OPTIMIZACION DE CONSULTAS[1].

1. Premisas básicas.

Para una Relación R, sea T_R = número de tuplas en ella. Sea B_R el número de bloques en los cuales se acomodan todas las tuplas de R. Además $B_R << T_R$ (mucho menor).

T_R / B_R = Número de tuplas por bloque.

Eje 11.1. *bloques de 1024. bytes.*

R= 1 millón de tuplas de 100 byte de long.

$T_R = 1.000.000$; luego $B_R = 100.000$

1.1. Costo de leer ó escribir una relación: Número de bloques que deben ser leídos desde disco a memoria principal ó escritos en sentido inverso. Para las operaciones de lectura y/o escritura masiva, si el almacenamiento es muy compacto, significa un costo bajo; en otro caso implica alto costo.

1.2. Uso de los índices: Hash o B-tree son los ideales; permiten ejecutar la selección de tuplas rápidamente. Asumimos que toda relación R tiene un índice en un atributo A. Luego $\sigma_{A=C}(R)$ tiene aproximadamente un costo del número de bloques donde existen tuplas de R que cumplen tal condicion. Consideramos descartable el número de bloques leídos para obtener el índice, pues dicho número es usualmente comparable a, o más pequeño que, el número de bloques de R que debo acceder.

Indice Cluster : Si tenemos tal índice en A, el número de bloques guardando tuplas con A = *c*, es aproximadamente el número de bloques en el cual esas tuplas pueden ser almacenadas. En el mejor de los casos un solo bloque para todos, en el peor un bloque por tupla.

Indice no cluster: Asumimos que las tuplas de R que cumplen la condición A=c, aparece cada una en bloques separados. Luego los bloques leídos equivalen al número de tuplas que cumplen tal condición.

---

[1] Tomado de : Ullman, Jeffrey D. **Database and Knowledge-Base System**. Vol II. Cap. 11. pp: 633-673. Computer Science Press. 1989.

Traducido por : *John Freddy Duitama Muñoz*. Profesor U. de A. Facultad de Ingeniería.

Eje: en un índice B-tree denso, solo será coincidencia si aparecen en el mismo bloque físico dos tuplas con el mismo valor para A.

1.3. Tamaño de la Imagen : Denotado por $I_{R.A}$, es el número esperado de diferentes valores de A hallados en R. Asumiendo que los valores son igualmente probables de ocurrir, podemos estimar el número de bloques recuperados en respuesta a $\sigma_{A=C}$ (R):

- Si tenemos un indice no-cluster , T_R / $I_{R.}$ Número esperado de tuplas recuperadas.
- Si hay un índice cluster en A, recuperamos cerca de B_R / $I_{R.}$ bloques, pues las tuplas escogidas están ocupando ese número de bloques.

Eje:

Sea R con T_R = 1.000.000 , B_R = 100.000 $I_{A\,R}$ = 50000

Sin cluster => 1.000.000 tuplas . / 50.000 diferentes valores de A. = 20 lecturas.

Con cluster => 100.000 bloques / 50.000 valores de A = 2 lecturas.

1.4 Bases de la optimización. Hay dos tipos básicos.

Manipulación algebraica : independiente de los datos y de su estructura física actual.

Estimación de costos : Accesos a disco. Considera por ejemplo el uso de índices; selecciona, entre varias alternativas, la mejor estrategia para los datos y las estructuras de almacenamiento que se usen.

Ejemplo 11.2.

Manipulación algebraica : Sea la relación AB con atributos A y B. Sea la relación BC con atributos B y C. Sea :

$\sigma_{A=d}$ (AB $\otimes$ BC) *(11.1)* $\otimes$ *Léase reunión natural.*

Tal consulta es más veloz si primero ejecuto la selección y despues el join, independiente de la presencia o ausencia de índices en sus atributos.

($\sigma_{A=d}$ (AB)) $\otimes$ BC *(11.2)*

En general, los join son más caros que las selecciones, porque requieren acceder (al menos una vez) todo bloque en los que tuplas de estas dos relaciones residan; además se requieren almacenar resultados temporales. Por otro lado la relación (AB $\otimes$ BC) puede

ser mucho mayor que la relación AB o que la relación BC, si la tupla típica de AB tiene un valor B que es compartido por varias tuplas de BC.

En promedio, tomar primero la selección, reduce el número de tuplas en el primer argumento del join por un factor $I_{A.}$, lo que provoca bajar por este factor al tamaño del resultado del join. (una cifra significativa)..

- La Selección en la fórmula *11.1.* es de menor costo que en el caso *11.2.*, solo si el resultado del join es mucho más pequeño que la relación AB; ó sea, si la tupla típica de AB tiene una probabilidad baja de hacer join con una tupla de BC. (caso poco común).
- En la mayoría de los casos, el costo de la selección en la opción *11.2.* no es mayor que en la Fórmula *11.1.*; inclusive puede ser mucho menor, si el resultado del join es mucho más grande que AB y/ó si hay un índice para A en la relación AB que pueda ser usado en el caso *11.2.* pero no en el caso *11.1.*

La regla, haga selección tan pronto como sea posible es heurística y tiene sus excepciones. Si BC está vacia, el producto cartesiano será vacio y se hace innecesaria la selección.

2. Optimización de la selección en SYSTEM R.

Sea *SELECT* $A_1, \ldots, A_n$
FROM R *(11.3.).*
WHERE C_1 *AND* C_2 *AND . . . ;*

C_i = Condiciones que involucran R y que pueden a su vez estar conformadas por subcondiciones conectadas por *AND, OR, NOT, IN,* etc. ; pero con una estructura diferente a C_1 AND C_2

Este algoritmo exige que las condiciones se descompongan tan elementalmente como sea posible en condiciones conectadas por el AND.

SYSTEM R toma ventaja de los índices siempre que sea posible. P.e. si existe C_i de la forma A=c y hay un índice en A se obtienen las tuplas que cumplan esta condición, luego examino en las tuplas obtenidas las demás condiciones.

<u>Ejemplo 11.3</u>.

Cliente(nombre, dirección,balance);

Orden(#Orden,fecha,cliente)

Item(#orden,producto,cantidad)

Alternativa de diseño físico : índice cluster entre ordenes e Item por #orden. Indices no cluster en nombre, cliente y producto.

Asumimos que cada cliente ha colocado varias ordenes y que cada orden incluye varios Item.

2.1. Algoritmo de optimización para consultas simples.

Es un sistema ENUMERATIVO; es decir, existen un número de opciones en un menú preseleccionado; el procesador de consultas estima el costo de cada opción y escoje la mejor.

En situaciones complejas, por ejemplo una consulta con varias reuniones naturales, pueden existir miles de opciones. En un caso simple el número de opciones es poco.

El diseñador del optimizador parte del presupuesto que el tiempo consumido en optimizar una consulta típica, será compensado en el tiempo de ejecución de esta; lo que es altamente probable en relaciones de gran tamaño.

Como no es posible enumerar todos los modos imaginables de implementar una consulta, SYSTEM R, como cualquier sistema de optimización, limita el espacio de estrategias consideradas incluyendo únicamente aquellas de la forma : *Seleccionar una de la condiciones* C_i *, hallar todas las tuplas que satisfacen la condición y luego examinar estas tuplas resultantes para evaluar cuales de ellas satisfacen las otras condiciones.*

SYSTEM R también considera estrategias en las que comenzamos *examinando TODAS las tuplas de R y vemos cuales de ellas satisfacen todas las condiciones.*

ALGORITMO 11.1.

ENTRADA : Una consulta de la forma *11.3.*, y además información acerca de que índices existen en la relación, el número estimado de tuplas T_R, el número mínimo de bloques requeridos para almacenar $R = B_R$, el tamaño estimado de la imagen para los índices es I_R.
.

SALIDA : Una manera de computar la respuesta a la consulta.

METODO : Se considera la lista de alternativas para obtener R completa o, aplicando una condición, obtener a una selección de R. Después de seleccionar un método aplicamos las condiciones restantes a las tuplas obtenidas.

Para los métodos que involucran la escogencia de cual índice ó condición usar, debemos considerar todas las posibles alternativas. El costo se juzga con base en *T, B* e *I*. El método de más bajo costo es la salida para el algoritmo. Las opciones se listan en orden de costo, empezando por las de costo más bajo[2].

Opción 1: Obtener tuplas de *R* que satisfacen una condición de la forma *A=c.* Donde *A* tiene un índice cluster. Sea I_R el tamaño de la imagen del índice; luego, el número de bloques que deben leerse para obtener estas tuplas será igual al número de bloques que ellas ocupen si se acomodan de una manera compacta. Asi el número de bloques leídos para este método será **B/I**.

Opción 2 : Usar un índice cluster en un atributo *A*, donde $A \ \theta \ C$ es una de las condiciones C_i y $\theta \in$ (<, <= , >, >=). A continuación aplicamos las condiciones restantes al resultado. En este caso requerimos acceder cerca de $\mathbf{B_R\ /\ 2}$ bloques, pues :

a. En promedio requerimos leer cerca de la mitad de las tuplas y

b. Con un índice cluster estas se almacenan compactas en cerca de B/2 bloques.

Nota: El caso donde θ equivale a ≠ es omitido aquí, pues este simbolo implica selección de todas las tuplas y estamos suponiendo una selección más limitada.

Opción 3 : Si hay un índice no cluster que coincida con una condición *A=c*, use ese índice para hallar todas las tuplas que cumplen tal condición y aplique las otras condiciones a las tuplas resultantes. Siendo I_R la imagen del índice, deben recuperarse $\mathbf{T_R\ /\ I_R}$ tuplas, probablemente desde bloques diferentes.

Opción 4. : Si *R* está almacenado en un archivo independiente, podemos simplemente leer todas las tuplas y aplicar las C_i a éstas. El costo será $\mathbf{B_R}$, número de bloques donde está esparcida *R*.

---

[2] System R asume que solo hay un índice entre las columnas de las C_i . Para una consulta con condiciones que involucren a varios índices, por ejemplo: *Where A= 1 and B= 17* , con índices en A y B, es mejor intersectar en memoria la colección de apuntadores (índices) obtenidos por A= 1 y por B=17; luego acceder el disco solo para aquellos apuntadores que pertenezcan a la intersección.

Opción 5 : Si R no está almacenada independientemente , pero tiene un índice cluster en un atributo ó en una colección de atributos, sin importar que no estén involucrados en la condición de la consulta, se usa tal índice para obtener todas las tuplas de R y aplicar luego a ellas las condiciones. Su costo también es B_R , pues cualquier índice cluster garantiza que se pueden obtener todas las tuplas en no más lecturas que bloques utilizados en almacenarlas[3].

Opción 6 : Si hay un índice no cluster en un atributo A y $A\ \theta C$ es una condición donde $\theta \in (<, >, >=, <=)$; use el índice para obtener las tuplas de R que satisfagan la condición y aplique las otras condiciones al resultado. El costo es T_R **/ 2**, pues podemos esperar recuperar cerca de la mitad de los registros de R. Además las tuplas estarán dispersas independientemente en los bloques.

Opción 7 : Use un índice no cluster de cualquier clase para hallar las tuplas de R y aplicar todas las condiciones a ellas. Su costo es T_R .[4]

Opción 8 : Si ninguna de las anteriores opciones es posible, simplemente recupere todos los bloques que podrían contener tuplas de R. El costo de éste método es T_R ó quizas más , si no podemos asegurar cuales bloques contienen tuplas de R.

Ejemplo 11.4. **:**

SELECT #orden

FROM Item

WHERE cantidad >= 5 AND producto = 'Brie';

Supuestos**:** *índice cluster para item por #orden e índices no cluster en producto y cantidad. Suponga 1000 tuplas en item, T_R =1000 y que se acomodan 10 tuplas por bloque, B_R =100. Sea la imagen de producto I_R =50; Es decir, hay 50 diferentes tipos de productos en las ordenes. Para este ejemplo son irrelevantes las imagenes de los otros índices.*

Solución:

Opción 1: No factible; no hay índices cluster en las condiciones de nuestra consulta.

Opción 2: Idem 1.

---

[3] N. del T: Puedo apoyarme en el D.de D. para conocer los bloques que contienen al cluster y ahorrarme el barrido del índice. Asumiendo que la relación está almacenada en varios grupos de bloques, cada grupo con bloques contiguos.

[4] N. del T: Idem nota anterior.

Opción 3 : tenemos el índice no cluster sobre producto que hace parte de las C_i . Costo : $T_R / I_R = 1000/50 = 20$.

Opción 4 : Leer todas las tuplas. Si los item están almacenados independientemente $B_R = 100$.

Si los item tienen un cluster con ordenes, no es aplicable la opción 4.

Opción 5 : Si hay cluster entre ordenes e item. $B_R = 100$.

Opción 6: usar el índice cantidad. Costo aproximado $T_R / 2 = 500$.

Opciones 7 y 8 : costo $T_R = 1000$ pero no son aplicables en el ejemplo.

El menor costo lo da la opción 3, donde obtenemos aproximadamente 20 tuplas que cumplen las C_i ; de ellas examinamos cuales cumplen que la cantidad sea por lo menos cinco, además asumo que se encuentran en bloques diferentes.

3. Calculando el producto cartesiano.

Hay una variedad de estrategias disponibles y el optimizador requiere estimar la de menor costo para una situación dada. Generalmente el producto y la reunión natural son costosos comparados con operaciones sobre una sola relación, tal como la selección y la proyección. Luego, siempre que sea posible debe minimizarse su costo.

Tendremos en cuenta dos nuevos parámetros:

Sea U : El *número de bloques necesarios para almacenar el resultado de lo calculado*. Si el resultado debe ser escrito a disco, U bloques accesados serán necesarios para almacenar la salida. Este valor domina frecuentemente al costo total en productos y reuniones.

Sea M : El *número de bloques que pueden acomodarse en memoria principal a la vez.* Veremos que un valor pequeño para M obliga al mismo bloque a ser leido varias veces, incrementando el costo de la operación más allá de lo necesario para leer los argumentos y escribir los resultados.

Suponga que necesitamos calcular R x S. Asumimos R y S almacenadas independientemente pero compactas, tal que para ser leídas necesitamos B_R y B_S accesos. Si este no es el caso, es usualmente más eficiente leer las tuplas de la relación que no están compactas y hacer copias de ellas, lo que implica un costo adicional de T_R

accesos para leer a R, y de B_R accesos para escribir una copia. (se procede de una manera similar si es S).

El algoritmo consiste de un ciclo doble :

> ***FOR*** *cada tupla* μ *en R* ***DO***
>
> > ***FOR*** *cada tupla* ν *en S* ***DO***
> >
> > > *sale la tupla* $\mu\nu$ *(11.4)*

Podría por supuesto invertirse el rol de los ciclos.

Si una relación se ajusta en memoria principal.

Asumamos $B_S < B_R$. (el caso contrario se maneja simétricamente). Si $M > B_S$. ; entonces S se ajusta en memoria principal. Podemos leer a toda S con B_S accesos y luego leer cada bloque de R y descartarlo cuando leamos el siguiente. Para cada bloque de R ejecutamos el ciclo externo, sin costos adicionales de acceso para el ciclo interno,pues todas las tuplas están en memoria.

Costo para leer la entrada = $B_S + B_R$..

Costo del producto = Costo de la entrada + U bloques para la salida.

Podemos estimar U en terminos de R y S:

Sea L_R = número de bytes para cada tupla de R. Sea L_S = número de bytes para cada tupla de S. Sea b = byte por bloque. El resultado R x S tendrá tuplas que son la concatenación de tuplas de R con tuplas de S.

$$L_{R \times S} \cong L_R + L_S \text{ bytes.}$$

Resultan $T_R \,.\, T_S$ tuplas de salida; luego el número de bloques necesarios para U :

$$U = \frac{T_R \,.\, T_S \;(L_R + L_S)}{b} \qquad (11.5)$$

U es aproximado, pues cada bloque de tamaño b requiere espacio para información de control y para área libre que, entre otros factores, minimice el encadenamiento.

Note que $B_R \cong (T_R \,.\, L_R) / b$ y $B_S \cong (T_S \,.\, L_S) / b$ luego.

$U = (T_R \,.\, T_S \;\; L_R) / b \; + \; (T_R \,.\, T_S \;\; L_S) / b \;\; \Rightarrow$

Costo de salida : $U = B_R \,.\, T_S + T_R \,.\, B_S$. *(11.6)*

El costo total será : $B_R\ (T_S + 1) + B_S\ (T_R + 1)$.. *(11.7.)*

Ejemplo 11.5. : Suponga R = 5000 tuplas acomodadas en 500 bloques y S = 1000 tuplas acomodadas en 100 bloques. Sea M = 101.

Podemos leer toda S en memoria y todavía tener un bloque disponible para R.

Costo : 500 x 1001 + 100 x 5001 = 1.000.600

Lo que tiene sentido, pues resultan cerca de 5 millones de tuplas en la respuesta, cada tupla resultante es la concatenación de una tupla de R con una tupla de S e individualmente ocupan una decima parte de un bloque, entonces puedo acomodar cinco de las resultantes por bloque.

Productos de relaciones que no caben en memoria principal.

Se hace necesario mover varias veces dentro y fuera de memoria principal a algunos de los bloques. Asumamos $B_S < B_R$, pero $M < B_S$. Un desempeño casi óptimo se logra dividiendo S en segmentos de M-1 boques cada uno; Leemos un segmento de S y en el bloque restante de memoria principal, leemos uno por uno los bloques de R. Obtenemos el producto de cada tupla en el segmento de S en turno con cada tupla de R. Repetimos este proceso para cada segmento, hasta lograr el producto completo.

> ***FOR*** *cada segmento S de la relación S* ***DO***
>
> ***FOR*** *cada tupla* ν *en segmento S* ***DO***
>
> ***FOR*** *cada bloque B de la relación R* ***DO***
>
> ***FOR*** *cada tupla* μ *del bloque B* ***DO***
>
> *sale la tupla* $\mu\nu$

Fig. 11.2 *Algoritmo de un producto para grandes relaciones.*

Leo cada bloque de S una vez y cada bloque de R una vez por segmento. Si el número de segmentos es aproximadamente: $\frac{B_S}{M-1}$

Costo de entrada : $B_R\left(\frac{B_S}{M-1}\right) + B_S$ *(11.8)*

costo de Salida : $B_R T_S + T_R B_S$

Costo Total : $$B_R(T_S + \frac{B_S}{M-1}) + B_S(T_R + 1) \qquad (11.9).$$

Note que *11.9* es una generalización de *11.7*; reemplazando :

Ts + 1 por $T_S + \frac{Bs}{M-1}$; es decir, Ts + el número de segmentos de S.

Ejemplo 11.6

El mismo caso del ejemplo anterior , pero M = 11; R = 5000 tuplas S = 1000 tuplas. Br = 500 bloques. Bs = 100 bloques.

Divido S entre Bs / (M -1) segmentos. = 100 / 10 = 10 segmentos.

Costo : 500 x (1000 + 100/10) + 100 x 5001 = 1.005.100.

Comparado con el caso anterior, U continúa siendo el factor de más peso. Aunque el costo de entrada se ha incrementado. Lo que es más significativo en los join, pues en ellos la salida es generalmente más pequeña que para el producto, pero si no somos cuidadosos, el costo de entrada puede llegar a ser tan grande como para el producto.

Un límite más bajo para costos de entrada.

Evidentemente, el costo de salida no puede ser mejorado, porque asumimos que toda tupla de la salida debe ser escrita. Cómo saber cuál es el costo de entrada más bajo ? En realidad, aunque es posible mejorar el costo de *11.8.*, no es posible lograr mejoras significativas. El teorema 11.1. se aplica a cualquier operación, tal como una reunión natural, si actuamos como si fuese un producto; esto es, si cada tupla de una relación debe hallar cada tupla de la otra relación en memoria principal.[5]

TEOREMA 11.1.: Si ejecutamos un producto cartesiano de las relaciones R y S en una memoria que puede guardar M bloques, entonces el número de bloques requeridos para lograr leer R y S es al menos : $\frac{B_R B_S}{(M-1)}$

Prueba:

[5] Como la reunión natural requiere ser ejecutada de esta manera, el teorema motiva la búsqueda por un método más económico de calcular una reunión.

Para obtener una salida $\mu\nu$ es necesario que estén simultáneamente en memoria principal una tupla μ de R y una tupla ν de S. Para cualquier bloque β de R en memoria principal, el provecho sacado de este evento, *para obtener tuplas resultantes,* no puede ser mayor que el número de tuplas que contenga dicho bloque, multiplicado por el número de tuplas de S que en ese momento estén en memoria principal; este número no será mayor que

$$(M -1) (Ts/ Bs) (Tr / Br). \quad (11.10)$$

Idem si leo un bloque de S.

Para calcular el producto completo, debo obtener un número de tuplas tan grande como las que se requieren para la salida , o sea TrTs. Sea *A el número de bloques que requerimos usar como entrada* mientras calculamos R X S . Entonces, $T_S T_R \le A(M-1)\frac{T_S}{B_S}\frac{T_R}{B_R}$,

pues es posible que tengamos que leer varias veces el mismo bloque. Dado lo anterior podemos obtener : $A \ge \frac{B_S B_R}{M-1}$ Número de bloques usados como entrada.

Note que el límite superior en costo de entrada dado por *11.8* y el límite inferior dado por el teorema *11.1.* difieren solo en el término aditivo B_s.

Estimando el costo de salida para los join.

Vamos a considerar los algoritmos para calcular la reunión natural de dos relaciones. Si estamos interesados en los equijoin, usamos la misma técnica renombrando al atributo común en una de las relaciones, convirtiendola en una reunión natural[6].

Si deseamos un θ join, donde θ no es la igualdad; por lo general, no es posible superar el algoritmo para el producto de grandes relaciones que no caben en memoria; aunque ahorraremos en el tamaño de la salida si muchas tuplas del producto no cumplen la condición del join.

Un modelo estadístico para las relaciones.

[6] La reunión natural elimina una copia de cada par de atributos de enlace, mientras que el equijoin las retiene. Cuando estimemos el tamaño de las tuplas resultantes asumiremos, por conveniencia, la presencia de ambos atributos; por ello nuestro estimativo será levemente más alto que el de la reunión natural

Sean AB(a,b) y BC(b,c) dos relaciones, un factor crítico es cuán grande será el resultado. Vimos anteriormente que la decisión de ejecutar una selección antes que un join, depende del tamaño del resultado del join; este tamaño a su vez, depende de cuantas tuplas de BC hacen join con una tupla dada *ab* de la relación AB. No existe una respuesta general para esta pregunta. Nos propondremos un modelo razonable para una relación "típica" y veremos que nos dice dicho modelo.

En primer lugar, *asumamos que cada atributo de la relación tiene un dominio finito asociado*, el conjunto de valores "probables" de aparecer allí. Este dominio puede ser tomado del conjunto de elementos "activos" del dominio potencialmente infinito. Eje: la relación CLIENTE en el atributo nombre de la B.de.D. YVCB, en el momento de la reunión natural, tiene como dominio el conjunto de clientes de YVCB. Asumimos que el conjunto de tuplas potenciales de una relación dada R, es el producto cartesiano de los dominios de todos sus atributos. Si R tiene T_R tuplas, y este producto tiene *n* tuplas, asumimos que el valor corriente de R es un conjunto T_R aleatorio escogido dentro de las *n* posibles tuplas; así, cada tupla potencial tiene una probabilidad $T_R\ / n$ de ser escogida.

Finalmente asumamos que cuando tomamos la reunión natural de dos relaciones R y S , *los atributos del mismo nombre "significan" lo mismo* y por ello seleccionamos sus valores del mismo dominio. Decisión muy sutil, especialmente en el caso de la reunión natural que se origina de un equijoin renombrado, cuyos atributos de igualdad realmente no tengan nada que ver el uno con el otro. Por ejemplo: hallar dos clientes de YVCB cuyo balance iguala la cantidad de algún item ordenado por algún cliente.

Sinembargo, si el o los atributos del join se refieren a la misma noción, nuestra premisa tiene una alta probabilidad de ser real. Además, como veremos, la igualdad de dominios no es necesaria, solo es necesario contener un dominio en otro.

El siguiente ejemplo nos puede sugerir que esperar en tales situaciones:

<u>*Eje 11.7*</u> : Sean las relaciones CLIENTE, ORDEN e ITEM.

Cliente(nombre, dirección,balance);

Orden(#Orden,fecha,cliente)

Item(#orden,producto,cantidad)

Los atributos #*orden* de las relaciones ORDEN e ITEM, evidentemente significan la misma cosa en ambas relaciones. Esperamos que cualquier #*orden* que aparezca en una relación

aparezca en la otra. Si solo aparece en ORDEN sería una orden de nada, lo que no es erróneo, pero tampoco es el caso típico. Un #*orden* que aparezca solo en ITEM será probablemente un error.

Una situación menos obvia se da con el equijoin :

$$\text{CLIENTE} \underset{\text{nombre=cliente}}{\otimes} \text{ORDEN} \qquad (11.11.)$$

Evidentemente, *nombre* y *cliente* se refieren al mismo dominio. Dado que *nombre* es la clave de CLIENTE, esperamos que todos los clientes corrientes aparezcan en el dominio de *nombre*. Quizás todos o la mayoría de los clientes han pedido ordenes en algún momento, tal que podemos asumir idénticos los dominios de *nombre* y *cliente*; sinembargo, también es posible que solo una pequeña fracción de clientes hallan solicitado ordenes, tal que el dominio del *cliente* sea mucho más pequeño que el de *nombre*.

Afortunadamente, bajo nuestras premisas, no importa cual de ambos casos ocurre. El tamaño estimado del join *11.11* no depende de que fracción de los clientes colocó en el momento ordenes de pedido; la razón intuitiva es que cada tupla de ORDEN tendrá un valor para *cliente* que es igualmente probable de aparecer en el atributo *nombre* de cualquier tupla de CLIENTE.

En nuestro ejemplo: nombre es la clave de CLIENTE, entonces cada cliente debe aparecer exactamente una vez en esta relación. Así, el tamaño del join será el mismo del tamaño de la relación ORDEN, sin importar que fracción de clientes ha colocado ordenes[7].

Implicaciones del modelo respecto al tamaño de salida del join.

Relación AB (T_{AB} número de tuplas.)

Atributo A	Atributo B
D_A Dominio	E_B Dominio
I_A Tamaño	J_B Tamaño

[7] De una manera más general, si para los valores de *cliente* en ORDEN su promedio de aparición en el componente *nombre* de CLIENTE, es *k* ; entonces el tamaño de salida esperado debería ser *k* veces el tamaño de ORDEN, independiente de la fracción de valores de *nombre* que deberían ser valores de CLIENTE.

a Valor	b Valor

Relación BC (T_{BC} número de tuplas) $E_B \subseteq D_B$.

Atributo B	Atributo C
D_B Dominio	D_C Dominio
I_B Tamaño	I_C Tamaño
b Valor	c Valor

Generalizando la observación del ejemplo *11.7* respecto al equijoin *11.1* , consideremos la reunión natural AB ⊗ BC, donde existe una dependencia de inclusión en los atributos de B; específicamente, todo *valor-b* de la relación AB también aparece en el atributo B de por lo menos una tupla de BC, pero no necesariamente visceversa.

- Defino D_A, D_B , D_C los dominios de A en AB, B en BC y C en BC respectivamente.
- Uso E_B para el dominio de B en AB; note que $E_B \subseteq D_B$.
- Uso I_A, I_B, e I_C para el tamaño de la imagen de los atributos A,B,C, es decir el tamaño de D_A, D_B , D_C respectivamente.
- Uso J_B para el tamaño de la imagen de E_B .
- Sea T_{AB} el número de tuplas de AB y T_{BC} las tuplas de BC.

Para calcular el tamaño de AB ⊗ BC: Considere una tupla arbitraria *abc* en D_A x D_B x D_C donde el número posible de tales tuplas es $I_A I_B I_C$. *Necesitamos solo calcular la probabilidad que abc esté en el join*, para ello *ab* debe estar en AB y *bc* en BC.

Cuál es la probabilidad que una tupla aleatoria *bc* esté en BC?

Dadas nuestras premisas, esa probabilidad es : $\dfrac{T_{BC}}{I_B\, I_C}$

Ahora debemos hallar la probabilidad que *ab* esté en AB.

a. Si *b* es un valor que está en D_B pero no en E_B , entonces la probabilidad será cero; este caso ocurre una fracción $(I_B - J_B) / I_B$ de veces.

b. Si *b* es un valor tal que, $b \in E_B$ y $b \in D_B$, entonces la probabilidad que *ab* esté en AB es: $T_{AB} / (I_A J_B)$, lo que ocurre cada J_B / I_B de veces.

Luego, la probabilidad que una tupla aleatoria *ab*, escogida desde los dominios D_A x D_B , esté en AB es:

$$(\frac{I_B - J_B}{I_B} \times 0) + (\frac{J_B}{I_B} \times \frac{T_{AB}}{I_A J_B}) \qquad (11.12)$$

Simplificando tenemos : $\frac{T_{AB}}{I_A I_B}$.

Note que *11.12* no depende de J_B . Esta observación es la formalización del comentario intuitivo del ejemplo *11.7*, que el tamaño del join no depende de cuantos clientes han solicitado ordenes en el momento de su ejecución.

Multiplicando la probabilidad de encontrar *ab* en AB, por la probabilidad de hallar *bc* en BC, calculamos la probabilidad de *abc* en el join AB $\otimes$ BC :

$$\frac{T_{AB} T_{BC}}{I_A I^2{}_B I_C}$$

Finalmente, debemos multiplicar esta probabilidad por el número de tuplas posibles de hacer parte del joins I_A I_B I_C , para obtener:

tamaño estimado de AB $\otimes$ BC = **T_{AB} T_{BC} / I_B .**

Esto es, el tamaño de salida es el producto del tamaño de las dos relaciones, dividido por el tamaño de la imagen del atributo común; lo anterior lo podemos generalizar en el siguiente teorema.

TEOREMA 11.2. Sean $R(A_1 \ldots A_J, B_1 \ldots B_K)$ y $S(B_1 \ldots B_K, C_1 \ldots C_M)$ dos relaciones; supongamos que para cada i = 1,2,...,k el dominio de Bi en R es un subconjunto del dominio de Bi en S, o visceversa. Asumamos que cada tupla en el producto de los dominios de todos los atributos de R, es igualmente probable de estar en R y similarmente para la relación S. Entonces el tamaño esperado de la reunión natural R $\otimes$ S es :

$$\frac{T_R T_S}{I_{B_1} I_{B_2} \cdots I_{B_k}} \qquad (11.13)$$

Donde I_{B_i} es el tamaño del dominio de Bi en R o S y donde uno es un superconjunto del otro.

Podemos ahora estimar el número esperado de bloques ocupados por la salida del join. Sea I el denominador de *11.13.*, esto es, el producto del tamaño del dominio para cada

atributo compartido por R y S. Entonces $1/I$ parte de las tuplas del producto R X S aparecerán en el join, tal que el tamaño de salida es *11.6* dividido por I, o

$$\frac{B_R T_S + T_R B_S}{I} \qquad (11.14)$$

Sin embargo, recordemos que si tomamos una reunión natural en lugar de un equijoin, copias duplicadas del atributo del join son borradas de todas las tuplas. Luego, el estimado *11.14.* es ligeramente más alto, pues las tuplas no son tan grandes en la salida del join como en el producto ó en el equijoin. Para nuestro trabajo ignoraremos tal situación.

Ejemplo 11.8.

Consideremos las relaciones R y S del ejemplo 11.5. e imaginemos que R es R(A,B,C) mientras que S es S(B,C,D).Suponga que se mantiene la dependencia de inclusión $R.B \subseteq S.B$ y $S.C \subseteq R.C$. Finalmente, suponga que I_B, el tamaño del dominio para B en S, es 50, e I_C, el tamaño del dominio para C en R, es 40. Los demás parámetros para R y S son los mismos del ejemplo 11.5. Entonces el tamaño del join $R \otimes S$ está dado por:

$$U = \frac{B_R T_S + B_S T_R}{I_B I_C} = \frac{500 \times 1000 + 100 \times 5000}{50 \times 40} = 500$$

Esta conclusión, que solo una en 2000 pares de tuplas desde R y S se enlazan exitosamente, es razonable bajo nuestro modelo de relación. Un par de tuplas tienen una opción en 50 de coincidir en B y una opción de 1 en 40 de coincidir en C. Sin embargo, debemos ser muy cuidadosos si B y C no son independientes. Como un ejemplo extremo supongamos que existe una dependencia funcional $B \rightarrow C$ que se cumple en ambas relaciones R y S [8]. Luego, tuplas que coincidan en B con seguridad coinciden en C, y la probabilidad que dos tuplas provenientes de R y S coincidan es 1/50, no 1/2000. Puesto de otra manera, podríamos esperar un join de tamaño 20.000 bloques, en lugar de los 500 bloques calculados anteriormente.

La situación anterior puede ser acomodada en nuestro modelo, si definimos que B y C puedan ser tratados como un solo atributo, llamado E. La dependencia de inclusión $R.B \subseteq S.B$, al igual que la aseveración de la dependencia funcional $B \rightarrow C$, que se mantiene en

[8] Esto es, tuplas que coincidan en B deben coincidir en C, cuando las tuplas vengan desde R, desde S ó desde ambas relaciones.

la combinación de R y S, implican que R.E $\subseteq$ S.E. El tamaño del dominio de E en S es evidentemente 50, porque cada uno de los 50 valores en el dominio de B está asociado con únicamente un valor de C [9]. Así, podemos manejar al join R$\otimes$ S como AE $\otimes$ ED, y calcular el tamaño de salida como 1.000.000 dividido por I_E, o 20.000.

11.5. Métodos para calcular join.

Vamos a considerar y evaluar el costo de varios métodos para computar join, basados en el modelo definido en la sección previa. Los métodos considerados incluyen: la obvia opción de *selección-en-un-producto*, una técnica llamada *join-clasificado* (sort-join) donde ambas relaciones están clasificadas por los atributos del join, y métodos que toman ventaja de varios tipos de índices.

Las fórmulas para el tiempo esperado de ejecución de estos algoritmos son frecuentemente complicadas. Asi, al final de la sección, resumiremos las fórmulas y listaremos el término o los terminos dominantes para cada método.

Cómputo del join por selección en un producto.

El modo obvio para computar R $\otimes$ S es calcular R X S, usando la *fig. 11.2.* (como antes asumimos que S es la relación más pequeña) . Sin embargo, en lugar de emitir toda tupla *uv*, donde *u* está en R y *v* en S, emitimos solo esa tuplas que coinciden en los atributos comunes de R y S. El costo de este método es el costo de entrada , dado por la fórmula *11.8.*, más el costo de salida, dado por la fórmula *11.14*.

Ejemplo 11.9.

Suponga que tenemos al join del ejemplo *11.8.* con $I = I_B I_C = 2000$ y M= 101, como en el ejemplo *11.5*. Entonces el valor de la fórmula *11.8* es : 500 x (100/100) + 100 = 600

El valor de U, como vimos en el ejemplo *11.8*, es 500, tal que el costo total es 600 + 500 = 1100.

Si M = 11, como en el ejemplo *11.6.*, U no cambia, pero el costo de entrada se eleva a :

500 x (100/10) + 100 = 5100.

Así, con solo 11 bloques disponibles de memoria principal, el costo del join es 5600. Note que en ambos casos, el costo de entrada es significativo, a diferencia de los ejemplos

[9] El hecho que varios valores de B puedan aparecer con un valor de C no afecta el tamaño del dominio de E.

11.5 y *11.6.*, donde la operación era el producto cartesiano, y el costo de salida dominaba al de entrada.

El lector puede chequear la siguiente fórmula, la suma de las fórmulas *11.8* y *11.14* , que es el *costo total* para llevar a cabo un join por el método de *selección-en-un-producto*.

$$B_R\left(\frac{T_S}{I} + \frac{B_S}{M-1}\right) + B_S\left(\frac{T_R}{I} + 1\right) \qquad (11.15)$$

Join-clasificado (Sort-merge Join).

El teorema *11.1* nos dice que no podemos llevar a cabo mejoras más significativas que *11.15.*, si ejecutamos un join de tal manera, que cada uno de los bloques de una relación encuentre a cada uno de los bloques de la otra en memoria principal. Sin embargo, existen varias alternativas para preprocesar una relación (ó tomar ventaja de los índices existentes) y asi evitar la mezcla no selectiva de dos relaciones. Consideraremos uno de esos casos, el join clasificado.

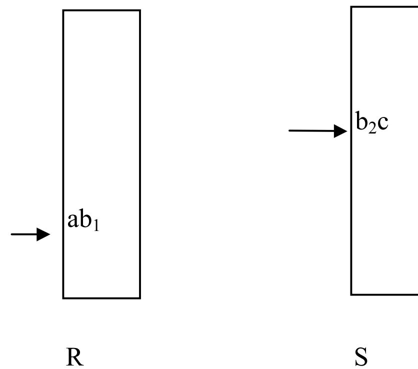


Fig. 11.3 recorriendo dos relaciones clasificadas.

a_1b_1	b_1c_1
a_2b_1	b_1c_2
a_3b_2	b_3c_3
a_4b_3	
R	S

Fig. 11.4. Ejemplo de relaciones clasificadas.

En lo que sigue, asumiremos que nuestras dos relaciones para join son R(A,B) y S(B,C). La idea se generaliza fácilmente al caso donde hay varios atributos en común y/o varios atributos pertenecientes solo a una de las relaciones. Comenzamos clasificando a R por

su atributo B, procedemos de la misma manera con S. Cuando usemos un par de cursores para que recorran los atributos de R y S, hallaremos primero los *valores-b* más bajos como sugiere la fig. *11.3.* Suponga que buscamos dos tuplas ab_1 desde R y $b_2\ c$ desde S. Si entre b_1 y b_2 alguno es más pequeño que el otro , avanzamos el cursor del más pequeño hacia abajo y dejamos fijo al otro. En este caso no es posible que el más pequeño coincida en algún momento con un valor de la otra relación. Si $b1 = b2$, entonces hallamos todas las tuplas siguientes de R y S con el mismo *valor-b*. Apareamos cada una de las tuplas de R con cada una de las tuplas de S y emitimos las tuplas resultantes (borrando una copia del *valor-b* común). Luego, movemos el cursor de las dos relaciones justo después de la última tupla con el *valor-b* en proceso.

Ejemplo 11.10.

Suponga que R y S consisten de la lista de tuplas de la Fig. *11.4*. Ambas relaciones ya han sido clasificadas por el *valor-b*. Comenzamos con a_1b_1 para R y b_1c_1 para S. Como los *valores-b* coinciden, hallamos todas las tuplas de R con este *valor-b*, esto es, { a_1b_1,a_2b_1}, y todas las tuplas de S con este mismo *valor-b*, { b_1c_1,b_1c_2}. Note que las tuplas deseadas deben estar en posiciones consecutivas en los cursores de las respectivas relaciones. Tomamos el producto y omitimos una copia de b_1 para cada tupla resultante, conseguimos entonces 4 tuplas, { $a_1b_1c_1$, $a_1b_1c_2$, $a_2b_1c_1$, $a_2b_1c_2$}, que pertenecen a $R \otimes S$.
A continuación, avanzamos el cursor de R a a_3b_2 y el cursor de S a b_3c_3. Como b_2, presumiblemente, es menor en orden de clasificación que b_3, sabemos que no hay tuplas de S con *valor-b* b_2. Si las hubiese, deberían preceder a b_3. Así, avanzamos el cursor de R a a_4b_3 y dejamos fijo al cursor de S.
Como resultado, tenemos ambos cursores en el *valor-b* b_3, y en cada uno de los casos, las tuplas recorridas son unicamente las tuplas de las relaciones con ese *valor-b*. Así, simplemente las apareamos para generar la quinta y última tupla del join, {$a_4b_3c_3$}. En este punto hemos recorrido ambos cursores hasta su final. En general, en el recorrido podríamos terminar con un cursor mientras que el otro todavía tenga tuplas por recorrer. .

Clasificación-mezcla Multimodo. (Multiway Merge-Sort).

Dado que ordinariamente las relaciones no están clasificadas, debemos calcular el costo de clasificar una relación. Bajo nuestro modelo de costos, donde contamos los bloques accesados, un método llamado clasificación-mezcla multimodo es una excelente alternativa. El lector interesado en los detalles de este algoritmo puede consultar Aho-Hopcroft y Ullman [1983][10]. Acá solo describiremos la idea general y daremos el análisis de su costo.

El algoritmo opera construyendo *corridas*, que no son otra cosa que secuencias de bloques que contienen una lista de tuplas clasificadas y pertenecientes a una relación R dada. En una serie de pasadas, R es particionada en un conjunto de *corridas,* progresivamente más grandes; hasta el final, cuando solo queda una corrida, la relación completa. Asumiremos que hay M bloques de memoria principal disponibles para guardar bloques de R. Algunos bloques adicionales de memoria son requeridos para cálculos, tal como hallar el más pequeño de un conjunto de M valores, y para la salida clasificada; asumiremos que ya están reservados y que no hacen parte de los M bloques disponibles para almacenar la relación a ser clasificada.

En la primera pasada, leemos los bloques de R a memoria principal, M al tiempo, y clasificamos las tuplas dentro del grupo de M bloques, usando cualquier algoritmo de clasificación interna apropiado, tal como el Quick-Sort (Véase Aho, Hopcroft y Ullman [1983] [8]). Note que no tomamos en cuenta los cálculos en memoria; en realidad, la clasificación de M bloques toma probablemente mucho menos tiempo que su lectura desde almacenamiento secundario. Como resultado, R está ahora particionado en B_R/M *corridas* de M bloques cada una.

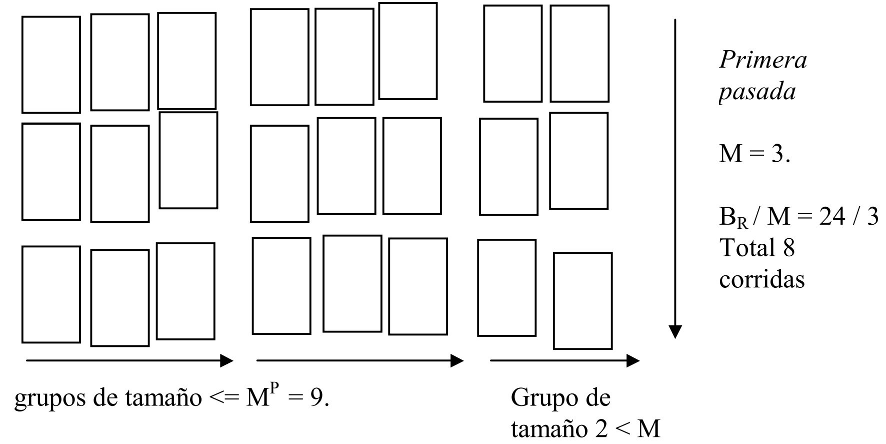
En la pasada *p*, con *p* > 1, arrancamos con *corridas* de M^{p-1} bloques cada una , y las reunimos en grupos de M *corridas*. Así , cada nuevo grupo consiste de M^p bloques. El último grupo puede tener un poco menos de M *corridas*, y por ello su longitud será menor que M^p bloques. Luego mezclamos cada grupo con el proceso sugerido en la fig. *11.5. b*

Para ejecutar la mezcla mantenemos un cursor para cada una de las M *corridas* en un grupo. Al comienzo, cada cursor está apuntando al elemento más pequeño al inicio de la *corrida*. Repetidamente seleccionamos al elemento más pequeño entre los señalados por

---

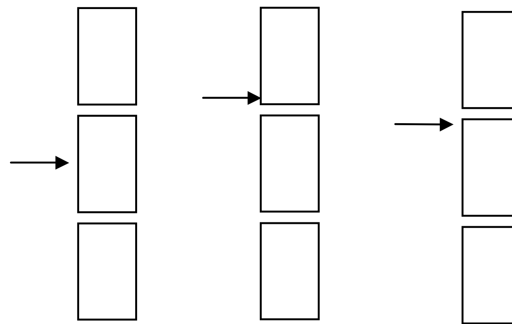
[10] Aho, A. V. , J.E. Hopcroft, and J.D. Ullman [1983]. *Data structures and Algorithms*. Addison-Wesley,Reading Mass.

los M cursores y el cursor cuyo elemento sea escogido es avanzado al siguiente elemento de esa corrida. El elemento seleccionado es adicionado al bloque de salida corriente.



Segunda pasada: Resultado 3 grupos de corridas.

Fig. 11.5.a



Mezcla de un grupo.

Fig. 11.5.b. Mezcla Multimodo. con M=3 y p=2

Necesitamos mantener en memoria principal solo un bloque de cada *corrida*. Cuando el cursor para una *corrida* termina de recorrer tal bloque, lo reemplazamos por el siguiente bloque de la misma *corrida*. Igualmente, necesitamos mantener en memoria principal, solo un bloque para la salida (Que no hemos contabilizado entre los M reservados para la relación de entrada). Cuando un bloque sea llenado, lo movemos a almacenamiento secundario, por lo que puede ser llenado de nuevo con las siguientes tuplas en el orden de clasificación.

Después de $\lceil Log_M B_R \rceil$ pasadas, las corridas producidas son por lo menos de longitud B_R. Así, la relación completa R es una corrida. Esto es, está clasificada.[11]

Analisis del Join-clasificado. (sort-join).

Estimaremos el costo de ejecutar un join clasificado entre las relaciones R y S. El proceso final del join, sugerido por la fig. *11.3.*, tiene como costo de entrada $B_R + B_S$, dado que necesitamos leer solo una vez cada bloque de las relaciónes clasificadas. El costo de salida es determinado obviamente por *11.14.*

Debemos ahora estimar el costo de clasificar las relaciones. Una pasada de la clasificación-mezcla multimodo lee cada bloque de la relación una vez, pues se lee cada corrida solo una vez; también escribe nuevos bloques, que almacenan mezcladas las corridas de cada grupo de la relación. Así, cada pasada usa 2B accesos a bloques, si la relación requiere de B bloques para almacenarse de una manera compacta. Como el número de pasadas es $\lceil Log_M B \rceil$, el costo de usar el algoritmo en cuestión en una relación de B bloques es $2B x log_M B$[12]. Así, el costo total del join es :

$$2B_R \log_M B_R + 2B_S \log_M B_S + B_R + B_S + \frac{B_R T_S + T_R B_S}{I} \qquad 11.16.$$

Los dos primeros términos son el costo de clasificar las relaciones, los siguientes dos son el costo de recorrer las relaciones clasificadas para aparear las tuplas coincidentes y el último término es el costo de la salida.

La fórmula *11.16* es válida siempre que no existan *valores-b* de los atributos de enlace, para los que el número de tuplas que comparten tales valores en las dos relaciones excedan al número de ellas que se pueden acomodar al mismo tiempo en memoria principal. Si lo anterior no se cumple, entonces el recorrido de las relaciones sugerido por la fig. *11.3.* no podría llevarse a cabo sin la relectura de algunos de los bloques que contengan tales *valores-b.*

Ejemplo 11.11.

---

[11] Si $B_R / M^P = 1$, la relación está clasificada.

[12] Omitiremos la función que convierte la fórmula al menor entero por encima del valor obtenido, pero el lector debe tener presente tal hecho.

Consideremos las relaciones del ejemplo *11.9.*, con I = 2000 y M = 101. Como $B_R = 500$, entonces $\log_M B_R = 2$. Como $B_S = 100$, tenemos $\log_M B_S = 1$. Recordemos $T_R = 5000$ y $T_S = 1000$. Así, *11.16*. es :

$$2 \times 500 \times 2 + 2 \times 100 \times 1 + 500 + 100 + (1.000.000/2000) = 3300.$$

Esta cifra es pobre en desempeño comparada con el estimado 1100 para el join directo dado en el ejemplo11.9..

Sin embargo, considere el caso donde M=11. Luego estime $\log_M B_R = 3$ y $\log_M B_S = 2$. Nuestro costo estimado es entonces:

$$2 \times 500 \times 3 + 2 \times 100 \times 2 + 500 + 100 + (1.000.000/2000) = 4500.$$

Esta cifra es menor que el estimado de 5600 para el join directo.

En general, si el tamaño de B_R y B_S, bloques ocupados por las relaciones, es grande, con M constante y $B_R >= B_S$, el costo de entrada para la selección en un producto crecerá tanto como $\frac{B_R B_S}{M}$. Mientras que el costo de entrada para el join-clasificado crecerá tanto como $B_R \log_M B_R$, cifra significativamente menor.

Join usando un índice.

Consideremos de nuevo el join R(A,B) ⊗ S(B,C), pero ahora suponga que S tiene un índice cluster en B. Asumamos que R tiene un almacenamiento compacto. Entonces podemos ejecutar el join tomando cada bloque de R, y para cada tupla de R, usamos el índice para hallar las tuplas de S que coinciden. Esto es, hacemos:

FOR cada bloque β de R DO

FOR cada tupla ab en el bloque β DO (11.17)

join ab con cada tupla en $\sigma_{B=b}(S)$

Sea I el tamaño de la imagen de B en S y asumamos que la dependencia de inclusión $R.B \subseteq S.B.$ se conserva. Esto es, todo *valor-b* que aparezca en R aparece en S. Entonces, el ciclo interno de *11.7* es llevado a cabo T_R veces y en cada iteración hemos recuperado, vía el índice cluster, cerca de B_S / I bloques. En el ciclo externo leemos cada bloque de R una vez, implicando un costo adicional de B_R bloques accesados. Así, el costo de entrada de *11.7* es

$$B_R + \frac{T_R B_S}{I} \qquad \textit{11.18.}$$

el costo total, la suma de *11.14* y *11.18*, es :

$$B_R(1+\frac{T_S}{I}) + \frac{2T_R B_S}{I} \qquad \textit{11.19.}$$

Debemos ser cuidadosos al interpretar *11.8.* y las fórmulas que se derivan de ella; pues hemos asumido tácitamente que I < = B_S. Si no, Entonces B_S/I no es un estimado adecuado del número de bloques recuperados en el cuerpo de *11.7.*, pues el número no puede ser menor que 1. Si I >= B_S, entonces recuperamos cerca de un bloque de S por cada tupla de R, y *11.18* debe ser reemplazada por $B_R + T_R$. Una fórmula que generaliza a 11.19 para el caso donde I puede ser mayor que ó menor que B_S es

$$B_R(1+\frac{T_S}{I}) + \frac{T_R B_S}{I} + T_R \times \max(1, \frac{B_S}{I}) \qquad \textit{(11.20)}$$

Algunos otros casos de Join con índices.

Podemos derivar varias fórmulas similares para situaciones relacionadas. Si la relación R no está compacta, entonces podemos tener acceso a T_R bloques, en lugar de B_R , para leer a R. En este caso, el costo de entrada debería ser T_R (1+B_S / I), en lugar de *11.18*; por supuesto el costo de salida no cambia. Si R está compacta, pero el índice de S es no cluster, entonces cada iteración del ciclo interno de *11.17*. debe recuperar cerca de T_S / I bloques, y el costo de entrada debería ser $B_R + T_R T_S / I$. Si el índice es no cluster y R no está compacta, entonces el costo de entrada es T_R(1+ T_S / I).[13] Finalmente, podríamos derivar fórmulas similares , con R y S intercambiadas, si hay un índice de R en B que podamos usar.

En todos los anteriores casos, hemos asumido que R.B $\subseteq$ S.B. Si la dependencia de inclusión se presenta de otra manera, entonces *11.8.* será todavía un buen estimado del costo de entrada, asumiendo que el tamaño de la imagen de S.B no es mayor que B_S , pero debemos interpretar I como $I_{R.B.}$, en lugar de $I_{S.B.}$, aunque el índice todavía sea de S. La fórmula apropiada para el caso donde S.B. $\subseteq$ R.B, permitiendo $I_{S.B.}$ < = B_S e $I_{S.B.}$ > B_S, es:

[13] Note que se conserva I < = T_S ,asi que no es necesario considerar la posibilidad que T_S / I sea un número subestimado de bloques recuperados.

$$B_R(1+\frac{T_S}{J})+\frac{T_R}{J}(B_S+\max(B_S,I)) \qquad (11.21)$$

donde $J= I_{.R.B.}$ e $I = I_{S.B.}$

Ejemplo 11.12.

Tomemos nuestro ejemplo de las corridas, con B_R= 500, T_R=5000, B_S=100 y T_S= 1000. Suponga que R.B $\subseteq$ S.B, y que I = $I_{S.B.}$ = 50. Finalmente, asumamos que hay un índice cluster en el atributo B de S. Como I < = B_S, aplicamos *11.19*, cuyos valores son :

$$500(1+\frac{1000}{50})+\frac{2\times 5000\times 100}{50}=30.500$$

Suponga la misma situación, pero ahora S.B. $\subseteq$ R.B., y J = $I_{R.B.}$ = 200. Además, suponga que el índice de S.B. es no cluster. Entonces el costo de entrada es:

$B_R + T_R T_S / J$ = 500 + (5000 x 1000) / 200 = 25.500

mientras que el costo de salida, dado por 11.14 con J en lugar de I, permanece constante, esto es 5000. Así, el costo total se incrementa desde 8000, si tuviese índice cluster, a 30.500.

Join usando dos índices.

Podemos aún mejorar el costo si hay un índice en B para ambas relaciones. Suponga primero que ambos son índices cluster. Podemos hallar a un conjunto de *valores-b* examinando uno de los índices, usamos para ello al índice con el tamaño de imagen más pequeña; asumamos que es el índice de B en S, y que sea I = $I_{B.S.}$ Como accesamos al índice de B en S, hallaremos todos los I valores del B en turno. Note que es innecesario hallar al conjunto de *valores-b* en R, porque si están ausentes en S de todas maneras no aparecerán en el resultado final del join.

Una vez tengamos al conjunto de *valores-b* , podemos recorrerlos, recuperando las tuplas relevantes de S y R. Formalmente:

FOR cada valor-b b DO

join las tuplas de $\sigma_{B=b}(R)$ *con* $\sigma_{B=b}(S)$ *(11.22)*

Análisis del join con dos índices.

Estimemos el costo de *11.22.*, asumiendo $S.B \subseteq R.B$. El cuerpo de *11.22* es ejecutado I veces. El número promedio de bloques recuperados en una iteración es a lo sumo el MAX(1, B_R / J), donde J es $I_{R.B.}$; Podría ser menor si muchos valores en el dominio de B en S no están en el dominio de B en R. El número de bloques de S recuperados es cerca del MAX(1, B_S / I). Asi, el costo de entrada para este método de join es a lo sumo

$$I \times (\max(1, \frac{B_R}{J}) + \max(1, \frac{B_S}{I}))$$

Cuando calculamos el costo de salida con *11.1.4,* debemos recordar que el tamaño de la imagen es la más grande entre las dos imagenes de los atributos usados para el join. Asi, I en *11.14.* es J para este caso, y el costo total del join con dos indices es:

$$I \times \max(1, \frac{B_R}{J}) + \max(I, B_S) + \frac{B_R T_S + T_R B_S}{J} \qquad 11.23.$$

Queda como ejercicio las modificaciones necesarias de *11.23*. cuando ambos índices no sean del tipo cluster.

Ejemplo 11.13.

Consideremos el join del ejemplo 11.12. con índices cluster en B para R y S. Asuma que I=50 y J=200. Entonces el valor de 11.23. es:

$$50 \times max(1, \frac{500}{200}) + max(50{,}100) + \frac{500 \times 1000 + 5000 \times 100}{200} = 5225$$

Note que el costo de salida, 5000, es casi el costo completo de este ejemplo.

Creando un índice cluster. (hash join)

Como parece tener considerable ventaja ejecutar los join con índices cluster en cada uno de los atributos involucrados en la condición, consideremos que tan rápido podríamos crear tal índice si no existe. Suponga que tenemos una relación R(A,B), y que sea I el tamaño de la imagen para B, el atributo en el que vamos a crear el índice cluster. Lo que hacemos es crear una tabla aleatoria con cerca de I cubetas, y en cada cubeta, ubicamos copias de todas las tuplas de R cuyo *valor-b* le corresponde al aplicar la función aleatoria. En promedio, tendremos un *valor-b* por cubeta.

Suponga que estamos forzados a usar solo M bloques de memoria principal para guardar el contenido de las cubetas que conformemos. Como en el análisis del join-clasificado, no contamos entre estos M bloques otros requerimientos de memoria, tal como espacio para almacenar el programa y un bloque para lecturas de entrada. Crearemos en varios pasos la tabla aleatoria con I cubetas, análogos a los pasos usados para la clasificación-mezcla multimodo.

Seleccione una función aleatoria que disperse las tuplas en las I cubetas numeradas de 0 a I-1. Si $I > M$, no podemos particionar de una vez a R en todas las I cubetas, porque gastaríamos tantos accesos a bloques, moviendolos varias veces en las cubetas a y desde la memoria, como tuplas de R fuesen ubicadas por la función hash en cada cubeta. Sin embargo, podemos dividir a R en M "supercubetas", cada una representando I/M de las cubetas originales. Separamos un bloque para cada supercubeta en memoria, y cuando esté lleno, lo movemos a memoria secundaria y arrancamos un nuevo bloque para la misma supercubeta. En la primera pasada, usaremos la supercubeta 0 para representar las cubetas 0 hasta (I/M) -1, la supercubeta 1 para representar I/M hasta (2I/M)-1, y asi sucesivamente. Leemos cada bloque de R, aplicamos la función a cada tupla del bloque, y si la tupla pertenece a la cubeta *i* , la colocamos en la supercubeta $\lfloor iM/I \rfloor$.

En general, en el paso p, con $p > 1$, arrancamos con supercubetas donde cada una representa I/M^{p-1} cubetas originales y finalizamos con un número mayor de supercubetas de menor tamaño, cada una representando I/M^{p} cubetas originales. Durante estas pasadas trabajamos en una supercubeta a la vez, partiendola para representar M grupos de cubetas, cada grupo pertenecerá a una de las nuevas supercubetas. Leemos cada vieja supercubeta una vez por pasada, y guardamos M bloques representando las M nuevas supercubetas más pequeñas, escribiendo a disco el bloque cuando se llene. Despues de $\log_M I$[14] pasadas cada nueva supercubeta representa solo una cubeta de las I originales, y el proceso está terminado.

Análisis del join con creación de índices.

[14] $\lceil I / M^{P} \rceil = 1.$

El número de bloques accesados que requiere la técnica anterior para la creación del índice se calcula de la siguiente manera. Cada pasada lee y escribe cerca de B_R bloques, todos los bloques de las supercubetas en que partimos a R son leídos y todas las supercubetas con un tamaño más pequeño son escritas. Así, el costo de las $Log_M I$ pasadas es :

$$2B_R Log_M I \qquad (11.24)$$

La fórmula 11.24. será una subestimación si $I > B_R$. Porque entonces, la pasada o pasadas finales dividirán a R en más cubetas que las necesarias para almacenarla compacta; por lo tanto, estas pasadas usarán más de $2B_R$ accesos a bloques. Un límite superior en el costo, sin asumir $I >= B_R$ es

$$2\max(I, B_R)\log_M I \qquad 11.25$$

Dejamos una fórmula más precisa como ejercicio.

Suponga que deseamos tomar el join $R(A,B) \otimes S(B,C)$, pero no hay índices en B para las relaciones. Hacemos uso de la técnica ya descrita para crear tales índices, pagando para cada creación la cantidad extra calculada en *11.25*.. Asumamos que $I_{R.B.}$ es denotado por J e $I_{S.B.}$ es denotado por I. Tambien asumamos la dependencia de inclusión $R.B \subseteq S.B$, con la consecuencia de $J >= I$. Entonces, el costo de este método es *11.23*. más el costo de crear los índices de R y S. El último costo está dado por *11.25.*, con la apropiada sustitución de las variables. La fórmula para el costo total es:

$$(I + 2J \times \log_M J) \times \max(1, \frac{B_R}{J}) + \max(I, B_S)(1 + 2 \times \log_M I) + \frac{B_R T_S + T_R B_S}{J} \qquad 11.26$$

<u>Ejemplo 11.14.</u>

Consideremos los datos del ejemplo *11.13.* . Con $M = 101$, tenemos $\log_M J=2$ y $\log_M I=1$, asi, el valor de *11.26*. es

$$(50 + 2 \times 200 \times 2)(\max(1, \frac{500}{200})) + (\max(50,100))(1 + 2 \times 1) + \frac{500 \times 1000 \times 5000 \times 100}{200} = 7425$$

Con $M = 11$, conseguimos $Log_M J=3$ y $\log_M I=2$; el costo se incrementa a 8625.

Resumen y comparaciones.

La figura *11.6* es una tabla de los principales métodos de join considerados en esta sección. En adición a las premisas ya descritas explícitamente, asumamos las siguientes premisas y convenciones globales.

1. El modelo de relaciones aleatorias dado en la sección 11.4 se cumple.
2. Las relaciones se almacenan de una manera compacta.
3. El join es R(A,B)⊗S(B,C). R y S tienen T_R y T_S tuplas respectivamente y se acomodan en B_R y B_S bloques.
4. I es $I_{S.B.}$, el tamaño de la imagen de B en S y J es $I_{R.B.}$, el tamaño de la imagen de B en R.

M es aproximadamente el tamaño de la memoria principal, excluyendo el espacio necesario para almacenar los programas, y en algunos casos, un buffer de entrada o salida.

Metodo	**Premisas**	**Costo**	**Término dominante**
Selección-en-un-producto.	$R.B \subseteq S.B$ o $S.B \subseteq R.B$	11.15	$B_R B_S / M$
Join-clasificado	$R.B \subseteq S.B$ $B_R / J + B_S / I <= M$	11.16	$B_R \log B_R$, $B_S \log B_S$ $B_R T_S / I$, $B_S T_R / I$
Indice cluster en S.B	$R.B \subseteq S.B$	11.20	$B_R T_S / I$, $B_S T_R / I$, T_R
Indice cluster en S.B.	$S.B \subseteq R.B$	11.21.	B_R , $B_R T_S / J$, $B_S T_R / J$ $T_R I / J$
Join con dos índices.	$S.B \subseteq R.B$ $B_R / J + B_S / I <= M$	11.23.	I, $B_R T_S / J$, $B_S T_R / J$
Join con creación de los dos índices	$S.B \subseteq R.B$ $B_R / J + B_S / I <= M$	11.26	$J \log_M J$, $B_R \log_M J$, $B_S \log_M I$, $B_R T_S / J$, $B_S T_R / J$

Fig. 11.6 Comparación de métodos de join.

Sabemos tambien que $I <= T_S$ y que $J <= T_R$. Además, si asumimos la dependencia de inclusión $S.B \subseteq R.B$, entonces sabemos que $I <= J$. Si es asumida la inclusión opuesta, entonces $I >= J$. Asumimos estas desigualdades para eliminar ciertos términos de las fórmulas que son por ende más pequeños que otros términos de la misma fórmula.

La segunda columna de la fig. *11.6.* da las premisas especiales necesarias para hallar las fórmulas de costos válidas, dadas por el número de ecuación de la tercera columna. Sin embargo, podemos distinguir dos tipos de premisas. La dependencia de inclusión entre los dominios B en R y S son necesarias para estimar el tamaño de salida. Si no se conservan las dependencias de inclusión, entonces, en promedio, las salidas serán más pequeñas, tal que las fórmulas de costo son límites superiores al costo verdadero. El segundo tipo de premisa establece que el tamaño de la memoria principal, debe ser lo suficientemente grande para almacenar a todas las tuplas de R y S que compartan un *valor-b* común, esto es,

$$B_R / J + B_S / I <= M.$$

Si la relación no se mantiene, entonces ciertas operaciones, que asumimos se llevan a cabo en memoria principal, requieren accesos adicionales a bloques para llevarlos y sacarlo desde memoria principal, como hicimos en el algoritmo del producto de la figura *11.2*. Asi, si estas condiciones son violadas, el costo dado en la columna 3 será un estimado bajo.

La última columna de la figura *11.6*. lista los términos dominantes. Para hallar el término dominante desde cada fórmula, asumimos que B_R , B_S , T_R , T_S , I y J crecen en proporcion, pero M permanece constante. Las fórmulas fueron expandidas y cualquier término que no fuera asintóticamente mayor que otro fué eliminado, dejando los términos dominantes. Por supuesto, dependiendo de los valores de los parámetros, algunos de los términos dominantes para una fórmula podría resultar ser el más grande.

El término dominante nos dice cuál método es el mejor a medida que crezca la relación. y el tamaño de la memoria principal permanezca fija. En particular, el único término dominante que es cuadrático respecto a la relación es $B_R B_S / M$ del join con selección en un producto. Otros términos, ó crecen como el producto del tamaño de la relación por el logaritmo de este tamaño, por ejemplo $B_R \log_M J$ desde el join con creación de índices, o crecen linealmente con el tamaño de la relación, por ejemplo, $B_R T_S / I$. Note como asumimos que I, el tamaño de la imagen de R.B., crece en proporción a B_R, el número de bloques tomados por la relación R.

Asi, para grandes relaciones, esperamos un join con selección en un producto como el peor desempeño, aunque puede ser eficiente en pequeñas relaciones, dada su

simplicidad.[15] El join que usa un índice es el de mejor desempeño, pero no es una opción si no existe tal índice. Entonces, o un join-clasificado o la creación de un índice son alternativas válidas. El valor particular de los parámetros determina cual es mejor en cada situación dada.

11.6. Optimización por manipulación algebraica.

Hasta ahora, en este capítulo, hemos comparado estrategias para ejecutar las operaciones relacionales básicas: selección, producto y join. Ahora miraremos el problema completo de afinar una consulta, escrita en algún lenguaje relacional, en una secuencia de pasos algebraícos relacionales que juntos forman un algoritmo eficiente para responderla.
La primero cosa que un optimizador de consultas lleva a cabo es convertir la consulta en una forma interna que se asemeje al algebra relacional. Por ejemplo, el procesador de consultas QUEL,[16] cuyo algoritmo de optimización discutiremos en la sección 11.10., comienza por asumir un producto cartesiano de relaciones R_1 x R_2 x ... R_K, si los rangos aplicables a las declaraciónes son de la forma

range of t_i *is* R_i para i = 1,2,...k.

Luego, la clausula *where* de la consulta QUEL es reemplazada por una selección y los componentes mencionados en la clausula *retrieve* son obtenidos por una proyección. En realidad, las formas básicas de consulta del Query-by-Example y del SQL son también reemplazadas por expresiones algebraicas similares, una proyección en una selección en un producto.

Equivalencia de expresiones.

---

[15] Debemos ser cuidadosos con nuestro modelo cuando tomamos el join con selección en un producto. La razón es que no contamos la computación realizada en memoria principal. Lo que es probablemente válido cuando decimos, leer pocos bloques y luego clasificarlos; el tiempo de clasificación es menor que el tiempo para leerlos desde memoria secundaria. Sin embargo, la selección en un producto lee repetidamente un bloque de R y luego compara sus tuplas con todas las tuplas de los M-1 bloques de S. Este computación en memoria principal puede significar costo si M es grande.

[16] Usado por el INGRES. Un ejemplo de consulta puede ser : *RANGE OF* e IS empleado
RANGE OF d IS departamento
RETRIEVE(e.nombre,d.nombre)
WHERE e.deptno = d.deptno;

Antes de poder "optimizar" expresiones debemos entender claramente cuando dos expresiones del algebra relacional (de ahora en adelante referidas como expresiones relacionales) son equivalentes. Primero, recordemos que hay dos definiciones de relación en uso y que ellas tienen propiedades matemáticas diferentes. El primer punto de vista es que una relación es un conjunto de k-tuplas para un k fijo, y dos relaciones son iguales si y solo si tienen el mismo conjunto de tuplas. El segundo punto de vista define a una relación como un conjunto de mapeos, desde un conjunto de nombres de atributos a valores. Dos relaciones son consideradas iguales si tienen el mismo conjunto de mapeos. Una relación en el primer sentido puede ser convertida a una relación en el segundo sentido dando nombre de atributo a las columnas. Podemos convertir desde la segunda definición a la primera estableciendo un orden fijo para los atributos.

Usaremos aquí solo la segunda definición, una relación es un conjunto de mapeos desde atributos a valores. La justificación está en que todos los lenguajes de consulta existentes permiten, y generalmente requieren, nombres para las columnas de una relación. De mayor importancia, en cualquier aplicación que seamos conocedores, el orden en el cual las columnas de la tabla son impresas no es significativo, en tanto cada columna está etiquetada con su propio nombre de atributo. Donde sea posible, adoptaremos nombres para cada atributo de la relación computada en una expresión relacional, desde los nombres de atributos para los argumentos de la expresión. También requerimos que los nombres sean suplidos por el resultado de una unión o diferencia de conjuntos.

Una expresión relacional cuyos operandos son relaciones $R_1,R_2,...,R_K$, define una función cuyo dominio son k-tuplas de las relaciones ($r_1,r_2,...,r_k$), donde cada r_i es una relación de la aridad apropiada a R_i. El valor de la función es la relación que resulta cuando sustituimos cada r_i por R_i y luego evaluamos la expresión. Dos expresiones E_1 y E_2 son equivalentes, escrito $E_1 \equiv E_2$, si ellas representan el mismo mapeo; esto es, cuando nosotros sustituimos las misma relación por idénticos nombre en las dos expresiones, obtenemos el mismo resultado. Con esta definición de equivalencia, podemos listar algunas transformaciones algebraícas útiles.

Leyes que involucran join y productos cartesianos.

En el teorema 2.2. (seccción 2.4. volúmen I), probamos como la reunión natural era conmutativa y asociativa. Leyes algebraicas similares se cumplen para los θ-join y para los productos; las estableceremos a continuación.

1. *Leyes conmutativas para los join y los productos*. Si E_1 y E_2 son expresiones relacionales, y F es una condición en los atributos de E_1 y E_2, entonces

$$E_1 \underset{F}{\otimes} E_2 \equiv E_2 \underset{F}{\otimes} E_1$$

$$E_1 \otimes E_2 \equiv E_2 \otimes E_1$$

$$E_1 \times E_2 \equiv E_2 \times E_1$$

2. *Leyes asociativas para join y productos*. Si E_1, E_2 y E_3 son expresiones relacionales, y F_1 y F_2 son condiciones, entonces

$$(E_1 \underset{F_1}{\otimes} E_2) \underset{F_2}{\otimes} E_3 \equiv E_1 \underset{F_1}{\otimes} (E_2 \underset{F_2}{\otimes} E_3)$$

$$(E_1 \otimes E_2) \otimes E_3 \equiv E_1 \otimes (E_2 \otimes E_3)$$

$$(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$$

El lector puede estar sorprendido por algunas de estas leyes; por ejemplo, habíamos afirmado en la sección 2.4. que el producto no era conmutativo. La diferencia radica en que acá estamos usando la definición de conjunto de mapeos para las relaciones, mientras que en la seccion 2.4. usamos la definición de conjunto de listas. Por ejemplo, veamos porque $E_1 \times E_2 \equiv E_2 \times E_1$ se conserva en el presente modelo. En primer lugar, note que nosotros distinguimos los atributos de las dos relaciones , aun si estos atributos tienen el mismo nombre. Así, sean R y S las relaciones que tienen los valores de E_1 y E_2 respectivamente. Entonces el atributo A de R es llamado R.A en el producto de E_1 y E_2, mientras que el atributo A de S es llamado S.A. Podemos reemplazar a R.A. o S.A por A, si A es solo un atributo de R ó de S, pero no de ambas.

Sea ρ una tupla de $E_1 \times E_2$. Entonces hay una tupla μ en R y una tupla ν en S, tal que para todos los atributos A de R, ρ[R.A] = μ[A], y para todos los atributos A de S, ρ[S.A] = ν[A]. Ahora consideremos al producto $E_2 \times E_1$, el cual también debe contener a una tupla τ formada desde μ y ν. Evidentemente, τ [R.A] = μ[A] y τ [S.A] = ν[A] para

todos los atributos A de R y S, respectivamente. Además, τ es ρ. Puesto que ρ es una tupla arbitraria, se cumple que $E_1 \times E_2 \subseteq E_2 \times E_1$. La inclusión opuesta no es menos trivial, tal que concluimos que $E_1 \times E_2$ y $E_2 \times E_1$ son la misma relación.

Leyes que involucran selecciones y proyecciones.

La cascada de varias proyecciones puede ser combinada en una proyección. Expresamos este hecho por

3. *Cascada de proyecciones.*

$$\pi_{A_1,...,A_N}(\pi_{B_1,...,B_m}(E)) \equiv \pi_{A_1,...,A_N}(E)$$

Note que los nombre de atributos $A_1,....A_N$ deben estar entre los B_i para que la cascada tenga sentido.

Similarmente, la cascada de selecciones puede ser combinada en una selección que chequea de una vez por todas las condiciones. La siguiente equivalencia nos permite combinar o descomponer arbitrariamente cualquier secuencia de selecciones.

4. *Cascada de selecciones.*

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_{21}}(E)$$

Dado que $F_1 \wedge F_2 = F_2 \wedge F_1$, se sigue inmediatamente que la selección puede ser conmutada, esto es,

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_2}(\sigma_{F_1}(E))$$

5. *Conmutando selecciones y proyecciones.* Si la condición F involucra únicamente atributos $A_1,....A_n$, entonces

$$\pi_{A_1,...,A_N}(\sigma_F(E)) \equiv \sigma_F(\pi_{A_1,...,A_N}(E))$$

Más generalmente, si la condición F también involucra atributos $B_1,...B_M$ que no están entre $A_1,...A_N$, entonces

$$\pi_{A_1,...,A_N}(\sigma_F(E)) \equiv \pi_{A_1,...,A_N}(\sigma_F(\pi_{A_1,...,A_N,B_1,...B_M}(E)))$$

En la equivalencia anterior puede parecer que la proyección extra a la derecha $\pi_{A_1,\dots,A_N,B_1,\dots B_M}$, es redundante. Sin embargo, esta es una sutileza de la notación. Una proyección malgasta atributos, así como también preserva algunos. En particular, si la relación para E tiene un atributo C que no está entre las A y las B, entonces C es proyectado fuera antes de la selección en la fórmula de la derecha, mientras que a la izquierda es proyectado fuera después de la selección.

6. _Conmutando selecciones con productos cartesianos_. Si todos los atributos mencionados en F son atributos de E_1, entonces

$$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$$

Como un corolario util, si F es de la forma $F_1 \wedge F_2$, donde F_1 involucra solo atributos de E_1, y F_2 involucra solo atributos de E_2, podemos usar las reglas 1,4 y 6 para obtener

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$$

Adicionalmente, si F1 involucra solo atributos de E1, pero F2 involucra atributos de ambos, podemos establecer que

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$$

realizando parte de la selección antes del producto.

7. _Conmutando selecciones con una unión_. Si tenemos la expresión $E = E_1 \cup E_2$, podemos asumir que los atributos de E_1 y E_2 tienen los mismos nombres que los de E, ó al menos, que hay una correspondencia dada que asocie cada atributo de E con un único atributo de E_1 y con un único atributo de E_2. Así podemos escribir

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$$

Si los nombres de los atributos para E_1 y/o E_2 difieren actualmente de los de E, entonces la fórmula F de la derecha debe ser modificada para que use los nombres apropiados.

8. _Conmutando selecciones con un conjunto diferencia_.

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$$

Como en 7, si los nombres de atributos de E_1 y E_2 difieren, debemos reemplazar los atributos de F en la derecha por los nombres correspondientes para E_1. Note también que no es necesaria la selección de la derecha $\sigma_F(E_2)$; podríamos reemplazarla por E2 si lo deseamos. Sin embargo, es usualmente al menos tan eficiente efectuar la selección como

obtener el valor de la expresión E_2, porque la primera es un conjunto más pequeño que la última.

No estableceremos todas las leyes para llevar a cabo la selección antes que el join, dado que un join puede ser expresado siempre como un producto cartesiano seguido por una selección, y en el caso del join natural, una proyección. Las reglas para llevar a cabo una selección antes que un join se derivan de las reglas 4, 5 y 6. Las reglas para mover una proyección antes que un producto cartesiano o una unión son similares a las reglas 6 y 7. Note que no hay una manera general de mover una proyección antes que un conjunto diferencia. Sin embargo, hay una regla útil que aplica a las reuniones naturales y toma ventaja de la ímplicita igualdad entre los atributos usados en la definición de esta operación. La estableceremos acá, y dejaremos la generalización para los equijoin como ejercicio.

9. *Conmutando selecciones con la reunión natural-caso especial*. Si F es una condición que involucra solo atributos compartidos por E_1 y E_2, entoces

$$\sigma_F(E_1 \otimes E_2) \equiv \sigma_F(E_1) \otimes \sigma_F(E_2)$$

Observe que no estamos llevando a cabo trabajo extra ejecutando la selección bajo ambos brazos del árbol de la expresión. La selección reduce el tamaño de las relaciones E_1 y E_2, ahorrando trabajo en ambas ramas.

10. *Conmutando una proyección con un producto cartesiano*. Sean E_1 y E_2 dos expresiones relacionales. Sea $A_1,\ldots,A_N$ una lista de atributos , de los cuales $B_1,\ldots,B_M$ son atributos de E_1, y el resto de atributos $C_1,\ldots,C_K$, son de E_2. Entonces

$$\pi_{A_1,\ldots,A_n}(E_1 \times E_2) \equiv \pi_{B_1,\ldots,B_M}(E_1) \times \pi_{C_1,\ldots,C_K}(E_2)$$

11. *Conmutando una proyección con una unión.*

$$\pi_{A_1,\ldots,A_n}(E_1 \cup E_2) \equiv \pi_{A_1,\ldots,A_N}(E_1) \cup \pi_{A_1,\ldots,A_N}(E_2)$$

Como en la regla 7, si los nombres de los atributos para E_1 y/o E_2 difieren de los de $E_1 \cup E_2$ debemos reemplazar $A_1,\ldots,A_N$ de la derecha por los nombre apropiados.

Principios para la manipulación algebraíca.

Ahora que hemos listado unas equivalencias útiles, podemos formular unas reglas estableciendo la dirección preferida para aplicarlas. No hay un algoritmo que, dada una expresión, garantice producir una expresión equivalente óptima, pero los siguientes principios son generalmente útiles.

1. *Ejecutar las selecciones tan pronto como sea posible*. Esta transformación en las consultas, más que en cualquier otra expresión, es responsable de ahorrar orden de magnitud en tiempo de ejecución, pues tiende a convertir los resultados intermedios en pequeñas evaluaciones de múltiples pasos. Así, en las reglas 6-9, preferimos reemplazar las expresiones de la izquierda con sus equivalentes de la derecha.

2. *Combinar en un join ciertas selecciones con un producto cartesiano previo*. Como hemos visto, un join, especialmente un equijoin, puede ser considerablemente más barato que un producto cartesiano en las mismas relaciones. Cuando el resultado del producto cartesiano R x S es el argumento de una selección, y esa selección involucra comparaciones entre atributos de R y S, el producto es realmente un join. Note que una comparación que no involucre atributos de R ó no involucre atributos de S, puede ser colocada delante del producto y ser aplicada a S ó a R respectivamente, lo que es mejor que convertir el producto en un join.

3. *Combinar secuencia de operaciones unarias*. Una cascada de operaciones unarias -selección y proyección- pueden ser combinadas, aplicandolas en un grupo al recorrer cada tupla. Similarmente, podemos combinar estas operaciones unarias con una operación binaria previa, si aplicamos la operación unaria a cada tupla en el resultado de la operación binaria.

4. *Buscar subexpresiones comunes en una expresión*. Si el resultado de una subexpresión común (una expresión que aparece más de una vez) no es una relación grande, y puede leerse desde memoria secundaria en menos tiempo que el que toma su calculo, entonces es ventajoso precalcularla una sola vez. Subexpresiones que involucren join que no pueden ser modificados moviendo a su interior a una selección caen generalmente en esta categoría. Subexpresiones comunes aparecerán frecuentemente cuando las consultas son expresadas en términos de vistas , pues debemos sustituir la misma expresión para cada ocurrencia de la vista. Si varias consultas son parte de un programa compilado, entonces

tenemos la oportunidad de mirar de una vez por subexpresiones comunes entre todos las consultas.

11.7. Un algoritmo para optimizar expresiones relacionales.

Podemos aplicar las leyes de la sección 11.6. para optimizar una expresión relacional. La expresión resultante "optimizada" está sujeta a los principios enunciados, aunque no hay una completa certeza que sea la óptima entre todas las expresiones equivalentes. Intentaremos mover las selecciones y las proyecciones tan abajo del árbol de reconocimiento de la expresión como sea posible, aunque si hay una cascada de tales operaciones se organizan en una selección seguida de una proyección. También cuando sea posible, agruparemos las selecciones y las proyecciones con la operación binaria que las preceda -unión, producto, join, o diferencia -.

Algunos casos especiales ocurren cuando una operación binaria tiene operandos que son selecciones y/o proyecciones aplicadas a las hojas del árbol. Debemos considerar cuidadosamente como las operaciones binarias se llevan a cabo, y en algunos casos es deseable incorporar la selección o proyección en la operación binaria. Por ejemplo, si la operación binaria es la unión, podemos incorporar las selecciónes y las proyecciónes más abajo en el árbol, sin pérdida de eficiencia, y de cualquier modo copiar los operandos para formar la unión. Por otro lado, si la operación binaria es el producto cartesiano, y ninguna selección es llevada a cabo en el equijoin, preferimos ejecutar las selecciones y las proyecciones en primer lugar, dejando el resultado en una relación temporal, pues el tamaño de la relación operando influye grandemente en el tiempo que toma ejecutar un producto cartesiano completo.

La salida de nuestro algoritmo es un programa completo consistente de de las siguientes clases de pasos.

1. La aplicación de solo una proyección ó selección,
2. La aplicación de una selección seguida de una proyección, o
3. La aplicación de un producto cartesiano, unión o diferencia de dos operandos, quizás precedida por selecciones y/o proyecciones aplicadas a uno o ambos operandos, y posiblemente seguida por esas operaciones.

Asumimos que los pasos 1 y 2 son implementados con una pasada a través de las relaciones operandos, creando una relación temporal. Los pasos del tipo 3 son implementados aplicando, si es apropiado, la selección y/o la proyección a cada tupla de la relación operando, cada vez que la tupla operando es accedida y posteriormente, si es apropiado, se aplicar la proyección ó selección siguiente. El resultado va a una relación temporal.

Algoritmo 11.2. Optimización de expresiones relacionales :

Entrada : Una expresión del álgebra relacional.
Salida : Un programa para evaluar tal expresión.
Método: Ejecutar en orden los siguientes pasos :

1. Use ley 4 para separar cada selección con condiciones de la forma $F_1 \wedge F_2$ en una cascada de selecciones.
2. Use leyes 4 a 8, Para mover cada selección tan abajo en el árbol como sea posible.
3. Use reglas 3,5, 10 y 11 para mover cada proyección tan abajo en el árbol como sea posible. Note que la regla 3 provoca que algunas proyecciones desaparezcan, mientras que la regla 5 divide una proyección en dos proyecciones, una de las cuales puede migrar en el árbol tan abajo como sea posible. Tambien elimine una proyección si ella proyecta todos los atributos de una expresión.
4. Use reglas 3 a 5 para combinar cascadas de selecciones y proyecciones en una sola selección seguida de una sola proyección ó al contrario. Note que esta alteración puede violar la heurística de "Realice una proyección tan rápido como sea posible", pero un momento de reflexión servirá para convencernos que es más conveniente hacer todas las selecciones , luego todas las proyecciones, en una pasada sobre una relación que alternar selecciones y proyecciones en varios pasos.
5. Particionar los nodos interiores del árbol resultante en grupos, de la siguiente manera: Todo nodo interior representando una operación binaria, (x , U , -) conforma un grupo, se incluyen además todos sus ancestros inmediatos que representen operaciones unarias, tambien se incluyen los descendientes que representen operaciones unarias que terminen en una hoja, excepto en los casos en que aparezca

la operación binaria producto cartesiano y no este seguida por una selección que combinada con este producto forme un equijoin.

6. Se produce un programa que evalua en cada paso uno de los grupos formados en un orden tal que ningún grupo es evaluado antes que sus grupos descendiente.