Frågebearbetning

Thomas Padron-Mc Carthy Tore Risch

23 juli 2005

Kapitel 1

Frågebearbetning

Databasfrågor i SQL och andra frågespråk uttrycks *icke-procedurellt* (även kallat *deklarativt*). Det innebär att SQL-användaren uttrycker *vad* som skall göras, och databashanteraren bestämmer sedan *hur* en given fråga skall utföras. "Vad" innebär i detta fall en specifikation av vilka tabeller som ingår i en fråga och vilka datavärden som skall matchas för att forma svaret på frågan. "Hur" innebär att databashanteringssystemet för en given fråga genererar ett (snabbt) sökprogram som traverserar de interna datastrukturer som representerar tabellerna i databasen, för att kombinera och hitta eftersökta datavärden. Man kan säga att detta är en form av *automatisk programmering* eftersom systemet automatiskt genererar sökprogram för en given icke-procedurell sökspecifikation i form av en fråga. De genererade sökprogrammen kallas *exekveringsplaner*.[1]

För en given icke-procedurell databasfråga finns det ofta ett stort antal korrekta exekveringsplaner. Olika exekveringsplaner kan ta mycket olika tid att köra. Frågeoptimering går ut på att databashanteraren automatiskt genererar den effektivaste exekveringsplanen (eller i alla fall en tillräckligt effektiv) för en given fråga.

I det här kapitlet går vi igenom hur databassystemet översätter en SQL-fråga till en sådan exekveringsplan så att den körs så snabbt som möjligt.

1.1 Varför ska jag lära mig det här?

Frågeoptimering är av central betydelse för databassystemets uppbyggnad. Vill man förstå hur en databashanterare fungerar måste man också förstå hur frågeoptimering fungerar.

Även den som (hur obegripligt det än kan låta!) *inte* är intresserad av att förstå databashanterarens inre arbete, har praktisk nytta av att att känna till hur frågeoptimering går till. Det kan nämligen vara nödvändigt att förstå lite om hur frågeoptimeringen fungerar för att kunna specificera maximalt effektiva SQL-frågor. Två olika frågor som egentligen är ekvivalenta (eller ännu hellre

[1] Kallas *evaluation plan* eller *execution plan* på engelska.

nästan ekvivalenta) kan vara olika svåra för databashanteraren att optimera, och kan därför ge olika exekveringsplaner. Skillnaden i exekveringstid mellan olika exekveringsplaner för en och samma SQL-fråga kan vara enorm, och en dåligt optimerad fråga kan lätt ta 1000 gånger längre tid att utföra än en optimal exkveringsplan.

I moderna databashanterare kan man för en given fråga be att få ut exekveringsplanen för att se om den ser bra ut. Om så inte är fallet kan man antingen formulera om frågan eller ge tips (pragma) till frågeoptimeraren för att frågan skall bli snabbare att utföra.

1.2 Nyttan av frågeoptimering

Olika möjliga exekveringsplaner har olika komplexitet[2] med avseende på databasens storlek. Till exempel visar vi senare att en naiv exekveringsplan kan ha komplexitet $O(N^2)$ där N är antal rader i de tabeller som berörs av frågan, medan den optimala planen kanske har komplexitet $O(logN)$. Eftersom N ofta är mycket stort i databassammanhang ger frågeoptimering oerhört stora effektivitetsvinster redan vid måttligt stor databas (till exempel redan med N=10000). Har man att göra med mycket små datamängder så har frågeoptimering inte så stor betydelse och man kan använda en förutbestämd ooptimerad statisk traverseringsording genom hela databasen. Exempelvis bygger programmeringsspråket Prolog på en inbyggd sådan förutbestämd datatraverseringsordning.[3] Eftersom redan en databas med N=10000 anses vara mycket liten duger statiska strategier inte för databaser. Frågorna måste alltså optimeras.

Varför inte procedurella sökprogram? I procedurella programmeringsspråk, till exempel Java eller C, får man explicit definiera sina sökalgoritmer. En exekveringsplan kan ses som ett automatiskt genererat sådant procedurellt sökprogram i ett speciellt programmeringsspråk för databasaccess. Innan relationsdatabaserna slog igenom var procedurella sökprogram den metod som användes för databasutsökning. Man kan hävda att automatisk frågeoptimering aldrig kan ge bättre prestanda än ett optimalt manuellt programmerat procedurellt sökprogram. Faktum är att sådana argument restes då relationsdatabaserna började utvecklas. Till exempel var det samma företag, IBM, som tidigare utvecklat det ledande "procedurella" databassystemet, IMS, som började utveckla de första relationsdatabaserna i slutet av 1970-talet. IMS-utvecklarna kunde med fog hävda att man alltid kan göra ett manuellt procedurellt IMS-program som är lika snabbt som den optimala exekveringsplanen för motsvarande relationsdatabas. Relationsdatabasutvecklarna vid IBM stod således inför en rejäl utmaning att automatiskt generera lika bra exekveringsplaner som optimal IMS-kod. Utmaningen ledde till utveckling av så kallad *kostnadsbaserad frågeoptimering*, vilken är den dominerande metoden att optimera databasfrågor.

Kostnadsbaserad frågeoptimering: Kostnadsbaserad frågeoptimering bygger på att optimeraren har en inbyggd metod för att uppskatta kostnaden för

[2]Komplexitet förklaras kort i avsnitt **??**.

[3]Prolog traverserar en trädstrukturerad databas djupet först.

en given exekveringsplan, en så kallad *kostnadsmodell*. Med kostnad menar man normalt den förväntade *tiden* för att köra frågan. Kostnadsmodellen är en matematisk modell för att uppskatta kostnaden att utföra en exekveringsplan, baserat på statistiska data om databasens innehåll samt kunskap om hur olika kommandon i en exekveringsplan uppför sig. Statistiska data kan till exempel vara att databassystemet vet hur många rader det finns i varje tabell eller hur datavärden är statistiskt fördelade i en kolumn. Vad kostnadsbaserad frågeoptimering går ut på är att från rymden av alla möjliga exekveringsplaner för en given databasfråga välja den billigaste med avseende på kostnadsmodellen. Problemet är här att antal möjliga exekveringsplaner kan vara mycket stort och exponentiellt beroende av storleken på SQL-frågan, till exempel hur många villkor den har. Om frågans storlek är Q så är komplexiteten hos kostnadsbaserad frågeoptimering i värsta fall $O(Q!)$, alltså exponentiellt beroende av frågans storlek. Kostnadsbaserad frågeoptimering lönar sig i alla fall om databasen är stor eftersom en optimerad fråga kanske tar $O(log(N))$ att utföra, där N är databasens storlek, medan en ooptimerad fråga tar $O(N^2)$. Det lönar sig därför att $N >> Q$.

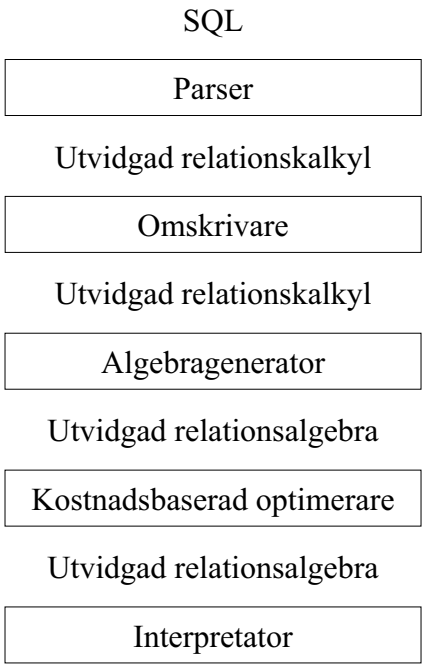
Heuristisk frågeoptimering: Som alternativ till kostnadsbaserad frågeoptimering kan man tänka sig att använda *heuristisk* frågeoptimering där man har ett antal tumregler för hur en exekveringsplan skall genereras. Heuristiska metoder har i allmänhet betydligt lägre komplexitet är kostnadsbaserad optimering (typiskt $O(Q^2)$). Emellertid skall man ha klart för sig att en dålig exekveringsplan kan vara 1000-tals gånger långsammare än en optimal plan och dålig optimering (till exempel med heuristiska metoder) kan leda till oacceptabla prestanda. För att vara konkurrenskraftig måste således databasföretagen tillhandahålla mycket bra frågeoptimerare; det har funnits många exempel där en given databasfråga har blivit oacceptabelt långsam efter byte till databashanterare med sämre optimerare. Detta har lett till att databasföretagen utvecklat oerhört avancerade kostnadsbaserade optimerare för att vara konkurrenskraftiga. Dessa optimerare innehåller också en del heuristiska metoder för att snabba upp själva optimeringen, men dessa heuristiska metoder är noggrant analyserade så att de inte genererar oacceptabelt dyra exekveringsplaner.

Manuell frågeoptimering: Man kan fråga sig om det inte skulle duga att låta användaren explicit påverka optimeringen, till exempel genom att manuellt ordna om villkorsuttrycken i frågan. Sådan halvmanuell frågeoptimering tillämpades i tidiga relationsdatabassystem. Programmeringsspråket Prolog tillämpar en liknande metod där programmeraren explicit kan påverka effektiviteten genom att ordna om villkor. Det allvarligaste problemet med manuella metoder är att det blir mycket svårt att manuellt optimera en fråga som anropar vyer vilka i sin tur är manuellt optimerade. Vidare beror exekveringseffektiviteten till stor del på vilka interna datastrukturer och algoritmer som används vid frågeexekveringen. En modern databashanterare kan använda många olika sådana exekveringsalgoritmer så uppgiften för användaren att manuellt påverka exekveringsplanen kan bli överväldigande. Ytterligare ett problem är att exekveringsstrategin också kanske måste ändras om databasens innehåll ändras mycket. Efter stora uppdateringar som påverkar exekveringsstrategierna måste

man således gå in och ändra alla manuella ”hack” i alla påverkade frågor och vyer.

1.3 Frågebearbetningsfaser

Figur 1.1 illustrerar de olika faserna av frågebearbetningen i en modern frågeoptimerare.



Figur 1.1: Frågebearbetningsfaser

Först överför en *parser* SQL-frågan till en intern representation, i likhet med vad som sker för de flesta programmeringsspråk. Därvid sker kontroll att frågan är syntaktisk korrekt och inte innehåller typfel. Parse-trädet är väsentligen en intern representation av SQL-frågan i utvidgad relationskalkyl.[4] SQL är mer kraftfullt än klassisk relationskalkyl, till exempel genom att arbeta med påsar (mängder med duplikat), null-värden, aggregeringsoperatorer, sortering, mm.

[4]Relationskalkyl beskrivs kort i avsnitt **??**.

Därför kan inte klassisk relationskalkyl användas för att internt representera SQL, utan en utvidgad relationskalkyl måste användas. Den utvidgade relationskalkylen är fortfarande deklarativ så det parsade uttrycket innehåller inte någon information om hur det skall exekveras. Eftersom den utvidgade relationskalkylen i princip är ekvivalent med motsvarande SQL-frågor använder vi nedan SQL-notation för att illustrera frågeomskrivningar.

Relationskalkylen förenklas och transformeras sedan av en *frågeomskrivare*. Här görs transformationer av frågan som inte påverkar dess resultat men som är garanterade att förbättra dess prestanda. Exempel på viktig sådan transformation är *vyexpansion* där referenser till vyer ersätts med vydefinitionerna. Moderna optimerare gör också en hel del mer eller mindre avancerade andra frågeomskrivningar, till exempel för att ta bort ekvivalenta deluttryck i frågor.

En *algebragenerator* transformerar därefter relationskalkyluttrycket till ett relationsalgebrauttryck. För ett givet relationskalkyluttryck finns det en systematisk översättning (se nedan) till relationsalgebra. Relationsalgebran är ett funktionellt språk där de inbyggda funktionerna tar tabeller som argument och returnerar en tabell som resultat.[5] Vidare är relationsalgebran *procedurell* i den meningen att för ett givet relationsalgebrauttryck finns det en väl definierad ordning i vilket det skall exekveras. Liksom för relationskalkylen är den klassiska mängdorienterade relationsalgebran inte tillräcklig vare sig för att representera alla SQL-frågor eller för att duga som bas för frågeoptimering. Därför behövs en utökad relationsalgebra som innehåller fler operatorer än den traditionella, till exempel sortering, hantering av påsar (mängder med duplikat), null-värden, aggregeringsoperatorer, och eliminering av duplikat. Vi kallar detta *utökad relationsalgebra*. För att den utökade relationsalgebran skall kunna användas för att representera exekveringsplaner behövs vidare operatorer (funktioner) som tar hänsyn till olika fysiska lagringsstrukturer och index som används av databassystemet. Ibland används termen *fysisk relationsalgebra*, till skillnad från den vanliga *logiska* relationsalgebran.

Den systematiska översättningen till utökad relationsalgebra skulle kunna tolkas (interpreteras) direkt och ge korrekt svar på databasfrågan. Emellertid är den systematiska planen så gott som alltid icke-optimal. Därför utförs *kostnadsbaserad frågeoptimering* i samband med översättningen. Där appliceras ett antal heuristiska och kostnadsbaserade transformationer (omskrivningar) på det utökade relationsalgebrauttrycket, för att generera ett optimalt ekvivalent uttryck. Frågeoptimeringen väljer vidare vilka algoritmer, till exempel för tabellhopslagning (join) som skall användas i den slutliga exekveringsplanen. Den utökade relationsalgebran innehåller således många olika join-operatorer. Den kostnadsbaserade frågeoptimeringen kan mycket radikalt förbättra frågeprestanda och är kritisk för effektiv frågeutförande. Principerna för kostnadsbaserad frågeoptimering kommer att förklaras närmare nedan.

I den sista fasen tolkas (interpreteras) det optimala utökade relationsalgebrauttrycket för att producera frågeresultatet. En viktig detalj här är att mellanresultat av algebraoperatorer kan vara mycket stora tabeller som inte får

[5] Man kallar ett språk som tar argument och ger resultat av samma typ för ett *slutet* språk.

plats i primärminnet. Därför är tolkningen *strömmad*,[6] dvs raderna i resultaten från algebraoperatorerna produceras en åt gången snarare än att temporära tabeller byggs upp (eller *materialiseras*, som det kallas).

1.4 Exempel

Som ett mycket enkelt exempel att illustrera betydelsen av frågeoptimering, antag att vi har en relationsdatabas med två tabeller med personnummer **pnr** som nyckel:

```
Persondata(pnr,namn)
Anställning(pnr,avdelning,lön)
```

En enkel fråga över dessa tabeller är:

```
select lön
from Persondata p, Anställning a
where p.pnr = a.pnr
      and namn = "Kalle Persson"
```

Vi antar att det finns 100000 rader i båda tabellerna. Vi antar vidare att det bara får plats 10 rader per diskblock[7] och att tabellens rader ligger lagrade sekventiellt på disk ordnat efter personnummer.

För snabbast möjliga sökningar finns det index (B-träd) för samtliga kolumner i båda tabellerna. I vårt exempel antas varje nod rymma 100 nyckel/pekarpar.[8]

1.5 Vyexpansion

För att illustrera principen för vyexpansion definierar vi en vy över våra exempeltabeller:

```
create view Löner
 as select namn, avdelning, lön
    from Persondata p, Anställning a
    where p.pnr = a.pnr
```

En typisk fråga över ovanstående vy är:

```
select lön from Löner where namn = 'Kalle Persson'
```

[6] Termen *pipelined* används också.

[7] I verkligheten beror antalet rader som får plats per diskblock på längden av raderna. Vidare kan det finnas oanvänt utrymme i varje block. För att förenkla beskrivningarna antar vi dock här alltid att antalet rader per block är konstant.

[8] I praktiken beror denna siffra på nyckel- och pekarlängden.

Som tidigare nämnts är vyexpansion en viktig form av frågeomskrivning. Vyexpansion ersätter vyreferenser med dess definitioner. I exemplet ovan substituerar systemet in definitionen av vyn **Löner** i frågan varvid man får följande omskrivna fråga:

```
select a.lön
from Persondata p, Anställning a
where p.pnr = a.pnr and
      p.namn = 'Kalle Persson'
```

Vad som skett här är att definitionen av vyn **Löner** har substituerats in och villkoret `namn = 'Kalle Persson'` på vykolumnen **namn** har ersatts med motsvarande kolumnvillkor i den tabell som lagrar **namn** i vydefinitionen.

Vyexpansion gör det möjligt för den efterföljande kostnadsbaserade frågeoptimeraren att upptäcka alla index som kan påverka utformningen av den optimala exekveringsplanen. I detta fall kan optimeraren se att det finns index för kolumnerna **pnr** och **namn** i de lagrade tabellerna.

Vyer är ofta definierade i termer av andra vyer. I sådana fall expanderas alla vyer rekursivt ända tills enbart lagrade tabeller refereras i den expanderade frågan. Till exempel skulle vi kunna ha en annan vy:

```
create view Administratörslöner
 as select namn, lön
    from   Löner
    where  avdelning = 'adm'
```

Antag frågan:

```
select lön from Administratörslöner
where namn = 'Kalle Persson'
```

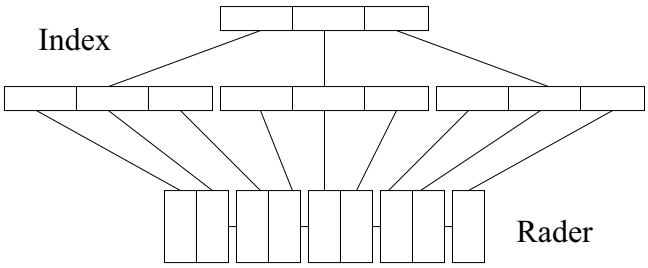
I detta fall sker vyexpansion rekursivt i två steg så att den slutliga expanderade frågan blir:

```
select a.lön
from Persondata p, Anställning a
where p.pnr = a.pnr and
      a.avdelning = 'adm' and
      p.namn = 'Kalle Persson'
```

1.6 Klustrade index

När det gäller effektiv användning av index skiljer man mellan *klustrade* och *oklustrade* index, där klustrade index, till skillnad mot oklustrade index, har (i stort sett) samma ordning som raderna i tabellen.[9]

[9] Klustrade index innefattar här alltså både det som vi i avsnitt **??** kallade *grupperade* index (engelska: clustering indexes) och *primärindex*.



Figur 1.2: Klustrat index

Figur 1.2 illustrerar ett klustrat index där varje diskblock antas ha plats för tre indexnycklar och två tabellrader.[10] I figuren ser vi fem datablock med rader, som vi kallar *radblock*, som indexeras av ett index med *indexblock* bestående av ett rotblock och tre lövblock. Radblocken är hoplänkade för snabb sekventiell genomsökning av tabellen. Antag att antal indexnycklar per block är I och att det finns $card(T)$ rader i tabellen.[11] Då har lägsta nivån i indexet $card(T)/I$ block och antal nivåer i indexet, indexets *djup* D, blir $log_I(card(T))$. I figur 1.3 är $card(T) = 9$ och $I = 3$ och således är djupet $log_3(9) = 2$. Antal block i indexet, B blir $B = 1+I^1+I^2+...+I^{D-1} = (1-I^D)/(1-I)$. Eftersom $card(T) = I^D$ får vi följande formel för antal noder i indexet: $B = (1 - card(T))/(1 - I)$. Indexet i figur 1.2 är förenklat i den meningen att antal tabellposter (9) är en exponent av förgreningsfaktorn i indexet (3). I praktiken är inte situationen så ideal och våra formler blir då approximativa.

För klustrade index gäller att det att om man m.h.a. indexet söker reda på ett intervall av X indexnyckelvärden och det får plats med B tabellrader i ett diskblock så behövs det X/B läsningar av radblock för att hämta motsvarade tabellrader. Om man till exempel i figur 1.2 traverserar hela tabellen sekventiellt genom indexet måste man läsa 4 indexblock och 5 radblock.[12]

Indexet i vårt exempel är något förenklat i den meningen att det antas att det är *unikt*, vilket innebär att för varje indexnyckel finns exakt *en* motsvarande rad.[13] I verkligheten finns ofta mer än en matchande rad, och då måste man också hantera detta. Till exempel om man har ett index över kolumn med personnamn så kan flera personer ha samma namn. För att förenkla vår diskussion antar vi dock i fortsättningen att alla index är unika.

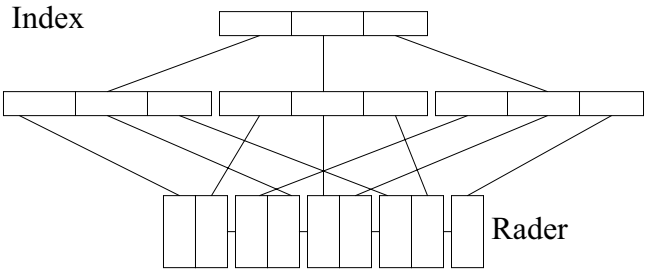
Figur 1.3 illustrerar motsvarande oklustrade index. Accesstiden för oklustrade index är långsammare än för klustrade index då oklustrade index slumpvis ref-

[10] Blockstorlek tre i bilden är vald för att förenkla illustrationen. I vårt räkneexempel nedan antar vi i stället den mer realistiska blockstorleken 100 för antal indexnycklar per block, samt 10 rader per block.

[11] Antalet rader i en tabell kallas tabellens *kardinalitet*, *cardinality* på engelska.

[12] Eftersom radblocken är länkade klarar man sig emellertid i detta fall med att bara läsa de 5 radblocken.

[13] Det är alltså ett *primärindex*, med terminologin i kapitel **??**.



Figur 1.3: Oklustrat index

ererar till diskblock i tabellen. Antal indexnoder är detsamma som för klustrade index, men genomsökning av indexet i indexnyckelordning resulterar i att ett nytt block måste läsas för varje indexnyckel. Således om man m.h.a. indexet söker reda på X indexnyckelvärden så behövs det också läsas X diskblock för att hämta motsvarade tabellrader. Till exempel om man i figur 1.3 traverserar hela tabellen genom indexet så måste man läsa 4 indexblock som förut, men 9 radblock.

Normalt är bara primärindexet klustrat. Således är indexen för **pnr** de enda som är klustrade i vårt exempel.

Med vårt antagande i exemplet att $I = 100$ finns det följande antal block på de olika indexnivåerna:

nivå	block
1	1
2	100
3	10000

Totala antalet indexblock blir $I = 1 + 100 + 10000 = 10101$ och eftersom det fanns 100000 rader i tabellen pekar varje lövblock i indexet till i genomsnitt 10 tabellrader[14].

1.7 Kostnadsbaserad optimering

Vi är nu mogna att diskutera betydelsen av kostnadsbaserad optimering och hur den fungerar. Kostnadsbaserad optimering tillämpas vanligen efter vyexpansion så att information av betydelse för optimeringen inte är gömd inuti vydefinitioner. Kostnaden påverkas mycket signifikant av tillgängliga indexstrukturer.

I princip utför kostnadsbaserad frågeoptimering följande steg:

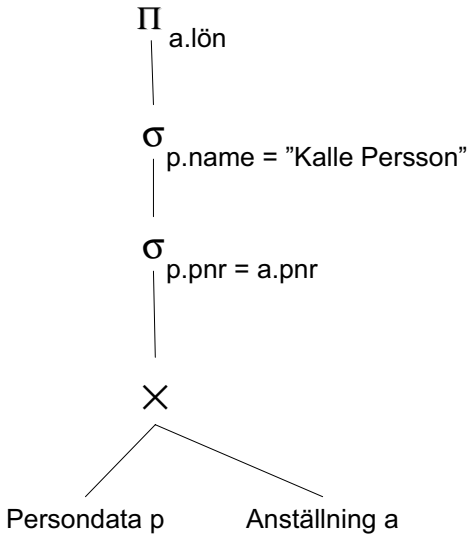
1. Generera alla möjliga exekveringsplaner för en given omskriven databasfråga.

[14] För enkelhets skull antar vi att alla block innehåller samma mängd data.

2. Uppskatta kostnaden för exekveringsplanerna.
3. Välj den billigaste.

Bara sådana exekveringsplaner som producerar korrekt svar på databasfrågan är möjliga exekveringsplaner. Olika exekveringsplaner traverserar databasen i olika ordning med olika kostnad. Detta resulterar i sin tur i olika ordning på de rader som produceras som resultat. Eftersom resultatet av deluttryck i regel är påsar eller mängder av rader, har ordningen i allmänhet inte någon betydelse. Om slutresultatet emellertid har en `order by`-klausul är ordningen signifikant, och systemet kan då behöva införa en explicit sorteringsoperator i exekveringsplanen.

1.7.1 Betydelsen av olika exekveringsplaner



Figur 1.4: Ooptimerad fråga

Figur 1.4 visar exempelfrågan i sektion 1.4 *systematiskt* översatt till relationsalgebra. Algoritmen för systematisk översättning är:

1. Forma kartestisk produkt av raderna i tabellerna i `from`-klausulen. Detta eftersom SQL är en radkalkyl (tuple calculus).

2. Applicera villkoren i `where`-klausulen på resultatet av kartesiska produkten så att bara utvalda rader finns kvar.

3. Projicera resultatkolumnerna i `select`-klausulen på de utvalda raderna i den kartesiska produkten.

Den ooptimerade exekveringsplanen kan tolkas (interpreteras) direkt. Varje operator i algebraträdet producerar därvid en ström av rader[15] som representerar mellanresultat. För att uppskatta kostnaden att utföra planen måste vi för varje nod i algebraträdet räkna ut följande:

1. Storleken på mellanresultat i varje nod, dvs. hur många rader den ström beräknas ha som produceras av noden.

2. Totala kostnaden att utföra en algebraoperator beror på hur många rader som producerats i strömmen från argumentnoderna under. Kostnaden beräknas i termer av *kostnadsenheter* och inte direkt i riktiga sekunder. Vi antar genomgående att det kostar 1000 enheter att läsa 1 diskblock och 1 kostnadsenhet att läsa en rad från resultatström producerad av algebraoperator. Detta eftersom varje rad i en resultatströmmen lagras i primärminnet; den strömmade tolkningen gör att man bara behöver primärminne för ett fåtal rader åt gången.

Vad en kostnadsenhet motsvarar i verklig tid beror på vilken hårdvara man använder, och är inte kritiskt för frågeoptimerarens funktion, men med moderna diskar, där en diskläsning tar 5-10 millisekunder, motsvarar en kostnadsenhet alltså 5-10 mikrosekunder.

I vårt exempel har tabellerna **Persondata** och **Anställning** 10^5 rader var, dvs. $100000/10 = 10000$ diskblock var. Vi får då följande uppskattningar:

1. Att forma ström av kartesiska produkten kräver att $10000 * 10000 = 10^8$ diskblock läses.[16] Eftersom det kostar 1000 enheter att läsa ett block blir kostnaden för kartesiska produkten $1000 * 10^8 = 10^{11}$ enheter.

2. Den kartesiska produkten kommer att producera en ström med $10^5 * 10^5 = 10^{10}$ rader.

3. Att selektera joinvillkoret på resultatet av kartesiska produkten kostar 10^{10} enheter då ju en rad kostar en enhet att läsa från strömmen.

4. Resultatet av joinvillkoret producerar ström med 10^5 rader om vi antar all det alltid finns matchande **pnr** i båda tabellerna.

[15] En sådan ström av rader kallas ibland på engelska *scan*.

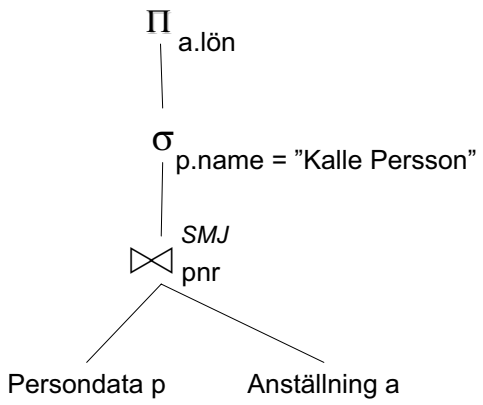
[16] Det blir billigare om man kan läsa in någon av eller bägge tabellerna i primärminnet först. Vi antar emellertid för enkelhets skull genomgående att det bara finns plats för ett diskblock i primärminnet och att det således inte finns tillräckligt primärminne för att läsa in några tabeller alls i förväg.

5. Att selektera **Kalle Persson** från strömmen kostar 10^5 enheter då det ju kostar en enhet att göra selektion från en rad i en ström.

6. Resultatet av selektionen producerar ström med 1 rad om det bara finns en **Kalle Persson**.

7. Projektionen kostar 1 enhet.

Den totala kostnaden blev $10^{11} + 10^{10} + 10^5 + 1$. Om vi antar att 1 enhet tar 10 mikrosekunder (10^{-5} sekunder) att utföra så tar den ooptimerade frågan 1.1 miljoner sekunder (ca 306 timmar) att utföra.



Figur 1.5: Kartesisk produkt ersatt med sort merge join

En uppenbar ineffektivitet med planen i figur 1.4 är formeringen av kartesisk produkt. I exemplet lönar det sig att utföra en join för att matcha kolumnvärden i tabellerna som i figur 1.5. Det finns ett antal olika sätt att joina tabeller. I detta fall vet vi att raderna i båda tabellerna ligger lagrade sorterade i **pnr**-ordning. Vi kan därför läsa igenom raderna i **Persondata** och **Anställning** parallellt för att hitta matchande rader. Den joinmetoden kallas *sort merge join*[17] och beskrivs detaljerat senare. Joinoperatorn i den utvidgade relationsalgebran är därför markerad *SMJ* för att indikera att sort merge join skall användas. Kostnadsuppskattningen blir denna:

1. Kostnad för sort merge join: Kostnad att läsa igenom 10000 block i **Persondata** och 10000 block i **Anställning**, dvs. läsning av 20000 block: 20000*1000 = $2 * 10^7$ enheter.

2. Antal rader som flödar upp från joinen är 100000 om vi antar att det alltid finns matchande **pnr** i båda tabellerna.

[17] Hela termen *sort merge join* är på engelska, så det är ingen felaktig särskrivning att skriva den så.

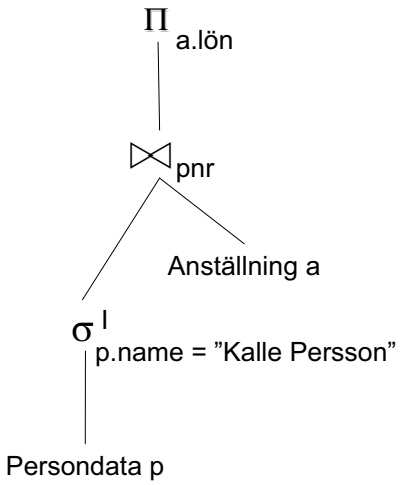
3. Selektionen av **Kalle Persson** från resultatströmmen kostar 100000 enheter.

4. Selektionen producerar 1 rad.

5. Kostnaden för projektionen är 1 enhet.

Den totala kostnaden i detta fall är $20000000 + 100000 + 1 = 20100001$ enheter. Om vi fortfarande antar att 1 enhet tar 10 mikrosekunder (10^{-5} sekunder) att utföra, så tar den här exekveringsplanen 200 sekunder att köra. Det är 5500 gånger snabbare än den ooptimerade planen.

I vårt exempel är argumenten till joinoperatorn hämtade från databastabeller. Argument kan emellertid också vara delresultat levererade som strömmar från andra delplaner, vilket påverkar kostnaden. Till exempel kan det hända att delresultat är mycket litet eller osorterat, vilket starkt påverkar val av join-metod, etc. Optimeraren väljer rätt metod baserat på en uppskattning av storleken på delresultat.



Figur 1.6: Selektion nerflyttad

En viktig observation är nu att vi kan snabba upp exekveringen ytterligare genom att flytta ner selektionen av **Kalle Persson** som i figur 1.6. Det kallas *selektionsnedflyttning* (på engelska *selection pushing*). Selektionen kan i detta fall göras mycket effektivt genom att det finns B-trädsindex på **namn** som kan användas för att snabbt hitta **Kalle Persson**. I den utvidgade relationsalgebran betecknar vi sådan *indexselektion* med $\sigma^{I}_{namn='KallePersson'}$. Kostnadsuppskattning:

1. Kostnaden för indexselektionen blir 4000 enheter (4 diskblocksläsningar, 3 för att söka genom indexet och 1 för att hämta raden från tabellen).

2. Resultatet från indexselektionen producerar 1 rad.

3. Joinen kostar 4000 enheter för att läsa matchande rad från **Anställning** genom B-trädsindex för **pnr** (4 diskblocksläsningar).

4. Som resultat från joinen produceras 1 rad.

5. Projektionen kostar 1 enhet.

Den totala kostnaden blir i detta fall 8001 enheter (motsvarande 0.08 sekunder), vilket är ca 14 miljoner gånger snabbare än ooptimerad kod. Exemplet visar således klart att frågeoptimering lönar sig. Anledningen till den enorma prestandaförbättringen är att SQL-frågor utgör icke-procedurella specifikationer av sökningar över den interna representationen av relationstabellerna. För varje sådan specifikation (fråga) finns det många olika sökstrategier och olika strategier är optimala beroende på vilka data som finns i databasen. Vidare har olika sökstrategier olika komplexitet och frågeoptimeringen förbättrar således komplexiteten hos sökningen genom att ändra sökalgoritm.

En enkel komplexitetsanalys visar att i vårt exempel har den sämsta strategin i figur 1.4 komplexitet $O(N^2)$ där N är databasens storlek.[18] Efter elimination av kartesisk produkt går komplexiteten ner till $O(N)$ i figur 1.5.[19] Den optimala strategin i figur 1.6 har komplexitet $O(log(N))$.[20]

Komplexitetsförbättringen innebär att frågebearbetningen *skalar upp* exekvering av frågor, dvs. databasens storlek kan öka utan att databasfrågorna blir för långsamma, vilket är fallet om sökningen är $O(N)$ eller $O(N^2)$. Skalbarhet är centralt för databassystem.

1.7.2 Heuristiska regler otillräckliga

Man kan sammanfatta optimeringen i det hittillsvarande exemplet med att vi tillämpat två tumregler: eliminering av kartesisk produkt och selektionsnedflyttning. Man kan förledas tro att det räcker med att tillämpa sådana heuristiska regler och att den mer noggranna kostnadsbaserade optimeringen inte behövs.

Låt oss variera vårt exempel så att vi inte har något index för **namn**. Då blir kostnaden för planen i figur 1.5 oförändrat 20100001 enheter.

För planen i figur 1.6 måste vi byta algebraoperatorn $\sigma^{I}_{namn='KallePersson'}$ (indexselektion) mot $\sigma_{namn='KallePersson'}$, vilket betecknar oindexerad selektion som läser igenom hela tabellen **Persondata**. Detta ökar kostnaden att selektera **Kalle Persson** drastiskt till kostaden 10^7 enheter då 10^4 block måste läsas.

Eftersom övriga kostnader för planen i figur 1.6 blir oförändrade blir den nya totala kostnaden $10000000 + 4000 + 1 = 10004001$. Det är fortfarande dubbelt så snabbt som plan 1.5, så selektionsnedflytting lönar sig fortfarande, men med andra tabeller och andra data kan det hända att det faktiskt blir *långsammare*. Det

[18] Domineras av den kartesiska produkten som producerar N^2 tupler.
[19] Domineras av sort merge join.
[20] Domineras av trädsökning i indexen.

är alltså inte säkert att en heuristik som selektionsnedflytting ger en snabbare plan.

Moderna optimerare grovklassificerar först den bearbetade frågan för att utröna huruvida selektionsnerflyttning lönar sig. Det lönar sig i allmänhet om den selekterade kolumnen är indexerad.

Det finns till och med fall då det lönar sig att behålla en kartesisk produkt framför att göra join. Antag till exempel att **Anställning** innehåller högst en rad. Då blir det billigast att göra kartesisk produkt. Sådant kan inträffa om indata produceras som resultat från underliggande delplan.

1.7.3 Joinmetoder

Det används vanligtvis tre olika joinmetoder i moderna databashanterare: *sort merge join*, *nested loop join* och *hash join*. Vi går nu igenom dessa algoritmer och jämför när de är användbara.

Sort merge join

Ovanstående exempel illustrerade sort-merge-join. Pseudokoden för att göra sort merge join mellan tabellerna T och U (skrivs $T \bowtie^{SMJ} U$) är:

```
{
  Stream T, U, R;
  Tuple t, u;
  open(R,'o'); // Öppna resultatström R
  open(T,'i'); // Öppna 1:a inströmmen
  open(U,'i'); // Öppna 2:a inströmmen
  t = next(T); // läs 1:a raden till t
  u = next(U); // detsamma för u
  while (not(eof(T)) and not(eof(U))) // så länge båda
                                      // inströmmarna
                                      // har mer data
  {
    if(t.k > u.k) u = next(U); // om nyckelfältet är
                               // mindre i u så
                               // flytta fram U
    else if(t.k < u.k) t = next(T); // detsamma för T
    else emit(t + u, R); // skicka  t och u
                         // konkatenerade som
                         // nästa resultatrad
  }
  close(T);
  close(U);
  close(R); // stäng alla strömmar
}
```

Funktionen `next` returnerar nästa rad i en ström. Alla strömoperationer är *buffrade* så att det finns en intern buffer för varje ström med storlek[21] B till vilken raderna först läses vilket är viktigt för att snabba upp strömmar från disk, eller från andra noder i en distribuerad databas. Funktionen `eof` testar om strömmen är slut. Funktionen `emit` sänder ny rad till resultatström (här `R`), som normalt också är buffrad så att data inte levereras till ovanförliggande operator förrän bufferten fyllts.[22]

För att sort merge join skall fungera måste indata vara sorterade. Om så inte är fallet kan optimeraren lägga in explicit sorteringoperator i exekveringsplanen. Kostnaden för sortering måste då tas med i kostnadskalkylen; i allmänhet lönar sig inte sort merge join om indata inte redan är sorterade. Mer om detta i sektion 1.7.4.

Om indata är strömmar mot tabeller med blockstorlek B och kostnaden för att läsa diskblock $Bcost$[23] så blir kostnaden $Bcost * (card(T) + card(U))/B$. Om indata är sorterade resultatströmmar från andra algebraoperatorer och kostnad att accessa strömelement $Scost$[24] så blir kostnaden $Scost * (card(T) + card(U))$ om vi antar att strömningen sker i primärminne.

Notera vidare att sort-merge-join är symmetrisk i den meningen att det blir exakt samma kostnad och resultat om operanderna kastas om.

Nested loop join

Nested loop join $T \bowtie^{NLJ} U$ traverserar ena operandströmmen och för varje rad där söker algoritmen matchande rad i andra operanden. Algoritm:

```
{
   Stream T, U, R;
   Tuple t, u;

   open(R,'o');
   open(T,'i');
   while (not(eof(T)))
   {
     t = next(T);
     open(U,'i');
     while (not(eof(U))
     {
       u = next(U);
       if(t.K == u.K) emit(t + u, R);
     }
     close(U);
   }
```

[21] I våra exempel antar vi konstant 100 rader men i praktiken måste man ta hänsyn till radlängden också.

[22] Operatorn '+' betyder här radkonkatenering.

[23] 1000 i våra exempel.

[24] 1 i våra exempel.

```
   close(T);
   close(R);
}
```

Ovanstående definition används då man gör join över strömmar eller oindexerade tabeller. Metoden är inte symmetrisk och den första strömmen (`T`) bör vara mindre. Kostnaden blir $(card(U)/B + card(U) * card(V) * Bcost$ om operanderna är tabeller[25] och $Scost * card(U) * card(V)$ om de är strömmade mellanresultat.

Om en av tabellerna är tillräckligt liten för att få plats i primärminnet är det mycket fördelaktigt att ha den som andra (inre) operand. Hur stor blir kostnaden då?

Den andra operanden refererar dock vanligtvis till tabell indexerad på jämförelsevillkoret. I sådana fall kan man använda en variant av nested loop join som utnyttjar indexet för att hitta matchande rader i den inre tabellen. Detta kallas *indexerad nested loop join*, $T \bowtie^{INLJ} U$. Algoritm:

```
{
   Stream T, U, R;
   Tuple t, u, r;

   open(R,'o');
   open(T,'I');
   while (not(eof(T)))
   {
     t = next(T);
     openIndexScan(U, U.k, t.k);
     while (not(eof(U))
     {
       r = next(U);
       u = getRow(U,r);
       emit(t + u, R);
     }
     close(U);
   }
   close(T);
   close(R);
}
```

I detta fall öppnar funktionen `openIndexScan` en ström till ett index över namngiven indexerad kolumn **k** i tabellen **U**, `U.k`, som andra argument samt aktuellt nyckelvärde i indexet `t.k` som tredje argument. Funktionen `next` hämtar i detta fall nästa matchande indexpost genom att traversera B-trädet.

Kostnad: Om varje nod i B-trädet innehåller I nyckel/pekar-par kommer B-trädet att ha djup $log_I(card(U))$ och det behövs således
$log_I(card(U)) + 1$ diskblocksläsningar för att traversera B-trädet och hämta en

[25] Kostnaden blir $Bcost * (card(U)/B + card(U) * card(V)/B)$ om den inre tabellen har samma klusterordning som den yttre, fast då är sort merge join billigare.

rad från U. Med *selektivitet* av ett villkor P, $s(P)$, avser vi hur stor andel rader som finns kvar efter det att P applicerats. Om selektiviteten av join-villkoret är $s(J(T,U))$ kommer det i genomsnitt att finnas $card(U) * s(J(T,U))$ matchande rader i U för varje rad i T. Om B betecknar antal rader per block som förut blir totala kostnaden:
$Bcost * (card(T)/B + card(T) * card(U) * s(J(T,U)) * (log_I(card(U)) + 1))$.
I specialfallet att exakt en rad matchar (som vi antog i vårt exempel) är $card(U) * s(J(T,U)) = 1$ och vi får då kostnaden:
$Bcost * (card(T)/B + card(T) * (log_I(card(U)) + 1))$

Vi antar här i våra kalkyler att alla block måste läsas in in minnet när de accessas; i praktiken har databashanteraren plats för ett antal block i primärminnet, vilket minskar kostnaden. I synnerhet lönar det sig att alltid hålla rotblocket i index i minnet då de ju alltid måste traverseras för att hämta nya data genom indexet. Hur stor blir kostnaden då?

Hash join

Om tillräckligt minne finns för att hålla alla nycklar i joinvillkoret för ena operanden fungerar *hash join*, $T \bowtie^{HJ} U$, bäst:

```
{
   Stream T, U, R;
   Tuple t, u;
   HashTable h;

   open(R,'o');
   h = createHashTable();
   open(T, 'i');
   while (not(eof(T)))
   {
     t = next(T);
     putHash(h,t.k,t);
   }
   open(U, 'i');
   while (not(eof(U))
     {
       u = next(U);
       t = getHash(h, u.k);
       if(t != NULL) emit(t + u, R);
     }
   close(U);
   close(T);
   close(R);
}
```

Problemet med hashning är att metoden kräver att hashtabellen helt får plats i primärminnet; i annat fall uppstår "thrashing", vilket innebär att delar av

processens primärminnesdata flyttas fram och tillbaka mellan primärminnet och disken, med mycket dåliga prestanda som följd. Det finns dock partitionerade varianter av hash join som hanterar detta också.

Kostnad: $Bcost * (card(T)/B + card(U)/B)$ under förutsättning att hashtabellen **h** får plats i primärminnet och att båda operanderna är tabeller. Vad blir kostnaden om de är strömmar?

1.7.4 Betydelse av sorteringsordning

Som framgick av beskrivningen av sort-merge-join kan systemet behöva lägga in en explicit sorteringsoperator, $SORT(T)$ för att sortera mellanresultat T. Likaså kan slutresultatet behöva sorteras om SQL-satsen innehåller en `order by`-klausul. Ofta kan systemet dock utnyttja att det finns B-trädsindex på olika fält för att generera rader i en viss ordning.

Eftersom det i allmänhet inte finns minne för ett helt mellanresultat måste man använda så kallad samsortering (på engelska *external sorting* eller *merge sort*) där materialet gås igenom i ett antal faser där man blandar (eng. *merge*) ökande sorterade sviter parvis. Samsortering har i allmänhet kostnad $Bcost * N/B * log_B(N)$, där $N = card(T)$. Antal faser är $log_B(N)$ eftersom B rader får plats i minnet åt gången för sortering där och N/B block måste läsas i varje fas. Notera att B i praktiken kan vara ett ganska stort tal, till exempel 500. Hur många faser behövs det då för att sortera 10^8 rader?

Som ett exempel på att sortering kan löna sig, antag att vi har en tabell över olika försäljares framgångsrikhet, sorterad på **pnr**:

```
Försäljning(pnr, namn, kr)
```

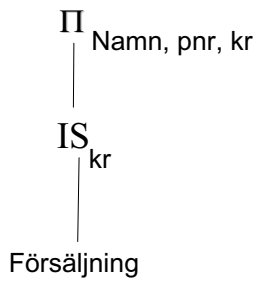
Vad man kanske vill ha reda på är vilka som säljer mest och man ställer därför denna fråga:

```
select pnr, namn, kr
from försäljning
order by kr descending;
```

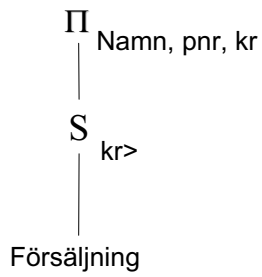
Antag att vi sätter ett (oklustrat) B-trädsindex på **kr**. Vi jämför planerna i figur 1.7 och 1.8.

I figur 1.7 utnyttjar vi index scan av B-trädsindexet på `kr` (IS_{kr}) för att slippa sortera. Antag att vi har 100000 rader med 10 rader per radblock och 100 rader per indexblock som förut. Vi måste traversera hela indexet, vilket kräver $1 + 100 + 10000 = 10101$ diskblock, dvs 10^7 kostnadsenheter. Eftersom indexet är oklustrat måste emellertid dessutom 1 diskblock läsas för varje rad i tabellen, dvs 10^5 block, vilket kostar 10^8 kostnadsenheter. Således blir kostnaden för IS_{kr} $1.01 * 10^8$.

Nu jämför vi hur mycket det kostar att läsa igenom tabellen sekventiellt och sedan sortera med flerfasig samsortering. Eftersom det bara får plats 10 rader att sortera i primärminnet åt gången kan vi anta att sorteringen kan göras i fem pass med 10000 lästa block i varje pass (först 10000 sviter, därefter 1000,



Figur 1.7: Index scan



Figur 1.8: Explicit sortering

därefter 100, därefter 10, och därefter resultatsviten) och således måste 5000 block läsas, vilket kostar $5 * 10^7$ kostnadsenheter vilket är 5 gånger billigare än att traversera det sorterade, men oklustrade, indexet.

Det kan vara lämpligt att ordna raderna i sina tabeller enligt den sorteringsordning som vanligtvis behövs. SQL ger möjlighet till att definiera sådan explicit ordning när man skapar tabellen.

1.7.5 Funktioner i utvidgad relationsalgebra

En exekveringsplan är ett funktionellt uttryck i utvidgad relationsalgebra. Den utvidgade relationsalgebran är ett parameterfritt funktionellt språk som utnyttjar länkade block, index och andra datastrukturer som representerar databasens innehåll. Alla operatorer är i allmänhet *strömmade*. Strömning gör att mellanresultat i allmänhet använder begränsade minnesresurser. I vissa lägen *materialiseras* dock mellanresultat som temporära tabeller när det är fördelaktigt, m.h.a. en speciell materialiseringsoperator, $MAT(S)$, som tar en ström S som argument och lagrar dess resultat i en tillfällig tabell, mot vilken resultatströmmen sedan kopplas.

Exekveringsplaner för moderna optimerare har bl.a. följande utvidgade re-

lationsalgebrafunktioner:

- *Tuple scan*, sekventiell genomläsning av raderna i en tabell i lagringsordningen producerade som en ström av rader.
- *Index scan*, traversering av de delar av ett index (till exempel B-träd) som uppfyller sökvillkor (till exempel `income > 5000`) för att producera ström av de tabellrader som indexvillkoret utpekar från indexet. Man skiljer på klustrad och oklustrad index scan.
- *Join-metod*, den algoritm som används för att utföra en join. Man använder numera i huvudsak tre algoritmer, *nested loop join*, *hash join* eller *sort-merge join*, vilka förklarats ovan. Join-metoderna tar två eller flera strömmar som input och producerar en ström av rader som resultat.
- *Sortering* av (del-)resultat. Tar en ström av oordnade rader som input och producerar en ström av ordnade rader som output. I allmänhet används en flerfasig samsorteringsmetod.
- *Duplikateliminering* tar bort redundanta rader i en ström. Kräver en temporär datastruktur (till exempel ett B-träd) för att hålla reda på vilka rader som producerats.
- *Materialisering* bygger upp en temporär tabell från rader i en ström. Tabellen kan också innehålla index som då också byggs dynamiskt.

1.7.6 Databasstatistik

För att uppskatta kostnaden av en exekveringsplan behöver kostnadsmodellen beräknas baserat på statistik av databasens innehåll. Bland annat följande statistik upprätthålls av databashanteraren:

1. *Kardinaliteten*, $card(T)$, av varje tabell T, dvs hur många rader tabellen har.
2. Antalet *distinkta värden* $D(T.c)$ i varje kolumn, $T.c$. För nycklar är $D(T.c) = card(T)$; för andra attribut är $D(T.c) <= card(T)$.
3. Max- och minvärden i varje kolumn, $max(T.c)$ och $min(T.c)$.
4. *Fördelningen* av datavärden i en kolumn. Tidiga optimerare antog rektangulärfördelning,[26] men moderna databashanterare upprätthåller histogram över datafördelningen.

Kostnaden är exempelvis mycket beroende av hur mycket data som produceras av en given relationsalgebraoperator, och därför ingår det i kostnadsmodellen att uppskatta hur selektivt varje villkor är, dvs *selektivitet*, $s(pred)$. Selektiviteten uppskattas m.h.a. databasstatistiken. Till exempel med rektangulärfördelade

[26] Vilket gör att bara $max(T.c), min(T.c)$ och $card(T)$ behöver lagras.

data i $T.c$ blir
$s(\texttt{T.c} =' \texttt{x}') = 1/D(T.c)$. I vårt exempel, om vi antar att alla **namn** är unika så är selektiviteten av villkoret `namn = 'Kalle Persson'` 1/100000. För att selektivitet av olikheter, till exempel villkor som `namn>'I' and namn<'K'` utnyttjar optimeraren kunskap om datafördelningen i kolumnen.

Databasstatistiken uppdateras genom ett systemprogram som läser alla tabeller för att samla in statistik. Eftersom databasstatistiken inte ändras speciellt intensivt kan statistikinsamlingen köras i bakgrunden vid lämpliga tillfällen, till exempel en gång i månaden eller efter stora inladdningar av data i databasen. Databasadministratören kan bestämma när statistikinsamling skall ske.

För den totala kostnaden att utföra ett givet utvidgat relationsalgebrauttryck måste systemet ta hänsyn till flera faktorer, till exempel:

1. Antal lästa block av data från disk. Denna siffra dominerar normalt för en diskbaserad databas.

2. CPU-tid för att utföra uttrycket.

3. Kommunikationstid för de fall då data ligger på annan nod som systemet därför måste kommunicera med. Detta är framför allt en viktig faktor i kostnadsmodeller för distribuerade databaser.

Kostnadsmodellen uppskattar kostnaden som ett vägt medelvärde av dessa faktorer. Normalt dominerar tiden att läsa block från disk.

För att korrekt uppskatta kostnaderna är följande information av stor betydelse:

1. Storlek på mellanresultat. Ju mer data som strömmar genom algebrafunktionerna ju dyrare blir förstås frågeexekveringen. Storleken på mellanresultat uppskattas m.h.a. databasstatistik.

2. Hur data är *klustrat*, dvs vilka data som ligger intill varann. Till exempel ligger normalt tabellrader sekventiellt på disk vilket innebär att kostnaden i antal lästa diskblock att sekventiellt läsa alla raderna (tuple scan) i en tabell T med $N = card(T)$ rader (*kardinalitet*) och blockstorlek B blir N/B blockläsningar. Å andra sidan är sekundärindex normalt inte klustrade och därför blir kostnaden minst N blockläsningar att sekventiellt gå igenom ett index för att läsa motsvarande tabellrader (index scan).

3. *Sorteringsordning* av tabellrader och index.

4. Kostnad att traversera olika typer av index. Till exempel är antalet block som måste läsas för att hitta pekaren till en enskild post i ett B-träd $log_I(N)$, där I är antal nyckel/pekar-par i varje indexnod som förut, och N är tabellens kardinalitet. Antal block som måste läsas för att i ett B-träd hitta alla poster vars nyckel ligger i ett intervall $[L, U]$ kan uppskattas genom att m.h.a. databasstatistik finna hur många poster M det beräknas finnas i intervallet. Antal lästa block blir då $M/I * log_I(N)$. Att införa en ny typ av index i ett databassystem kan vara mycket komplicerat och

kräver bl.a. noggrann analys av kostnader och andra egenskaper hos den nya indexstrukturen.

5. Kostnadsmodeller för lika typer av *join-algoritmer*. Dessa konstrueras genom analys av de specifika join-algoritmerna.

6. Kostnad att utföra primitiva operatorer. I SQL antas de primitiva jämförelseoperatorerna ($>, <$, etc.) normalt ha försumbar kostnad, vilket utnyttjas i optimeringsheuristiken för att undvika att generera många exekveringsplaner.

7. Modeller för hur man beräknar selektivitet av logiska uttryck. Till exempel antag att $s(\texttt{avd} =' \texttt{adm}') = 0.3$ och $s(\texttt{salary} > \texttt{50000}) = 0.1$. Då blir $s(\texttt{avd} =' \texttt{adm}' \texttt{ and salary} > \texttt{50000}) = 0.03$ under förutsättning att värdena i kolumnen *avd* och *salary* är statistiskt oberoende. Tidiga optimerare antog alltid statistiskt oberoende mellan kolumner men numera har man modeller som mäter och tar hänsyn till statistiska beroenden.

Man kan här konstatera att komplexa statistiska modeller ger precisa optimerare, till priset av kostnaden att beräkna statistiken. Databasstatistiken behöver emellertid inte vara exakt då den inte i första hand används för att exakt uppskatta kostnaden av ett uttryck, utan snarare är till för att jämföra kostnader för olika strategier. Om databasstatistiken är inaktuell blir frågorna eventuellt långsammare men svaren är fortfarande korrekta. Moderna kostnadsmodeller är därför designade att producera användbar statistik för lägsta möjliga beräkningskostnad. Man har till exempel utvecklat metoder att inkrementellt upprätthålla approximativa histogram över datafördelningar i kolumner.

1.8 Optimeringsmetoders komplexitet

Kostnadsbaserad optimering är NP-komplett över frågans storlek, Q (dvs komplexitet $O(2^Q)$ eller sämre). Så kallad *dynamisk programmering* snabbar upp kostnadsbaserad optimering jämfört med naiv generering av alla möjliga exekveringsplaner, men komplexiteten är fortfarande NP-komplett fast $O(Q^2)$ i bästa fall. Detta har till följd att kostnadsbaserad optimering bara kan tillämpas på relativt enkla frågor; det brukar i allmänhet fungera mycket bra upp till 8 join. För större uttryck gör optimerarna i allmänhet bara partiell kostnadsbaserad optimering och tillämpar heuristiska metoder på delar av frågan, eller lämnar delar av frågan ooptimerad.

Heuristiska metoder för sökning bland exekveringsplanerna i samband med kostnadsbaserad optimering (ej att förväxla med den typ av heuristisk optimering som går ut på att göra om exekveringsplanen enligt tumregler) baseras på att starta med en initial plan och sedan systematiskt modifiera den så länge som kostnaden går ner bland intilliggande planer (hill climbing). Man får då förstås definiera vad man menar med 'intilliggande' så att de kan genereras systematiskt. De har i allmänhet komplexitet $O(Q^2)$ men kan ge suboptimala exekveringsplaner.

Slumpmässiga metoder (även kallade Monte Carlo-metoder) genererar i princip slumpmässigt olika exekveringsplaner, uppskattar deras kostnader m.h.a. kostnadsmodellen, och väljer den billigaste. Slumpmässiga metoder har fördelen att de konvergerar mot en optimal plan och kan därför avbrytas när som helst. En bra strategi är att slumpmässigt generera en plan och sedan göra heuristisk hill climbing för att hitta den billigaste planen som kan nås så länge kostnaden går ner.

1.9 De viktigaste begreppen

Fråga (engelska: **query**). En deklarativ specifikation, uttryckt i ett frågespråk som SQL, av det svar man önskar få från en sökning i databasen.

Exekveringsplan (engelska: **execution plan**). En steg-för-steg-beskrivning av hur en fråga ska köras av databashanteraren. En och samma fråga brukar kunna översättas till många olika exekveringsplaner.

Frågeoptimering (engelska: **query optimization**). Processen när databashanteraren väljer bland de olika exekveringsplaner som en fråga kan översättas till, för att hitta den snabbaste. (Eller i alla fall en som är tillräckligt snabb.)

Heuristisk frågeoptimering (engelska: **heuristic query optimization**). Frågeoptimering som görs genom att databashanteraren tillämpar några tumregler för hur en exekveringsplan ska se ut, till exempel att operationen *selektion* (σ) bör utföras före operationen *join* ($\bowtie$). Heuristisk frågeoptimering är relativt enkel och snabb att utföra, jämfört med kostnadsbaserad frågeoptimering, men är å andra sidan sämre på att hitta optimala exekveringsplaner. Tar normalt inte hänsyn till lagringsstrukturer, eller alternativa algoritmer för de operationer som ska utföras.

Kostnad (engelska: **cost**). Kostnaden för att köra en fråga kan mätas på olika sätt, men för det mesta är det bara den förväntade tiden att köra frågan som man bryr sig om. Kostnaden kan räknas i sekunder, eller i mer abstrakta kostnadsenheter. I en vanlig diskbaserad databas är det åtkomst av disken som tar mest tid, och kostnaden kan därför också räknas i antalet diskaccesser.

Kostnadsbaserad frågeoptimering (engelska: **cost-based query optimization**). Frågeoptimering som görs genom att databashanteraren jämför den uppskattade kostnaden för olika exekveringsplaner, och väljer den billigaste (dvs snabbaste). Kostnadsbaserad frågeoptimering är relativt komplicerad och långsam att utföra, jämfört med heuristisk frågeoptimering, men är å andra sidan bättre på att hitta optimala exekveringsplaner.

Kostnadsmodell (engelska: **cost model**). Den matematiska modell som en kostnadsbaserad frågeoptimerare använder för att uppskatta kostnaden för en exekveringsplan.

1.10 Litteratur

- Connolly et al, fjärde upplagan: kapitel 21
- Date, åttonde upplagan: kapitel 18
- Elmasri/Navathe, fjärde upplagan: kapitel 15
- Silberschatz et al, fjärde upplagan: kapitel 13-14