Información de interés:

Optimización de consultas

5.1. Introducción.

En Los modelos de Red y Jerárquico la optimización de consultas es tarea propia del programador de aplicaciones, puesto que las instrucciones de manipulación de datos son propietarias del lenguaje anfitrión. Por el contrario las consultas de base de datos relacionales son o bien declarativas o algebraicas. Los lenguajes algebraicos permiten la transformación algebraica de la consulta, luego, basándose en la especificación algebraica de la consulta es relativamente fácil para el optimizador generar diversos planes equivalentes para la consulta y elegir el menos costoso.

Dado este nivel de generalidad, el optimizador puede ser visto como el generador de código de un compilador para el lenguaje SQL, que produce el código que será interpretado por el motor de ejecución de consultas, excepto que el optimizador marca énfasis en la capacidad de producir el código más eficiente, haciendo uso para tales efectos del catálogo de la base de datos, de donde obtiene información estadística de las relaciones referenciadas por la consulta, algo que los lenguajes de programación tradicionales no hacen.

Un aspecto de la optimización de consultas se sitúa en el nivel del álgebra relacional. Dado un conjunto de reglas se trata de encontrar una expresión que sea equivalente a la expresión dada pero que sea más eficiente en la ejecución.

Con el fin de seleccionar la mejor estrategia para la recuperación de datos el optimizador “estima” un costo que estará relacionado a cada plan de ejecución. Este costo está determinado por fórmulas predefinidas en base a información que se posee de la tabla y que se ha rescatado previamente del catálogo de la base de datos, en realidad el optimizador no siempre escoge el plan más óptimo, ya que encontrar la estrategia óptima puede consumir mucho tiempo, por lo tanto se dice que el optimizador “sólo escoge una estrategia razonablemente eficiente”. La manera con la que el optimizador utiliza esa información, las distintas técnicas y algoritmos que aplica y las transformaciones algebraicas que se realizan son las que diferencian a los optimizadores de bases de datos.

Un optimizador basado en el costo genera una serie de planes de evaluación para una consulta y luego elige el que tiene un menor costo asociado, las medidas de costo comúnmente tienen que ver con la E/S y el tiempo de CPU utilizado en ejecutar la consulta, sin embargo, es cuestión de cada SGBD el elegir las medidas de costo que mejor representen el criterio de minimización en la utilización de recursos.

5.2. Información del catálogo.

Como se ha mencionado anteriormente, la información del catálogo de la base de datos le sirve al SGBD para estimar el costo de los planes de ejecución de una consulta. La precisión de esta información esta ligada directamente a la periodicidad con la que se actualizan las estadísticas del catálogo. En un caso ideal, cada vez que se compromete una transacción de base de datos se deberían actualizar las estadísticas del sistema, sin embargo, esto no es posible en un entorno OLTP[13] dada la sobrecarga que estas operaciones le dan al sistema (básicamente bloqueos en las tablas del catálogo). La solución está entonces en la posibilidad de actualizar estas estadísticas en periodo de poca carga del sistema, algunos SGBD tienen técnicas de optimización automáticas de las estadísticas (como es el caso de MS-SQLServer [Bjeletich99]) pero en general se recomienda que esta tarea la ejecute el DBA con la periodicidad que le dicte la experiencia. En todo caso, la información será más precisa cuanto más bajo sea el nivel de actualizaciones en el intervalo de tiempo entre actualizaciones de Estadísticas.

Cada SGBD tiene distinta información que guardar en el catálogo y por ende, el catálogo de la base de datos es distinto entre cada uno de estos motores, sin embargo, hay información que todo optimizador debe guardar:

n_r	Número de tuplas de la relación r
b_r	Número de bloques que contienen tuplas de la relación r
t_r	Tamaño en bytes de una tupla de r
f_r	Factor de bloqueo de r. Número de tuplas de r que caben en un bloque.
$V(A,r)$	Número de valores distintos del atributo A de la relación r
$CS(A,r)$	Número medio de tuplas de r que satisfacen una condición de igualdad sobre el atributo A de la relación r
	$CS(A,r)=1$ Si A es clave de r
	$CS(A,r)=\frac{n_r}{V(A,r)}$ Si A no es clave de r, se asume una distribución uniforme de los datos de A sobre r

Tabla 5-1 - Información del Catálogo para relaciones simples.

Por ejemplo, si n_r = 10 y $V(sexo,r)$ = 2 entonces $CS(sexo,r)$ = 10 / 2 = 5, se asume por lo tanto que el atributo $sexo$ se distribuye homogéneamente sobre r.

Se asume además que si $\frac{n_r}{b_r}=f_r$, entonces $b_r=\left\lceil\frac{n_r}{f_r}\right\rceil$. Además de la información del catálogo para las relaciones, se utiliza información acerca de los índices:

g_i	Grado de salida de los nodos internos del índice i (para índices con estructura de Arbol B+)
AA_i	Altura del índice para el atributo A de r. $AA_i=\lvert\log_{f_r}(V(A,r))\rvert$
MB_i	Número de bloques que ocupa el nivel más bajo (nivel de hojas) del índice i

Tabla 5-2 - Información del catálogo para índices de Relaciones.

5.3. Medidas de costo.

El costo de un plan de ejecución se puede expresar en términos de distintos recursos de hardware, por ejemplo, en el caso de system R se hace en función de la cantidad de CPU utilizada y de la cantidad de páginas de disco rescatadas. En los SGBD distribuidos se agrega a las medidas el costo de la comunicación de redes. En los sistemas que no son centralizados conviene creer que una buena medida del costo de utilización de recursos es la que está dada por la transferencia de datos desde el disco a la memoria. Para simplificar los cálculos de costo se asume que toda operación que recupere datos de disco tiene el mismo costo, obviando, por simplicidad, los tiempos involucrados en latencia rotacional y el tiempo de búsqueda (ver apéndice C apartado C.1.1).

El costo de todos los algoritmos de optimización depende en gran manera del tamaño de la memoria intermedia (caché) que tenga la memoria principal. En el mejor de los casos todos los datos requeridos se encuentran en la memoria principal por lo que no se hace necesario acceder al disco.

Al igual que en el caso del optimizador de system R, este trabajo presenta los algoritmos de optimización considerando el peor caso, en el cual sólo algunos datos caben en la memoria principal, más bien dicho, sólo un bloque por relación.

5.3.1. Caminos de acceso a selecciones simples.

5.3.1.1. Exploraciones sin índices.

Los siguientes algoritmos implementan la operación de selección sobre una relación cuyas tuplas están almacenadas de manera contigua en un archivo[14].

A1. Búsqueda lineal (Full scan o table scan): Se examina cada bloque del archivo y se comprueba si los registros cumplen con la condición de selección. El costo estimado de este algoritmo es :

$$C_{A1} = b_r$$

A2. Búsqueda binaria: Si la tabla esta ordenada físicamente por el atributo A y la condición de selección es una igualdad sobre el atributo entonces se puede utilizar búsqueda binaria [Kruse84].

El costo estimado será : $$C_{A2} = \lceil \log_2(b_r) \rceil + \left\lceil \frac{CS(A,r)}{f_r} \right\rceil - 1$$

Basado en la suposición de que los bloques de la relación se almacenan de manera contigua en el disco, donde $\lceil \log_2(b_r) \rceil$ es el costo de localizar la primera tupla por medio de búsqueda binaria, y $\left\lceil \frac{CS(A,r)}{f_r} \right\rceil - 1$ es el número de bloques que cumplen la condición de igualdad sobre el atributo A menos el bloque que se ha rescatado con la búsqueda binaria.

5.3.1.2. Exploraciones con índices.

Se denomina “camino de acceso” (access path) a cada una de las formas de acceder a los

registros de una tabla por medio de la utilización de índices. Se entiende como índice primario al índice que permite recorrer los registros de una tabla en un orden que coincide con el orden físico de la tabla, un índice que no es primario se denomina índice secundario. En sybase, un índice en cluster siempre recibe el número 1 dentro del conjunto de índices que pertenecen a una tabla en estrecha relación con la definición de índice primario.

Los algoritmos de búsqueda que utilizan un índice reciben el nombre de "exploraciones de índice" o "index scan". Aunque los índices aseguran la mayor velocidad de acceso a los datos, su utilización implica el acceso a los bloques propios del índice, lo que se puede considerar como un gasto adicional y el cual debe ser tomado en cuenta en la elaboración de algoritmos de acceso a los datos.

Los siguientes algoritmos son los algoritmos de acceso a datos por medio de índices, ya sean primarios o secundarios.

A3. (índice primario, igualdad en la clave): Dado que el índice está construido sobre el (los) atributos claves, la búsqueda puede utilizar este índice para rescatar los datos. Por lo tanto para recuperar el único registro que cumple con la condición se necesita leer sólo un bloque.

El costo estimado será: $C_{A3} = AA_i + 1$

A4. (índice primario, igualdad basada en un atributo no clave): El costo está determinado por la cantidad de bloques que contienen registros que cumplen con la condición de igualdad más la altura del índice.

El costo estimado será: $C_{A4} = \left\lceil \frac{CS(A,r)}{f_r} \right\rceil + AA_i$

Ejemplo: supóngase que

n_{viaje} = 36000,

b_{viaje} = 2391,

t_{viaje} = 136,

f_{viaje} = 15,

Y que existe un índice B+ primario para las columnas *PATENTE+CORRELATIVO*. Se desea seleccionar todos las tuplas de la tabla viajes que tengan como patente = 'HL-8483'.

Como *V(PATENTE,VIAJES)* =30, se estima que 36000/30=1200 tuplas de la relación viajes sean del móvil con patente 'HL-8483'. Se puede utilizar el índice primario para leer los 1200/15=80 bloques que cumplen con la condición.

Ahora bien, supongamos que el índice tiene un tamaño del puntero a disco de 8 bytes, sabiendo que el tamaño de la clave del índice es de 12 bytes es lógico pensar que para un bloque de 2 Kb. El número de claves por bloque de índice es aproximadamente 100. Entonces, una búsqueda por el índice necesitará a lo sumo $\lceil \log_{50}(36000) \rceil$= 3 accesos a bloques de disco, por lo tanto el número total de bloques a disco a leer es de 83 (Una explicación acerca de los cálculos realizados para la estimación de accesos a bloques se puede encontrar en el apéndice B).

A5. (Índice secundario, igualdad): Si el campo indexado no es clave se asume que se rescatará

$CS(A,r)$ registros, como el índice es secundario se estudia el peor caso, el cual propone que cada uno de los registros se encuentra en un bloque diferente, por lo tanto el costo de este caso es:

$$C_{A5} = AA_i + CS(A,r)$$

o

$C_{A5} = AA_i + 1$ si el atributo es clave.

A6. (Índice primario, desigualdad): Si lo que se desea es una selección del tipo $\sigma_{A \leq v}(r)$ con v disponible en el momento de la estimación del costo y asumiendo que los datos del atributo A se distribuyen uniformemente entonces, el número de registros que cumplen con la desigualdad es:

$n_\sigma = 0$ si $v <$ min(A,r)

$n_\sigma = n_r$ si $v >$ max(A,r)

$n_\sigma = nr \cdot \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$ en otro caso.

En cuanto al costo:

Dado que el índice es primario se sabe que la tabla está ordenada por el valor de la clave, por lo tanto, si el operador es > ó >= se realiza una búsqueda por el índice primario para encontrar a v en A y luego se realiza un recorrido lineal partiendo desde esta tupla y hasta el fin de la tabla, lo que devolverá todas las tuplas que cumplen con la condición $A >= v$. Para una desigualdad del tipo < ó <= no es necesario utilizar el índice, simplemente se inicia el recorrido lineal por el archivo que terminará cuando $A = v$ (excluyendo a $A = v$ si el operador es <).

En cualquiera de los dos casos, el costo es:

$C_{A6} = AA_i + \dfrac{b_r}{2}$ si v no se conoce en el momento de la optimización.

$C_{A6} = AA_i + \left\lceil \dfrac{n_\sigma}{f_r} \right\rceil$ si v es conocido en el momento de la optimización.

A7. (índice secundario, desigualdad): al igual que en el caso anterior si v es desconocido se asume que al menos la mitad de los registros cumplen la condición, por lo tanto, el costo de este camino de acceso será:

$C_{A7} = AA_i + \dfrac{MB_i}{2} + \dfrac{n_r}{2}$ si v no se conoce en el momento de la optimización.

que se entiende como el costo de recorrer el índice hasta encontrar la primera hoja del índice que se va a usar, sumarle la mitad de los bloques del nivel de hojas del índice y sumarle (para el caso de los bloques de datos) la mitad de la cantidad de tuplas que tiene la tabla asumiendo en el peor de los casos que se todos los registros se encontraran en bloques diferentes.

No siempre es conveniente utilizar los índices para recuperar rangos de valores, considérese el siguiente ejemplo:

Supóngase que la información estadística es la misma que se proporcionó anteriormente, que existe un índice B+ secundario para la columna TARIFA de la tabla viajes y que se desea seleccionar todas las tuplas de la tabla viajes que tengan una tarifa menor a 5000.

Como el tamaño de la clave del índice es de 4 bytes, es lógico pensar que en una página de índice caben 170 claves. Por lo tanto el índice necesita $\lceil \log_{85}(36000) \rceil$= 3 accesos a bloques de disco para recuperar la primera hoja del índice que se va a ocupar. Suponiendo que existen 600 valores distintos de tarifas, entonces existen entre 600/170 $\cong$ 4 a 8 nodos hojas en el árbol, por lo que, en el peor de los casos se deben leer 8/2 = 4 bloques de índices más 36000/2 = 18000 accesos a bloques de datos. Por lo tanto, el costo de este camino de acceso es de 18000+4+3 = 18007.

Por el contrario un full scan sobre la tabla empleará solamente b_{viaje} = 2391 bloques en vez de los 18007 calculados para el camino de acceso por índice secundario, por lo tanto, de presentarse este ejemplo, el optimizador de consultas optará por un full scan.

5.3.2. Conjunción, Disyunción y Negación.

Conjunción.

Una selección conjuntiva es expresión una de la forma $\sigma_{\theta 1 \wedge \theta 2 .. \wedge \theta n}(r)$. Suponiendo que cada una de las condiciones θ es independiente de las otras se puede estimar el tamaño de cada selección $t_i = \sigma_{\theta i}(r)$. La probabilidad de que una tupla satisfaga la condición de selección es $\frac{t_i}{n_r}$ (llamada selectividad de la selección) y la probabilidad de que una tupla satisfaga la conjunción de selecciones es el producto de todas estas probabilidades.

Por lo tanto se estima el tamaño de la selección completa como:

$$n_r \cdot \frac{t_1 \cdot t_2 \cdot t_3 \cdot \ldots \cdot t_m}{{n_r}^m}$$

Disyunción.

Una selección disyuntiva es una expresión de la forma $\sigma_{\theta 1 \vee \theta 2 .. \vee \theta n}(r)$. Como $\frac{t_i}{n_r}$ denota la probabilidad de que una tupla cumpla θ entonces, la probabilidad de que una tupla cumpla la disyunción es 1 menos la probabilidad de que no cumpla ninguna de las condiciones.

$$1 - (1 - \frac{t_1}{n_r}) \cdot (1 - \frac{t_2}{n_r}) \cdot \ldots \cdot (1 - \frac{t_m}{n_r})$$

por lo tanto, al multiplicar este valor por n_r se obtiene el número de tuplas que satisfacen la relación.

Negación.

Como el resultado de una negación son simplemente las tuplas de r que no están en $\sigma_\theta(r)$, entonces el número de tuplas que cumplen con la condición de negación es:

$tamaño(r) - tamaño(\sigma_\theta(r))$

Algoritmos para la conjunción y disyunción.

A8.(selección conjuntiva utilizando un índice simple): primero hay que determinar si hay disponible un camino de acceso a una de las condiciones simples, de ser así, se puede utilizar cualquier algoritmo del A1 al A7. Luego, en memoria intermedia se verifica que los registros cumplan el resto de las condiciones. Se utiliza la selectividad de las condiciones para determinar en que orden efectuar las selecciones. La condición más selectiva (la que tiene la selectividad más pequeña) recupera un mayor número de registros, por lo tanto esta condición se debe comprobar en primer lugar.

A9.(selección conjuntiva, índice compuesto): si la selección especifica una condición de igualdad en dos o más atributos y existe un índice compuesto en estos campos, se podría buscar en el índice directamente. El tipo de índice determina cual de los algoritmos A3, A4 o A5 se utilizará.

A10.(selección conjuntiva, intersección de identificadores): este algoritmo utiliza índices con punteros a registros en los campos involucrados para cada condición individual. De este modo se examina cada índice en busca de punteros cuyas tuplas cumplan alguna condición individual. La intersección de todos los punteros recuperados forma el conjunto de punteros a tuplas que satisfacen la condición conjuntiva.

A11. (selección disyuntiva mediante la unión de identificadores): al igual que en el caso anterior se utilizan los caminos de acceso en cada una de las condiciones, la unión de los punteros forma el conjunto de punteros a tuplas que satisfacen la condición disyuntiva, sin embargo se deberá efectuar una búsqueda lineal si sólo una de las condiciones (o más) no tiene caminos de acceso.

Ejemplo:

Dada la consulta:

```
SELECT * FROM VIAJES WHERE PATENTE = 'HL-8483' AND TARIFA = 2000
```

Suponiendo que la información estadística es la misma de los ejemplos anteriores, si se utiliza el índice que contiene el atributo PATENTE se necesitan 83 accesos a disco como se observo anteriormente.

Si se utiliza el índice en TARIFA, como hay 600 valores diferentes en el atributo tarifa, significa que $CS(TARIFA,VIAJES)$ = 36000/600 = 60; por lo que en el peor de los casos hay que leer 60 bloques de disco para recuperar cada una de las tuplas. Se sabe además que la altura del índice en el atributo tarifa es de 3, por lo tanto se necesitan sólo 60+3=63 accesos a disco a diferencia de las 83 que se necesitan por medio del uso del índice primario.

Otra técnica que se puede utilizar es la de intersección de identificadores. Para este efecto se sabe que ambos índices tienen altura = 3. para el índice primario, el número de punteros a recuperar es 1200 los cuales caben en 1200/100 = 12 bloques de índice, para el índice secundario el número de punteros a rescatar es de 60 los cuales caben en una sola página. Por lo que hasta el momento se necesitan 6 + 12 + 1 = 19 bloques. Para los bloques de datos se necesita estimar la cantidad de

tuplas que cumplen con la conjunción, como $CS(PATENTE,VIAJES)=1200$ y $CS(TARIFA,VIAJES)=60$ entonces se sabe que (1200*60)/36000 = 2 registros cumplen con la condición conjuntiva, los que en el peor de los casos se rescatan con 2 accesos a bloques de disco, por lo tanto este algoritmo ocupa un total de 21 accesos a disco a diferencia de los 63 que ocupa el camino de acceso del índice secundario. Esta estimación se fundamenta en la suposición de que la distribución de los datos en PATENTES y en TARIFAS es independiente y de que ambos atributos están distribuidos uniformemente en la relación.

5.3.3. JOIN.

Estimación del tamaño.

Sean r(R) y s(S) dos relaciones:

Si $R \cap S = \varnothing$ entonces $r \bowtie s$ es lo mismo que $r \times s$, y por lo tanto se puede utilizar la estimación del producto cartesiano.

Si $R \cap S$ es una clave de R entonces el número de tuplas en $r \bowtie s$ no es mayor que el número de tuplas en S. Si $R \cap S$ es una clave externa de R entonces el número de tuplas de $r \bowtie s$ es exactamente el número de tuplas de S.

Si $R \cap S$ no es clave de R ni de S entonces se supone que cada valor aparece con la misma probabilidad, por lo tanto, sea t una tupla de r y sea $R \cap S = \{A\}$, entonces se estima que la tupla t produce:

$$CS(A,s) = \frac{n_s}{V(A,s)}$$

tuplas en s, por lo tanto se estima el tamaño de $r \bowtie s = \frac{n_r \cdot n_s}{V(A,s)}$ (a)

al cambiar los papeles de r y s se tiene $\frac{n_r \cdot n_s}{V(A,r)}$ (b)

Estos valores serán distintos si y sólo si $V(A,r) \neq V(A,s)$, si este es el caso, la más baja estimación de ambas será la más conveniente.

Algoritmos.

5.3.3.1. Join en bucles anidados.

Si $z = r \bowtie s$, r recibirá el nombre de relación externa y s se llamará relación interna, el algoritmo de bucles anidados se puede presentar como sigue.

```
para cada tupla t_r en r
para cada tupla t_s en s
si (t_r,t_s) satisface la condición θ entonces añadir t_r • t_s al resultado
```

Algoritmo 5-1 - Join en bucles anidados.

Donde $t_r \bullet t_s$ será la concatenación de las tuplas t_r y t_s.

Como para cada registro de r se tiene que realizar una exploración completa de s, y suponiendo el peor caso, en el cual la memoria intermedia sólo puede concatenar un bloque de cada relación, entonces el número de bloques a acceder es de $b_r + n_r \cdot b_s$. Por otro lado, en el mejor de los casos si se pueden contener ambas relaciones en la memoria intermedia entonces sólo se necesitarían $b_r + b_s$ accesos a bloques.

Ahora bien, si la más pequeña de ambas relaciones cabe completamente en la memoria, es conveniente utilizar esta relación como la relación interna, utilizando así sólo $b_r + b_s$ accesos a bloques.

5.3.3.2. Join en bucles anidados por bloques.

Una variante del algoritmo anterior puede lograr un ahorro en el acceso a bloques si se procesan las relaciones por bloques en vez de por tuplas.

```
para cada bloque B_r de r
para cada bloque B_s de s
para cada tupla t_r en B_r
para cada tupla t_s en B_s
si (t_r,t_s) satisface la condición θ entonces añadir t_r • t_s al resultado
```

Algoritmo 5-2 - Join en bucles anidados por bloques.

La diferencia principal en costos de este algoritmo con el anterior es que en el peor de los casos cada bloque de la relación interna s se lee una vez por cada bloque de r y no por cada tupla de la relación externa, de este modo el número de bloques a acceder es de $b_r + b_r \cdot b_s$ donde además resulta más conveniente utilizar la relación más pequeña como la relación externa.

5.3.3.3. Join en bucles anidados por índices.

Este algoritmo simplemente sustituye las búsquedas en tablas por búsquedas en índices, esto puede ocurrir siempre y cuando exista un índice en el atributo de join de la relación interna. Este método se utiliza cuando existen índices así como cuando se crean índices temporales con el único propósito de evaluar la reunión.

El costo de este algoritmo se puede calcular como sigue: para cada tupla de la relación externa r se realiza una búsqueda en el índice de s para recuperar las tuplas apropiadas, sea c = costo de la búsqueda en el índice, el cual se puede calcular con cualquiera de los algoritmos A3, A4 o A5. Entonces el costo del join es $b_r + n_r \cdot c$; si hay índices disponibles para el atributo de join en ambas relaciones, es conveniente utilizar la relación con menos tuplas.

5.3.3.4. Join por mezcla.

El algoritmo de Join por mezcla se pude utilizar para calcular un Join natural o un equi-join. Para tales efectos ambas relaciones deben estar ordenadas por los atributos en común.

Este algoritmo asocia un puntero a cada relación, al principio estos punteros apuntan al inicio de cada una de la relaciones. Según avanza el algoritmo, el puntero se mueve a través de la relación.

De este modo se leen en memoria un grupo de tuplas de una relación con el mismo valor en los atributos de la reunión.

```
Pr = dirección de la primera tupla de r;
ps = dirección de la primera tupla de s;
mientras (ps <> nulo y pr <> nulo)
inicio
ts = tupla a la que apunta ps;
S = {ts};
ps = puntero a la siguiente tupla de s;
hecho = falso;
mientras (not hecho y ps <> nulo)
inicio
ts' = tupla a la que apunta ps;
si (ts'[AtribsReunión] = ts[AtribsReunión])
S = S union {ts'};
ps = puntero a la siguiente tupla de s;
sino
hecho = verdadero;
fin mientras;
tr = tupla a la que apunta pr;
mientras (pr <> nulo y tr[AtribsReunión] < ts[AtribsReunión])
inicio
pr = puntero a la siguiente tupla de r;
tr = tupla a la que apunta pr;
fin mientras;
mientras (pr <> nulo y tr[AtribsReunión] = ts[AtribsReunión])
inicio
para cada ts en S
añadir ts x tr al resultado
pr = puntero a la siguiente tupla de r;
tr = tupla a la que apunta pr;
fin mientras
fin mientras
```

Algoritmo 5-3 - Join por mezcla.

Dado que las relaciones están ordenadas, las tuplas con el mismo valor en los atributos de la reunión aparecerán consecutivamente. De este modo solamente es necesario leer cada tupla en el orden una sola vez. Puesto que sólo se hace un ciclo en ambas tablas, este método resulta eficiente; el número de accesos a bloques de disco es igual a la suma de los bloques de las dos tablas $b_r + b_s$. Si una de las relaciones no está ordenada según los atributos comunes del join, se puede ordenar primero para luego utilizar el algoritmo de join por mezcla. En este caso hay que considerar también el costo de ordenar la o las relaciones y sumarlo al costo de este algoritmo. Como se menciono anteriormente este algoritmo necesita que el conjunto S de tuplas quepa completamente en la memoria principal, este requisito se puede alcanzar normalmente, incluso si la relación s es grande. De no poder cumplirse este requisito se deberá efectuar un join en bucle anidado por bloques entre $\{S\}$ y r con los mismos valores en los atributos del join, con lo que el costo del join aumenta.

Una variación de este algoritmo se puede realizar aún en tuplas desordenadas si existen índices secundarios en los atributos de join de cada una de las relaciones, sin embargo, dada la naturaleza de este tipo de caminos de acceso existe una alta probabilidad de que se necesite un acceso a bloque de disco por cada tupla, lo que lo hace sustancialmente más costoso. Para evitar este costo se puede utilizar un algoritmo de join por mezcla híbrido, que combina índices con un join por mezcla. Si sólo una de las relaciones está desordenada pero existe un índice secundario en el

atributo de join, el algoritmo de join por mezcla híbrido combina la relación ordenada con las entradas hojas del índice secundario. El resultado de esta operación se debe ordenar según las direcciones del índice secundario permitiendo así una recuperación eficiente de las tuplas, según el orden físico de almacenamiento.

5.3.3.5. Join por asociación.

Al igual que el algoritmo de join por mezcla, el algoritmo de join por asociación se puede utilizar para un Join natural o un equi-join. Este algoritmo utiliza una función de asociación h para dividir las tuplas de ambas relaciones. La idea fundamental es dividir las tuplas de cada relación en conjuntos con el mismo valor de la función de asociación en los atributos de join.

Se supone que:

- h es una función de asociación que asigna a los atributos de join los valores $\{0,1,...max\}$
-

- $H_{r_0}, H_{r_1}, .. H_{r_{max}}$ denotan las particiones de las tuplas de r, inicialmente todas vacías. Cada tupla t_r se pone en la partición H_{r_i} donde $i = h(t_r[Atributos\_de\_Join])$

- $H_{s_0}, H_{s_1}, .. H_{s_{max}}$ denotan las particiones de las tuplas de s, inicialmente todas vacías. Cada tupla t_s se pone en la partición H_{s_i} donde $i = h(t_s[Atributos\_de\_Join])$

La idea detrás del algoritmo es la siguiente, supóngase que una tupla de r y una tupla de s satisfacen la condición de join y por lo tanto tienen los mismos valores en los atributos de join. Si estos valores se asocian con algún valor i, entonces la tupla de r tiene que estar en H_{r_i} y la tupla de s tiene que estar en H_{s_i}, de este modo sólo será necesario comparar las tuplas de r en H_{r_i} con las tuplas de s en H_{s_i} y no con las tuplas de s de otra partición.

El algoritmo es el siguiente.

para cada tupla t_s en s
$i = h(t_s[Atributos\_de\_Join])$;
$H_{s_i} = H_{s_i} \cup \{t_s\}$;
fin para;
para cada tupla t_r en r
$i = h(t_r[Atributos\_de\_Join])$;
$H_{r_i} = H_{r_i} \cup \{t_r\}$;
fin para;
para i = 1 hasta *max*
leer H_{s_i} y construir un índice asociativo en memoria sobre él;

para cada tupla t_r en H_r
explorar el índice asociativo en H_s para localizar todas las tuplas
tales que $t_s[Atributos\_de\_Join] = t_r[Atributos\_de\_Join]$;
para cada tupla t_s que concuerde en H_s
añadir t_r x t_s al resultado;
fin para;
fin para;
fin para;

Algoritmo 5-4 - Join por asociación.

El índice asociativo (ver Apéndice B) en H_s se construye en memoria por lo que no se hace necesario acceder al disco para recuperar las tuplas, la función de asociación utilizada para construir este índice es distinta a la función h utilizada para construir las particiones. Después de la división de las relaciones el resto del código realiza un join en bucle anidado por índices en cada una de las particiones. Para lograr esto primero se construye un índice asociativo en cada H_s y luego se *prueba* (se busca en H_s) con las tuplas de H_r.

Se tiene que elegir el valor de max lo suficientemente grande como para que, para cada i, las tuplas de la partición H_s junto con el índice asociativo de la relación quepan completamente en la memoria. Claramente será más conveniente utilizar la relación más pequeña como la relación s. Si el tamaño de la relación s es b_s bloques, entonces para que cada una de las max particiones tenga un tamaño menor o igual a M, max debe ser al menos $\lceil b_s / M \rceil$ más el espacio adicional ocupado por el índice asociativo. Una consideración a seguir para que las divisiones de las relaciones se efectúe en un solo paso es que $M > max+1$ o equivalentemente si $M > (b_s/M)+1$ que de manera aproximada se simplifica a $M > \sqrt{b_s}$. Por ejemplo, si una relación contiene 440 bloques de 2KB cada uno, entonces $M > \sqrt{440}$, osea $M > 20$. De tal manera sólo se necesitarán 40 Kb. de espacio de memoria (más el espacio requerido por el índice asociativo) para el particionamiento de la relación.

El cálculo del costo de un join por asociación tiene la siguiente lógica: el particionamiento de ambas relaciones reclama una lectura completa de cada relación, como también su posterior escritura, esta operación necesita $2(b_r + b_s)$ accesos a bloques. Las fases de construcción y prueba (como se le denomina al join en bucle anidado por índices) lee cada una de las particiones una vez empleando $b_r + b_s$ accesos adicionales. El número de bloques ocupados por las particiones podría ser ligeramente mayor que $b_r + b_s$. Debido a que los bloques no están completamente llenos,. El acceso a estos bloques puede añadir un gasto adicional de $2 \cdot max$ a lo sumo, ya que cada una de las particiones podría tener un bloque parcialmente ocupado que se tiene que leer y escribir de nuevo. Así el costo estimado para un join por asociación es:

$$3(b_r + b_s) + 2 \cdot \max$$

5.3.3.6. Join por asociación híbrida.

El algoritmo de join por asociación híbrida realiza otra optimización; es útil cuando el tamaño de la memoria es relativamente grande paro aún así, no cabe toda la relación s en memoria. Dado que el algoritmo de join por asociación necesita $max + 1$ bloques de memoria para dividir ambas relaciones se puede utilizar el resto de la memoria ($M - max - 1$ bloques) para guardar en la memoria intermedia la primera partición de la relación s, esto es H_{s_0}, así no es necesaria leerla ni escribirla nuevamente y se puede construir un índice asociativo en H_{s_0}.

Cuando r se divide, las tuplas de H_{r_0} tampoco se escriben en disco; en su lugar, según se van generando, el sistema las utiliza para examinar el índice asociativo en H_{s_0} y así generar las tuplas de salida del join. Después de utilizarlas, estas tuplas se descartan, así que la partición H_{r_0} no ocupa espacio en memoria. De este modo se ahorra un acceso de lectura y uno de escritura para cada bloque de H_{r_0} y H_{s_0}. Las tuplas de otras particiones se escriben de la manera usual para reunirlas más tarde.

5.3.3.7. Join Complejos.

Los join en bucle anidado y en bucle anidado por bloques son útiles siempre, sin embargo, las otras técnicas de join son más eficientes que estas, pero sólo se pueden utilizar en condiciones particulares tales como join natural o equi-join. Se pueden implementar join con condiciones más complejas tales como conjunción o disyunción, aplicando las técnicas desarrolladas en la sección 5.3.2 para el manejo de selecciones complejas.

Dado un join de las forma $r \bowtie_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n} s$ se pueden aplicar una o más de las técnicas de join descritas anteriormente en cada condición individual $r \bowtie_{\theta_i} s$, el resultado total consiste en las tuplas del resultado intermedio que satisfacen el resto de las condiciones. Estas condiciones se pueden ir comprobado según se generen las tuplas de $r \bowtie_{\theta_i} s$. La implementación de la disyunción es homóloga a la conjunción.

Outer Join (Join externos)

Un outer join es una extensión del operador join que se utiliza a menudo para trabajar con la información que falta. Por ejemplo, suponiendo que se desea generar una lista con todos los choferes y los autos que manejan (si manejan alguno) entonces se debe cruzar la relación *Chofer* con la relación *Movil*. Si se efectúa un join corriente se perderán todas aquellas tuplas que pertenecen a los choferes, en cambio con un outer join se pueden desplegar las tuplas resultado incluyendo a aquellos choferes que no tengan a cargo un auto. El outer join tiene tres formas distintas: por la izquierda, por la derecha y completo. El join por la izquierda (⟕)toma todas las tuplas de la relación de la izquierda que no coincidan con ninguna tupla de la relación de la derecha , las rellena con valores nulos en los demás atributos de la relación de la derecha y las añade al resultado del join natural. Un outer join por la derecha (⟖) es análogo al procedimiento anterior y el outer join completo es aquel que efectúa ambas operaciones.

Se puede implementar un outer join empleando una de las dos estrategias siguientes. La primera efectúa un join natural y luego añade más tuplas al resultado hasta obtener el join externo. Sea *r(R)*

y $s(S)$, para evaluar $r ⟕ s$ se calcula primero $r ⋈ s$ y se guarda este resultado en la relación temporal $q1$, a continuación se calcula $r - \prod_R(q1)$ que produce las tuplas de r que no participan del join, estas tuplas se rellenan con valores nulos en los atributos de s y se añaden a $q1$ para obtener el resultado deseado. El procedimiento es homólogo para los outer join por la derecha o completo.

La segunda estrategia significa modificar el algoritmo de join. Se pude extender el algoritmo de join en bucles anidados para calcular el outer join por la izquierda. Las tuplas de la relación externa que no concuerden con ninguna tupla de la relación interna se completan con valores nulos y se escriben en la salida.

Para el caso de un outer join completo se puede calcular mediante extensiones de los algoritmos de join por mezcla y join por asociación. En el caso del algoritmo de join por mezcla, cuando se está produciendo la mezcla de ambas relaciones, las tuplas de la relación que no encajan con ninguna tupla de la otra relación se rellenan con valores nulos y se escriben en la salida. Puesto que las relaciones están ordenadas, es fácil detectar si una tupla coincide o no con alguna tupla de la otra relación. El costo estimado para realizar un outer join utilizando el algoritmo de join por mezcla es el mismo que si fuera un join normal, la única diferencia es el tamaño del resultado, por lo tanto el número de bloques a escribir en el resultado puede aumentar.

5.3.3.8. Agregación.

La agregación es otra de las características del álgebra de relaciones extendida. La idea de este tipo de funciones es que tomen un conjunto de valores y que retornen un solo valor. se utilizan comúnmente para información agregada (de ahí su nombre), analizan una agrupación de registros y rescatan el valor solicitado, se aplican sobre un atributo o sobre una composición de atributos, el valor retornado puede ser min, max, count, sum, avg, que corresponden respectivamente a el mínimo valor del subconjunto; el máximo valor; una cuenta de valores; la suma sobre algún atributo o composición de atributos y el promedio.

Para la implementación de la operación de agregación se puede utilizar un algoritmo de ordenación o uno de asociación, para el caso de la asociación, lo que se hace es agrupar las tuplas que tengan el mismo valor en los atributos de la agregación y luego se aplica la función de agregación en cada uno de los grupos para obtener el resultado.

El tamaño de la agregación sobre A es simplemente $V(A,r)$. El costo estimado de la implementación de la agregación es el mismo de una ordenación (cualquiera sea el algoritmo).

En lugar de reunir todas las tuplas en grupos y aplicar entonces las funciones de agregación, se pueden implementar sobre la marcha según se construyen los grupos. Si las $V(A,r)$ tuplas del resultado caben en memoria, las implementaciones basadas en ordenación y las basadas en asociación no requieren escribir bloques adicionales a disco, según se leen las tuplas se pueden ir insertando en una estructura ordenada de árbol o en un índice asociativo, de esta manera sólo se necesitan b_r transferencias de bloques.

---

[14] Se entenderá como archivo al medio de almacenamiento físico en disco ya sea por medio de sistemas de archivos o como device de datos (raw devices) dependiendo de las características del motor de base de datos y del sistema operativo que lo soporta.

5.4. Evaluación de expresiones.

En este apartado se considerará como evaluar una expresión que contiene varias operaciones, la forma intuitiva de evaluar una expresión es evaluar una operación a la vez en un orden apropiado, el resultado de cada operación se *materializa* en una relación temporal para su inmediata utilización. El inconveniente de esta aproximación es la creación de relaciones temporales que implican la escritura y lectura de disco. Una aproximación alternativa es evaluar operaciones de manera simultanea en un *cauce*, con los resultados de una operación pasados a la siguiente sin la necesidad de almacenarlos en relaciones temporales.

5.4.1. Materialización.

Este enfoque de implementación toma la expresión y la representa en una estructura anexa (comúnmente un árbol de operadores). Luego se comienza por las operaciones de más bajo nivel, las entradas a estas operaciones son las relaciones de la base de datos, estas operaciones se ejecutan utilizando los algoritmos ya estudiados y almacenando sus resultados en relaciones temporales. Luego se utilizan estas relaciones temporales para ejecutar las operaciones del siguiente nivel en el árbol.

Una evaluación como la descrita se llama evaluación materializada, puesto que los resultados de cada operación intermedia se crean (materializan) con el fin de ser utilizados en la evaluación de las operaciones del siguiente nivel.

El costo de una evaluación materializada no es simplemente la suma de los costos de las operaciones involucradas. Dado que los costos estimados de los algoritmos presentados anteriormente no consideran el resultado de la operación en disco, por lo tanto, al costo de las operaciones involucradas hay que añadir el costo de escribir los resultados intermedios en disco. Suponiendo que los registros del resultado se almacenan en una memoria intermedia y que cuando esta se llena, los registros se escriben en el disco. El costo de copiar los resultados se puede estimar en n_r/b_r donde n_r es el número aproximado de tuplas de la relación resultado y f_r es el factor de bloqueo de la relación resultado.

5.4.2. Encauzamiento.

Se puede mejorar la evaluación de una consulta mediante la reducción del número de archivos temporales que se producen. Por ejemplo, considérese el join de dos relaciones seguida de una proyección. Si se aplicara materialización en la evaluación de esta expresión implicaría la creación de una relación temporal para guardar el resultado del join y la posterior lectura de esta para realizar la proyección. Estas operaciones se pueden combinar como sigue. Cuando la operación de join genera una tupla del resultado, esta se pasa inmediatamente al operador de proyección para su procesamiento. Mediante la combinación del join y de la proyección, se evita la creación de resultados intermedios, creando en su lugar el resultado final directamente.

La implementación del encauzamiento se puede realizar de dos formas:

1. Bajo demanda (enfoque top-down)
2. Desde los procedimientos (enfoque bottom-up)

En un encauzamiento bajo demanda el sistema reitera peticiones de tuplas desde la operación de la cima del encauzamiento. Cada vez que un operador recibe una petición de tuplas calcula la siguiente tupla a devolver y la envía al procesador de consultas. En un encauzamiento desde los procedimientos, los operadores no esperan a que se produzcan peticiones para producir las tuplas, en su lugar generan las tuplas impacientemente. Cada operación del fondo del encauzamiento genera continuamente tuplas de salida y las coloca en las memorias intermedias de salida hasta que se llenan. Asi, cuando un operador en cualquier nivel del encauzamiento obtiene sus tuplas de entrada de un nivel inferior del encauzamiento, produce las tuplas de salida hasta llenar su memoria intermedia de salida.

El sistema necesita cambiar de una operación a otra solamente cuando se llena una memoria intermedia de salida o cuando una memoria intermedia de entrada está vacía y se necesitan más tuplas de entrada para generar las tuplas de salida. Las operaciones de encauzamiento se pueden ejecutar concurrentemente en distintos procesadores.

El encauzamiento bajo demanda se utiliza comúnmente más que el encauzamiento desde los procedimientos dada su facilidad de implementación.

5.4.3. Algoritmos de encauzamiento.

Supóngase un join cuya entrada del lado izquierdo esta encauzada, dado que esta entrada no está completamente disponible, implica la imposibilidad e utilizar un join por mezcla (dado que no se sabe si la esta entrada viene o no ordenada). El ordenar la relación significa transformar el procedimiento en materialización. Este ejemplo ilustra que la elección respecto al algoritmo a utilizar para una operación y las elecciones respecto del encauzamiento son dependientes una de la otra.

El uso eficiente del encauzamiento necesita la utilización de algoritmos de evaluación que puedan generar tuplas de salida según se están recibiendo tuplas por la entrada de la operación. Se pueden distinguir dos casos:

1. Solamente una de las entradas está encauzada.
2. Las dos entradas de un join están encauzadas.

Si únicamente una de las entradas está encauzada, un join en bucle anidado indexado es la elección más natural, ahora bien, si se sabe de antemano que las tuplas de la entrada encauzada están ordenadas por los atributos de join y la condición de join es un equi-join también se puede usar un join por mezcla. Se puede utilizar un join por asociación híbrida con la entrada encauzada como la relación para probar (relación r). Sin embargo, las tuplas que no están en la primera partición se enviarán a la salida solamente después de que la relación de entrada encauzada se reciba por completo. Un join por asociación híbrida es útil si la entrada no encauzada cabe completamente en memoria, o si al menos la mayoría de las entradas caben en memoria.

Si ambas entradas están encauzadas, la elección de los algoritmos de join se limita. Si ambas entradas están ordenadas por el atributo de join y la condición de join es un equi-join entonces se puede utilizar el método de join por mezcla. Otra técnica alternativa es el join por encauzamiento que se presenta a continuación. El algoritmo supone que las tuplas de entrada de ambas relaciones r y s están encauzadas. Las tuplas disponibles de ambas relaciones se dejan listas para su procesamiento en una estructura de cola simple. Asimismo se generan marcas especiales llamadas Fin_r y Fin_s, que sirven como marcas de fin de archivo y que se insertan en la cola después de que se hayan generado todas las tuplas de r y de s (respectivamente). Para una evaluación eficaz, se deberían construir los índices apropiados en las relaciones r y s. Según se añaden las tuplas a ambas relaciones se deben mantener los índices actualizados.

El algoritmo de join encauzado es el siguiente :

```
hecho_r = falso;
hecho_s = falso;
r = ∅;
s = ∅;
resultado = ∅;
mientras not hecho_r or not hecho_s
si la cola está vacía entonces esperar hasta que la cola no este vacía;
t = primera entrada de la cola;
si t = Fin_r entonces
hecho_r = verdadero;
sino
```

si t = Fin_s entonces
$hecho_s$ = verdadero;
sino
si t es de la entrada r entonces
$r = r \cup \{t\}$;
resultado = resultado $\cup$ ($\{t\} \rtimes s$);
sino
$s = s \cup \{t\}$;
resultado = resultado $\cup$ ($r \rtimes \{t\}$);
fin si
fin si
fin si
fin mientras

Algoritmo 5-5 - Join encauzado.

5.4.4. Transformación de expresiones relacionales.

Hasta ahora se han estudiado algoritmos para evaluar extensiones de operaciones del álgebra relacional y se han estimado sus costos. Dado que una consulta se puede evaluar de distintas maneras y por lo tanto con distintos costos estimados, este apartado considerará formas alternativas y equivalentes a sus expresiones.

5.4.4.1. Equivalencia de expresiones.

Cada implementación de base de datos tiene su forma de representación interna de consultas independientes del lenguaje de consultas utilizado. La representación interna debe cumplir con la característica de ser relacionalmente completo, es por eso que comúnmente los motores de BD eligen la representación del álgebra relacional en forma de árbol sintáctico abstracto para su representación interna.

Dada una expresión del álgebra relacional, es trabajo del optimizador alcanzar un plan de evaluación que calcule el mismo resultado que la expresión dada pero de la manera menos costosa de generar. Para encontrar este plan de evaluación el optimizador necesita generar planes alternativos que produzcan el mismo resultado que la expresión dada y elegir el más económico de ellos. para implementar este paso el optimizador debe generar expresiones que sean equivalente a la expresión dada por medio del uso de las reglas de equivalencia que se explican a continuación.

5.4.4.2. Reglas de equivalencia.

Una regla de equivalencia dice que las expresiones de dos formas son equivalentes, por lo tanto se puede transformar una en la otra mientras se preserva la equivalencia. Se entiende como preservar la equivalencia al hecho de que las relaciones generadas por ambas expresiones tienen el mismo conjunto de atributos y contienen el mismo conjunto de tuplas.

Formalmente se dice que se representa una expresión en su forma canónica.

La noción de forma canónica es central a muchos brazos de la matemática y otras disciplinas relacionadas. Esta puede ser definida como sigue:

Dado un conjunto de Q objetos (digamos consultas) y una noción de equivalencias entre objetos (digamos, la noción de que q1 y q2 son equivalentes si y sólo si ellas producen el mismo resultado), un subconjunto C de Q se dice la forma canónica de Q (bajo la definición de equivalencia expuesta anteriormente) si y sólo si cada objeto q en Q es equivalente a sólo un objeto c en C. El objeto c es llamado la forma canónica de el objeto q. Todas las propiedades de interés que se aplican al objeto q también se aplican a su forma canónica c; por lo tanto es suficiente estudiar sólo el pequeño conjunto de formas canónicas C y no el conjunto Q con el fin de probar una variedad de resultados.

Las reglas de equivalencia para llevar la expresión relacional a una equivalente son:

1. Cascada de proyecciones:

$$\Pi_{L1}(\Pi_{L2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L1}(E)$$

1. Cascada de selecciones:

$$\sigma_{\theta 1 \wedge \theta 2}(E) = \sigma_{\theta 1}(\sigma_{\theta 2}(E))$$

1. Conmutación de selecciones:

$$\sigma_{\theta 1}(\sigma_{\theta 2}(E)) = \sigma_{\theta 2}(\sigma_{\theta 1}(E))$$

1. Conmutación de selección y proyección.

$$\Pi_{L}(\sigma_{\theta}(E)) = \sigma_{\theta}(\Pi_{L}(E))$$

1. Conmutación del Join.

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

1. Asociatividad del Join Natural.

caso1.
$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

caso 2. θ_2 involucra sólo atributos de E_2 y E_3.
$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$

1. Distributividad de la selección con respecto al join.

caso 1. θ_0 involucra sólo atributos de E_1.
$\sigma_{\theta 0}(E_1 \bowtie_{\theta} E_2) = \sigma_{\theta 0}(E_1) \bowtie_{\theta} E_2$

caso 2. θ_1 involucra sólo atributos de E_1 y θ_2 involucra sólo atributos de E_2

$\sigma_{\theta 1 \wedge \theta 2}(E \bowtie_{\theta} E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_{\theta} (\sigma_{\theta 2}(E_2))$
1. Distributividad de la proyección con respecto al join. Si L_1 y L_2 son los atributos de E_1 y E_2 respectivamente. $\Pi_{L1 \cup L2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L1}(E_1)) \bowtie_{\theta} (\Pi_{L2}(E_2))$
1. Conmutatividad de la unión y la intersección. $E_1 \cup E_2 = E_2 \cup E_1$ $E_1 \cap E_2 = E_2 \cap E_1$ La diferencia de conjuntos no es conmutativa.
1. Asociatividad de la unión e intersección. $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$ $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
1. Distributividad de la selección con respecto a la unión, intersección y diferencia. $\sigma_{\theta}(E_1 \cup E_2) = \sigma_{\theta}(E_1) \cup \sigma_{\theta}(E_2)$ $\sigma_{\theta}(E_1 \cap E_2) = \sigma_{\theta}(E_1) \cap \sigma_{\theta}(E_2)$ $\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2) = \sigma_{\theta}(E_1) - E_2$
1. Distributividad de la proyección con respecto a la unión. $\Pi_{L}(E_1 \cup E_2) = (\Pi_{L}(E_2)) \cup (\Pi_{L}(E_1))$

Tabla 5-3 - Reglas de equivalencia para expresiones relacionales.

Ejemplo:

Supóngase que lo que se desea es notificar a todos los dueños de vehículos del año, que estén siendo conducidos por los choferes con menos experiencia (supóngase un año o menos); la expresión relacional sería:

$$\Pi_{Dueño.nombre}(\sigma_{Moviles.año=2001 \wedge Choferes.Fecha\_licencia\_desde > Fecha()-1\_año}((Dueño \bowtie Moviles) \bowtie Choferes))$$

Se puede utilizar la regla 6.1 con el fin de asociar el Join.

$$\Pi_{Dueño.nombre}(\sigma_{Moviles.año=2001 \wedge Choferes.Fecha\_licencia\_desde > Fecha()-1\_año}(Dueño \bowtie (Moviles \bowtie Choferes)))$$

Aplicando la regla 5 se puede conmutar el Join.

$$\Pi_{Dueño.nombre}(\sigma_{Moviles.año=2001 \wedge Choferes.Fecha\_licencia\_desde > Fecha()-1\_año}((Movile \bowtie Choferes) \bowtie Dueños))$$

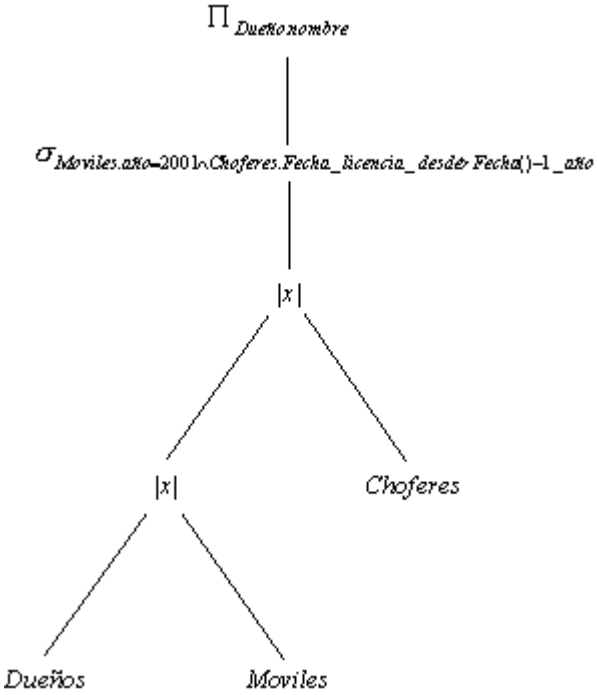
Luego se aplica la regla 7.1 con el fin de distribuir la selección sobre el join.

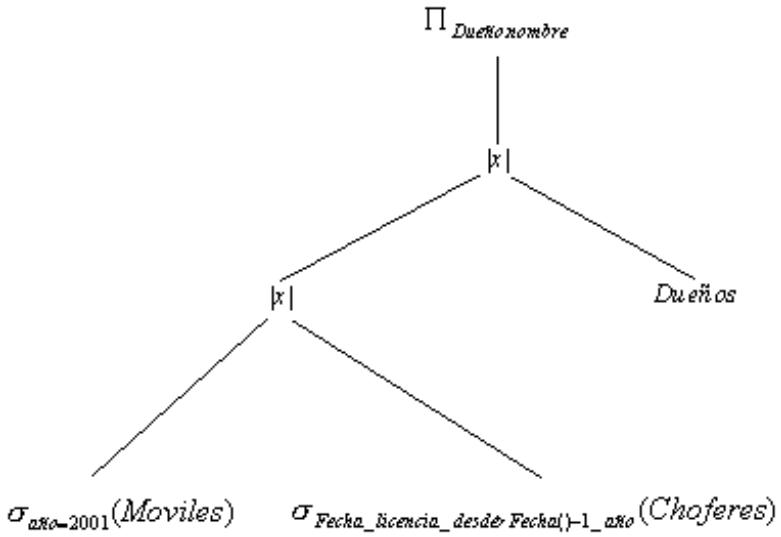
$$\Pi_{Dueño.nombre}((\sigma_{Moviles.año=2001 \wedge Choferes.Fecha\_licencia\_desde > Fecha()-1\_año}(Movile \bowtie Choferes)) \bowtie Dueños)$$

Aplicando la regla 7.2 se obtiene :

$$\Pi_{Dueño.nombre}(((\sigma_{año=2001}(Movile)) \bowtie (\sigma_{Fecha\_licencia\_desde > Fecha()-1\_año}(Choferes))) \bowtie Dueños)$$

Las siguientes figuras muestran la expresión inicial y la final en una estructura de árbol sintáctico.





Así, los optimizadores generan de manera sistemática expresiones equivalentes a la consulta dada. El proceso se entiende como sigue: dada una expresión, se analiza cada subexpresión para saber si se puede aplicar una regla de equivalencia. De ser así se genera una nueva expresión donde la subexpresión que concuerda con una regla de equivalencia se reemplaza por su equivalente. Este proceso continúa hasta que no se pueden generar más expresiones nuevas. Una optimización en términos de espacio se puede lograr como sigue. Si se genera una expresión E1 de una expresión E2 mediante una regla de equivalencia, entonces E1 y E2 son equivalentes en su estructura y por lo tanto sus subexpresiones son idénticas. Las técnicas de representación de expresiones que permiten a ambas expresiones apuntar a la subexpresión compartida pueden reducir el espacio de búsqueda significativamente.