednost Oracle DBMS optimizatora je mogućnost adaptivne optimizacije upita. Ovaj mehanizam omogućava optimizatoru da prilagođava planove u toku samog izvršenja upita, na osnovu dostupnih informacija o upitu i adaptivnih statistika o kojima se vodi evidencija.

Podrazumevano, svaki generisani plan izvršenja je adaptivni. Adaptivni planovi predstavljaju skup većeg broja definisanih podplanova. Najoptimalniji plan se generiše dinamički, u toku izvršenja upita i može da se menja u zavisnosti od stanja sistema u datom trenutku. Plan koji optimizator generiše je **dinamički plan** - predstavlja skup grupa podplanova (gde svaka grupa sadrži više mogućih podplanova za jednu operaciju). Posebna komponenta – kolektor statistika – zadužena je da „snabdeva“ plan informacijama od interesa, na osnovu kojih koordinator dinamičkih planova određuje koja grupa podplanova će se koristiti prilikom izvršenja upita. Izgled dinamičkog plana prikazan je na slici 13.



Slika 13: Dinamički plan i koordinator planova

Na slici 14 prikazano je kako izgleda adaptivni plan za izvršenje SQL upita. Crticama su označene operacije koje se ne izvršavaju, već predstavljaju alternativne strategije.

```
14 ------------------------------------------------------------------------------------------------------
15 |   Id  | Operation                          | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
16 ------------------------------------------------------------------------------------------------------
17 |     0 | SELECT STATEMENT                   |              |       |       |     8 (100)|          |
18 |     1 |  SORT UNIQUE                       |              |     5 |   112 |     7  (15)| 00:00:01 |
19 |     2 |   UNION-ALL                        |              |       |       |            |          |
20 |  *  3 |    CONNECT BY WITH FILTERING (UNIQUE)|            |       |       |            |          |
21 |- *  4 |     HASH JOIN                      |              |     1 |    16 |     2   (0)| 00:00:01 |
22 |     5 |      NESTED LOOPS                  |              |     1 |    16 |     2   (0)| 00:00:01 |
23 |-    6 |       STATISTICS COLLECTOR         |              |       |       |            |          |
24 |  *  7 |        INDEX RANGE SCAN            | I_CODEAUTH1  |     1 |     8 |     1   (0)| 00:00:01 |
25 |  *  8 |       INDEX RANGE SCAN             | I_SYSAUTH1   |     1 |     8 |     1   (0)| 00:00:01 |
26 |- *  9 |      INDEX FAST FULL SCAN          | I_SYSAUTH1   |     1 |     8 |     1   (0)| 00:00:01 |
27 |- * 10 |     HASH JOIN                      |              |     3 |    63 |     3   (0)| 00:00:01 |
28 |    11 |      NESTED LOOPS                  |              |     3 |    63 |     3   (0)| 00:00:01 |
29 |-   12 |       STATISTICS COLLECTOR         |              |       |       |            |          |
30 |    13 |        CONNECT BY PUMP             |              |       |       |            |          |
31 |  * 14 |       INDEX RANGE SCAN             | I_SYSAUTH1   |     3 |    24 |     1   (0)| 00:00:01 |
32 |- * 15 |      INDEX FAST FULL SCAN          | I_SYSAUTH1   |     3 |    24 |     1   (0)| 00:00:01 |
33 |  * 16 |    INDEX RANGE SCAN                | I_CODEAUTH1  |     1 |     8 |     1   (0)| 00:00:01 |
34 ------------------------------------------------------------------------------------------------------
35
```

Slika 14: Adaptivni plan upita

Na ovoj slici možemo da vidimo da optimizator razmatra dva potencijalna plana za izvršenje operacije spoja (redovi 21, 22 i 27, 28). Kolektor statistika snabdeva optimizator informacijama o kardinalnosti tabela koje učestvuju u spoju (redovi 23 i 29). Inicijalno, početnim planom izvršenja predviđeno je korišćenje ugnježdenih petlji uz korišćenje indeksa za pristup unutrašnjoj tabeli. Ova strategija je opravdana pod uslovom da kardinalnost tabele koja je deo spoljašnje petlje nije velika. Međutim, ukoliko ove vrednost premaši neku unapred definisanu vrednost (*threshhold*), kolektor statistika će o tome obavestiti koordinator planova, koji će plan izvršenja operacije spoja promeniti u hash-join (koji je optimalniji kada postoji veći broj vrsta).

Tehnika adaptivne optimizacije pruža značajno poboljšanje performansi, zbog toga što inicijalno generisani plan često nije dovoljno optimalan. Razlog tome je loša inicijalna procena parametara koji utiču na cenu operacija. Mogućnost prilagođavanja planova u toku izvršenja, na osnovu preciznih i tačnih informacija drastično utiče na efikasnost finalnog plana – naročito u sistemima koji su podložni čestim promenama tabela. Osim planova za izvršavanje upita, adaptivna optimizacija pruža mogućnost dinamičkog određivanja raspodele podataka po serverima u slučaju parelelnih izvršenja upita.

2.2 – Aproksimacija rezultata upita

Često je za potrebe analize podataka potrebno primenjivati agregacione funkcije, kao što su AVG, COUNT, MEDIAN, nad ogromnom količinom podataka. Takođe, potrebno je dobiti ove rezultate u nekom doglednom vremenu. Sa druge strane, kako se ovakvi upiti često koriste da bi se uvideli određeni trendovi i obrasci u podacima, korisnicima nije presudno da uvek dobiju 100% tačne rezultate. Za ovakve potrebe, Oracle nudi set funkcija kojima se vrednosti rezultata ovakvih

upita aproksimiraju, sa tačnošću od oko 97% i zahvaljujući činjenici da se ne računaju apsolutno tačne vrednosti, ovi upiti se izvršavaju jako brzo. Primeri ovakvih funkcija su APPROX_SUM, APPROX_DISTINCT, APPROX_COUNT, APPROX_MEDIAN, itd.

2.3 – Upravljanje SQL planovima (SQL Plan Management)

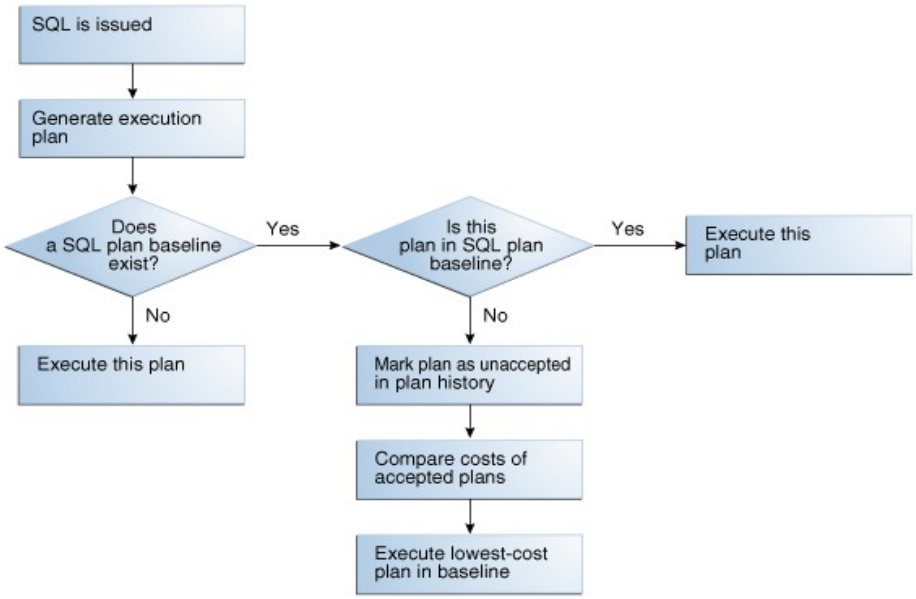
Svaka baza podataka vremenom prolazi kroz niz promena. Ažuriranje softvera optimizatora, ili operativnog sistema, kreiranje novih indeksa i drugih struktura za pristup fajlovima, promena hardverskih komponenti sistema, su samo neki od razloga koji mogu da dovedu do promene generisanih planova izvršenja za SQL upite. Kako su performanse izvršenja upita od ključne važnosti za svaki softverski sistem, potrebno je srprečiti retke, ali povremene situacije u kojima može da dođe do narušavanja performansi izvršenja nekog upita, usled promene ranije generisanih planova. Mehanizam Oracle DBMS-a koji ovo sprečava je SQL Plan Management.

Ovaj mehanizam vodi evidenciju o skupu planova za različite SQL upite – koji se zove **SQL Plan Baseline**, kojim je za svaki upis definisana donja granica performansi izvršenja. Svaki novogenerisani plan koji ima lošije performanse od onih u baseline-u će biti odbačen. Glavne komponente SQL Plan menadžera su:

- **Plan Capture –** Komponenta zadužena za snimanje svih podataka o planu izvršenja SQL upita.
- **Plan Selection –** Komponenta koja je zadužena za selekciju plana za izvršenje, tako da ne dođe do gubitka performansi. Ova komponenta vrši selekciju odgovarajućeg plana po sledećem algoritmu:
  1. Nakon parsiranja SQL upita, optimizator generiše plan izvršenja sa najmanjom cenom. Ukoliko ne postoji plan za upit u SQL Plan baseline-u, ovaj plan se i izvršava.
  2. Ukoliko ovaj plan izvršenja već postoji u SQL Plan baseline-u, plan se izvršava.
  3. Ukoliko postoji drugačiji plan (ili više njih) za isti upit u SQL Plan baseline-u, novi plan se dodaje i označava kao „neprihvaćen“. Postojeći, prihvaćeni planovi se upoređuju i za konačni plan izvršenja se bira onaj sa najmanjom cenom.

  Ilustracija ovog algoritma je prikazana na slici 15.

- **Plan Evolution –** Zadatak ove komponente je da proceni planove koji su označeni kao „neprihvaćeni“ od strane komponente za selekciju planova. Za svaki od ovih planova Plan Evolution komponenta vrši prvo **verifikaciju –** proverava da li su performanse plana bolje ili iste postojećemim planovima za isti upit u SQL Plan baseline-u. Ukoliko je to slučaj, dodaje ih u SQL Plan baseline kao „prihvaćene“, u suprotnom ih odbacuje.



Slika 15: Plan Selection algoritam

2.4 – Nagoveštaji (Hints)

Nagoveštaji[4] su specijalni anotacije u SQL naredbe, kojima se izdaju direktive optimizatoru. Pogodni su kada je potrebno „premostiti“ podrazumevano ponašanje optimizatora, npr. u slučajevima kada se za neki upit generišu neoptimalni planovi. Ukoliko razumemo šta je razlog tome, pomoću nagoveštaja je možemo sugerisati optimizatoru koje optimizacije da primeni (odnosno ne primeni). Nagoveštajima je moguće naterati optimizator da iskoristi određenu strukturu za pristup za neku tabelu, fiksirati redosled spojeva kod JOIN operacije, forsirati određeni algoritam za implementaciju JOIN operacije, sprečiti ili primeniti neku transformaciju, itd. Na slici 16 je dat primer plana izvršenja za upit Q iz poglavlja 2.1.2, sa direktivom da se izbegne spajanje pogleda.

```
Plan hash value: 2844867154

-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |             |     1 |    50 |     4   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                       |             |     1 |    50 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES   |     1 |    18 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | EMP_NAME_IX |     1 |       |     1   (0)| 00:00:01 |
|   4 |   VIEW PUSHED PREDICATE             |             |     1 |    32 |     2   (0)| 00:00:01 |
|   5 |    NESTED LOOPS                     |             |     1 |    38 |     2   (0)| 00:00:01 |
|   6 |     TABLE ACCESS BY INDEX ROWID     | DEPARTMENTS |     1 |     7 |     1   (0)| 00:00:01 |
|*  7 |      INDEX UNIQUE SCAN              | DEPT_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|   8 |     TABLE ACCESS BY INDEX ROWID     | LOCATIONS   |     1 |    31 |     1   (0)| 00:00:01 |
|*  9 |      INDEX UNIQUE SCAN              | LOC_ID_PK   |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Slika 16: NO_MERGE hint

4 Nagoveštaji se navode između specijalnih zagrada /* */ odmah posle SELECT naredbe, npr. za sprečavanje transformacije spajanja pogleda komanda je SELECT /*+NO_MERGE (view_name)*/ ...

ZAKLJUČAK

U ovom seminarskom radu istražili smo neke od ključnih koncepata vezanih za optimizaciju upita. Objašnjeni su mnogi bitniji teorijski koncepti iz ovog domena i dat je pregled različitih pristupa za rešavanje problema određivanja optimalnih strategija izvršenja. Navedeni su neki primeri metoda zasnovanih na heurističkim pravilima, koje su ranije bile zastupljenije, kao i primeri danas aktuelnijih metoda zasnovanih na ceni.

Ove teorijske osnove su dale potreban uvod za razumevanje konkretnih rešenja koje pruža optimizator Oracle sistema za upravljanje bazama podataka – koji je bio tema drugog dela rada. Kroz pokazne primere su ilustrovane bitnije tehnike koje ovaj optimizator koristi kako bi složene upite transformisao u oblik koji je pogodniji za izvršenje.

Danas, sa rastom podataka u softverskim sistemima, potrebno je stalno nadgledanje i prilagođavanje strategija optimizacije. Ovo naglašava značaj kontinuiranog praćenja performansi sistema i prilagođavanja optimizacionih strategija i potvrđuje značaj ove komponente svakog DBMS-a.

LITERATURA

[1] Fundamentals of Database Systems - Navathe, Elmasry

[2] Oracle Database SQL Tuning guide - https://docs.oracle.com/en/database/oracle/oracle-database/23/tgsql/

[3] Oracle Database Administration Workshop kurs