ISSN: 1692-7257 Volumen 2 Número 6 año - 2005

Revista Colombiana de
Tecnologías de Avanzada

OPTIMIZATION OF RELATIONAL JOIN ALGORITHMS FOR DATABASE TECHNOLOGIES

OPTIMIZACIÓN DE ALGORITMOS DE JOIN RELACIONAL PARA TECNOLOGÍAS DE BASES DE DATOS

Ing. Bell Manrique Losada, MSc. Francisco Javier Moreno
MSc. Jaime Alberto Guzmán Luna.

Universidad Nacional de Colombia Sede Medellín
Escuela de Sistemas, Facultad de Minas, Núcleo Robledo
bmanriq@unalmed.edu.co, fjmoreno@unalmed.edu.co, jaguzman@unalmed.edu.co

Abstract: In the following paper we present the Relational Join Algorithms more frequently utilized for the optimization in databases systems, by specifying the performance of each one of them, their determinant characteristics and demonstratives examples of their application in real cases. We also describe the influence of the *interesting orders* in query optimization and its exemplification, to utilize theirs explicitly in the Databases Management Systems.

Resumen: En este artículo se presentan los algoritmos de Join Relacional más utilizados en el procesamiento y optimización en Bases de Datos, especificando el comportamiento de cada uno de ellos, sus características determinantes y ejemplos demostrativos de su aplicación en casos reales. Además, se describe la influencia de los órdenes interesantes en la optimización de consultas y su ejemplificación, con miras en utilizarlos explícitamente en la optimización en Sistemas de Gestión de Bases de Datos.

Keywords: Query Optimization, Interesting Orders, Join Algorithms, Relational Join, Database Technologies.

1. INTRODUCCIÓN

El objetivo de este artículo se centra en presentar los principales algoritmos que implementan la operación de coincidencia binaria conocida como *Join Relacional*. Éstos son: Nested Loop Join, Merge Join y Hash Join. El interés principal es ejemplificar la estructura de estos algoritmos, sus características predominantes, su aplicación a casos reales y las consideraciones necesarias para escoger los caminos de acceso adecuados en el procesamiento y optimización de consultas en bases de datos.

El interés secundario es que esta ejemplificación pueda ser utilizada como base conceptual acerca de los algoritmos de Join, que permitan un planteamiento posterior con modificaciones a su

estructura y aplicación, para obtener un mejor resultado. El aporte se centra en la comparación de los algoritmos para Joins y en las pautas para la elección de cada uno de ellos, teniendo en cuenta diversos parámetros que serán presentados.

En el artículo se procede de la siguiente forma: en la Sección 2 se presenta una introducción al procesamiento de consultas en Bases de Datos. En la Sección 3, se realiza el análisis de los algoritmos de Join: Nested Loop, Merge y Hash Join. En la Sección 4, se muestra la utilización de los *Órdenes Interesantes*. En la Sección 5, se presenta un Análisis Comparativo de los Algoritmos, por medio de un ejemplo que utiliza sus fórmulas de costos. En la Sección 6, se presentan las Conclusiones y Trabajos Futuros.

2. EL PROCESAMIENTO DE CONSULTAS

El Procesamiento de Consultas hace referencia a la serie de actividades que se deben seguir para llevar a cabo la recuperación de datos desde una base de datos, entre las que se incluye la traducción de consultas a expresiones que se puedan implementar en un nivel físico y algoritmos de evaluación y optimización de consultas (Cisterna N., M. 2002; Graefe, G., 1993). Para empezar el Procesamiento de una consulta, el sistema debe traducirla a un lenguaje de representación interno con el cual le sea fácil operar. El plan de ejecución es un árbol que se obtiene a partir de la consulta y que es entendido por su motor de ejecución. Para la transformación de la consulta en un plan, se usa la información del catálogo de la base de datos que le permite obtener información acerca de las relaciones, los datos, su almacenamiento físico y la forma de acceder a ellos.

El siguiente paso es traducir el árbol de la consulta en el plan de ejecución. Generalmente existe un gran número de planes que pueden implementar el árbol de la consulta, el proceso de encontrar el mejor de estos planes se denomina ***optimización de consultas***, la cual viene dada por una medida de rendimiento para la ejecución de consultas (como por ejemplo el tiempo de ejecución).

El objetivo es que el plan sea el óptimo o el más cercano al óptimo, dada la información contenida en el catálogo de la Base de Datos y los costos asociados a cada uno de los tipos de algoritmos de ejecución. Finalmente el optimizador envía el plan óptimo al motor de ejecución, el cual ejecuta el plan usando como entrada las relaciones almacenadas en las base de datos y produce como salida una tabla con los datos solicitados.

3. ALGORITMOS DE JOIN

Cuando se requiere buscar información almacenada en más de una relación, se habla de una *reunión* o *Join* (Cisterna N., M. 2002), cuyo símbolo es ⋈. El operador Join permite seleccionar información de más de una relación y combinar los resultados en una relación de salida. Su Se conoce como un **operador binario**, pues trabaja con dos operandos (relaciones de entrada). El algoritmo para implementar un Join varía dependiendo del operador usado, que puede ser: =, <, >, >=, etc., incluso para un mismo operador existen varias alternativas. (Arroyo, J., 2000)

3.1 Nested Loop Join.

Es el Join realizado por medio de dos Bucles Anidados (Bernstein, A., 1980; Zhang, P., 2004). En esta operación R⋈S, ocurre el Join entre las relaciones R y S, en donde ***R*** recibe el nombre de **relación outer** y ***S*, relación inner**. El algoritmo procede como se muestra en la Figura 1.

para cada tupla **r** en **R (openScan)**
  para cada tupla **s** en **S (segment scan)**
    si (r • s) satisface la condición **C** sobre os atributos de join,
      entonces añadir **r • s** al resultado

Figura 1. Algoritmo Join en Bucles Anidados. (Bernstein, A., 1980).

Para cada tupla en la **relación externa R**, se recorre completamente la **relación interna S**, lo que significa que para cada registro de R se tiene que realizar una exploración completa de S, para recuperar una a una las tuplas que coinciden con el predicado de Join (condición **C**). Si la más pequeña de las relaciones cabe completamente en memoria, es conveniente utilizarla como la externa. El optimizador puede escoger este tipo de algoritmo si la condición de Join no contiene una condición de igualdad; y normalmente es menos eficiente que los otros algoritmos que se describirán, debido al recorrido completo que hace de las dos relaciones.

3.2 Merge Join.

Es el Join por Mezcla y se puede utilizar para calcular un Join natural o un Equi-Join (Zhang, P., 2004). Está basado en la idea de ordenar las relaciones y luego inspeccionarlas. Esta operación lee las relaciones de entrada, que deben estar ordenadas por los atributos de Join, y para cada fila de la relación externa, el algoritmo lee todas las tuplas que coinciden de la interna, accesando las filas en forma ordenada. Si las relaciones de entrada no están ordenadas por los atributos Join, el optimizador adiciona una operación de ordenamiento, la cual adiciona un costo al Join. El algoritmo se muestra en la Figura 2.

Este algoritmo asocia un puntero a cada relación, comenzando en la tupla inicial de cada una. Según avanza el algoritmo, el puntero se mueve a través de la relación, de tal forma que se leen en memoria un grupo de tuplas de una relación con el mismo valor en los atributos de Join. Dado que las relaciones están ordenadas, las tuplas con el mismo valor en el atributo de Join aparecerán consecutivamente, de esta manera solamente es necesario leer cada tupla una sola vez. Puesto que sólo se hace un ciclo en ambas relaciones, este método resulta bastante eficiente. El optimizador escogerá este algoritmo si es similar el tamaño de las relaciones de entrada, o en el caso de presencia de órdenes interesantes (Ver Sección 4).

```
pr = dirección de la primera tupla de r;
ps = dirección de la primera tupla de s;
mientras (ps <> nulo y pr <> nulo)
  {     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)
  {     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])
  {     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])
  {     para cada ts en S
                        añadir ts x tr al resultado
final
                pr = puntero a la siguiente tupla
de r;
                tr = tupla a la que apunta pr;
        }
  fin mientras
fin mientras
```

Figura 2. Algoritmo Join por Mezcla. (Cisterna N., M., 2002).

3.3 Hash Join.

Es el Join por Asociación (Zhang, P., 2004), y se puede utilizar, al igual que el Merge, para calcular un Join natural o un Equi-Join. La idea de este algoritmo es dividir las tuplas de cada relación en conjuntos con el mismo valor, utilizando una *función h* la cual asocia un valor (*clave hash*) a cada uno de los datos existentes en los atributos de Join. El algoritmo se muestra en la Figura 3, y se tiene que:

- ***h*** es una función de asociación que asigna a los atributos de Join los valores *{0,1,...n}*
- $H_{r0}...H_{rn}$ denota las particiones de las tuplas de ***r***, inicialmente vacías. Cada tupla t_r se pone en la partición H_r donde ***i =h (*** t_r***[Atributos Join])***
- $H_{s0}...H_{sn}$ denotan las particiones de las tuplas de *s*, inicialmente vacías. Cada tupla t_s se pone en la partición H_s donde ***i =h (*** t_s***[Atributos Join])***

```
para cada tupla ts en s
    i=h( ts [Atributos de Join])
    Hsi = Hsi U ts
fin para;
para cada tupla tr en r
    i=h( tr [Atributos de Join])
    Hri = Hri U tr
fin para;
para i = 1 hasta n
    leer Hsi y construir un índice asociativo en
    memoria sobre él;
    para cada tupla tr en Hri
        explorar el índice asociativo en Hsi para
        localizar todas las tuplas tales que ts
        [Atributos de Join] = tr [Atributos de Join];
          para cada tupla ts que concuerde en Hsi
                añadir tr x ts al resultado;
          fin para;
    fin para;
```

fin para;
Figura 3. Algoritmo Join por Asociación. (Cisterna N., M., 2002).

El algoritmo construye en memoria una tabla de asociación de la más pequeña de sus dos relaciones de entrada, llamada *relación de construcción*, y luego lee la más grande, llamada *relación de prueba*, y prueba en memoria la tabla de asociación para encontrar las tuplas coincidentes en el atributo Join que serán escritas en una relación resultante. Si la entrada más pequeña no cabe en memoria, se particiona en tablas de trabajo más pequeñas tal que sólo es necesario comparar las tuplas de *R* en H_{ri} con las de *S* en H_{si}, y no con las de *S* de otra partición.

4. ÓRDENES INTERESANTES

Los *Interesting Orders* u *Órdenes Interesantes,* son ciertos ordenamientos que tratan las relaciones base (las almacenadas en la Base de Datos) y los resultados intermedios (producidos por operaciones relacionales) para evitar ordenamientos innecesarios en el procesamiento de consultas y reducir los costos de procesamiento de Joins. (Ramakrishnan, R. y Gehrke, J., 1998). Éstos se tratan como *ordenamientos lógicos*, pues especifican una condición que un grupo de tuplas debe encontrar para satisfacer un ordenamiento dado. El *ordenamiento físico* es la sucesión real de las tuplas en disco (Selinger, P. *et al.*, 1979). Para la optimización de una consulta estos *órdenes* son relevantes pues pueden ayudar a reducir los costos de su ejecución, debido a que el generador del plan puede economizar en operadores de ordenamiento (Silberschatz, K. y Sudarshan, 1997). Un Orden Interesante puede encontrarse como: 1) Un Atributo, si participa en un predicado de Join, o si ocurre en la cláusula OrderBy o GroupBy; o como 2) Resultado Intermedio, si es ordenado por alguno de los atributos de OrderBy, GroupBy o Join. Este tipo de orden ocurre por ejemplo cuando a una Relación se le aplica una operación de ordenamiento sobre su atributo de Join (*Ordenamiento Físico*); luego se le aplica una operación *Selección* sobre uno de sus atributos, generando una nueva relación (resultado intermedio) que se considera un *Orden Interesante*, pues tiene un *ordenamiento Lógico* que el optimizador asume por ser el resultado de la ejecución de un operador sobre una relación con ordenamiento físico; por lo tanto, este resultado intermedio también está ordenado sobre el atributo de Join y no requiere una operación de ordenamiento adicional. De esta manera podría ser usado en un Merge-Join, por ejemplo en el caso que este resultado se reúna con otra relación (Join).

5. ANÁLISIS DE TIPOS DE JOIN

Para comparar el desempeño de cada uno de los algoritmos vistos, se presenta a continuación un ejemplo analítico, con base en los costos de ejecución de cada uno de ellos, los cuales tienen unas métricas asociadas. En este caso se tomará la Métrica de Costos: número de operaciones de Entrada/Salida de páginas, y la siguiente información de las relaciones DEPARTAMENTO y EMPLEADO, las cuales se llamarán de ahora en adelante relaciones R y S. Sea nr= Número de tuplas de r = 50, br= Número de tuplas/página de r= 10 (por lo tanto r ocupa entonces 5 Páginas), ns= Número de tuplas de s= 6000, bs= Número de tuplas/página de s = 500 (por lo tanto s ocupa 12 Páginas).

Nested-Loop. En el peor caso, si hay suficiente memoria solamente para llevar una página de cada relación, el costo estimado es ***nr/br + (nr/br * ns/bs)*** accesos a disco (cesma.usb.ve, 2004). Si la relación más pequeña cabe completamente en memoria, se usa como la relación interna. Es más eficiente si se usa la relación más pequeña como la relación externa, es decir la relación con menos páginas. En el ejemplo, si se toma DEPARTAMENTO como la relación externa:
nD/bD + (nD/bD ∗ nE/bE) = 50/10 + (50/10 * 6000/500) = 65
Si se toma EMPLEADO como la relación externa:
nE/bE + (nE/bE ∗nD/bD) = 6000/500 + (6000/500*50/10) = 72

Merge-Join. Como cada bloque necesita ser leído una sola vez, asumiendo que todas las tuplas para algún valor dado de los atributos de Join caben en memoria, el número de accesos a bloques para este Join es (cesma.usb.ve, 2004): ***nr/br + ns/bs*** o ***nr/br + ns/bs + z***, con *z* el costo de ordenar las relaciones si no lo están. En el ejemplo, cuando **r** y **s** están ordenados, el costo es: nr/br + ns/bs = (50/10) + (6000/500) = 17

Sort-Merge Join. Para hallar su costo general

(udlap.mx, 2004), primero se halla el costo de los ordenamientos, **2 nr/br $\log_{B-1}$nr/br + 2 ns/bs $\log_{B-1}$ ns/bs** y se le suma el de la mezcla **(nr/br + ns/bs)** (Ramakrishnan, R. y Gehrke, J., 1997), de tal forma que el costo general de éste es: ***2[(nr/br)$\log_{B-1}$(nr/br)+(ns/bs)$\log_{B-1}$(ns/bs)]+(nr/br+ns/bs).*** En el ejemplo, con B=3, el costo es:
2[(nD/bDr) $\log_{B-1}$(nD/bD)+(nE/bE)$\log_{B-1}$(nE/bE)]+(nD/bD+nE/bE) = 2[5 $\log_2$ 5 + 12 $\log_2$ 12] + (5 + 12) = 2 [11.7+43.1] + (17) = 126.6

Hash Join. En este algoritmo es mejor escoger la relación más pequeña como la relación de construcción. Si no se requiere particionamiento recursivo, debido a que cada registro es leído y escrito una vez en la fase de particionamiento, y es leído una segunda vez para hacer el Join en la fase de prueba, el costo es (cesma.usb.ve, 2004): ***3(nr/br + ns/bs).*** Cuando una relación cabe en el buffer de memoria, no necesita particionamiento (Graefe, G., 1993) y el costo se reduce a ***nr/br + ns/bs*.** En el ejemplo: 3 [(nr/br) + (ns/bs)] = 3[(50/10) + (6000/500)] = 51

Si requiere particionamiento recursivo (Ramez A., E., y Shamkant B., N., 2000), el costo es:
2 [(nr/br)+(ns/bs)][$\log_{(B-1)}$(ns/bs)-1]+(nr/br+ns/bs).

En el ejemplo:
2 [(nD/bD)+ (nE/bE)][$\log_{(B-1)}$(nE/bE)-1]+ (nD/bD+ nE/bE) =
2 [(50/10)+ (6000/500)][$\log_{(3-1)}$(6000/500)-1]+ (50/10+ 6000/500) = 2 [5 + 12][3.6-1]+(17) = 2[17][2.6]+[17] = 105

La Figura 6 muestra gráficamente la comparación de los costos de cada algoritmo de Join, considerados en número de operaciones E/S de páginas, tomando los valores aleatorios de la Tabla 1, que especifican para dos relaciones el número de tuplas/número de tuplas por página, de cada una, así:

Tabla 1. Valores aleatorios para Tipos de Join.

#Pág. / Join	50/10, 6000/500 (5, 12)	60/10, 7000/500 (6, 14)	100/10, 10000/500 (10, 20)	120/10, 15000/500 (12, 30)
NLJ	65	90	210	372
MJ	17	20	30	42
SMJ	126.6	158	270	423
HJ	51	60	90	126

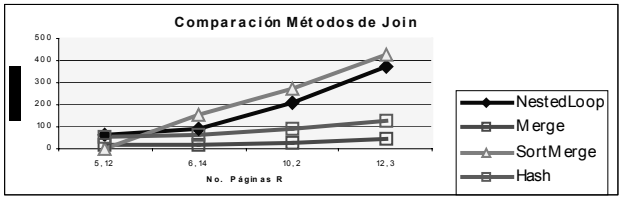

Figura 4. Comparación Métodos, con X=# Páginas R

Teniendo en cuenta los costos asociados a cada operación de Join, el siguiente ejemplo muestra su utilización para escoger los algoritmos adecuados en la optimización de una consulta, teniendo en cuenta la información que reposa en la Tabla 2.

Tabla 2. Datos de Relaciones Ejemplo.

	(R)	(S)	(T)	(W)
# tuplas	50 (nr)	6000 (ns)	2000 (nt)	600 (nw)
# tuplas/página	10 (br)	500 (bs)	100 (bt)	60 (bw)
# de páginas	5(pr)(nr/br)	12 (ps)	20 (pt)	10 (pw)

Teniendo las Relaciones anteriores, se quiere realizar una consulta cuya condición de Join es: R.Id = S.Id AND R.Id = T.Id AND R.Id = W.Id. La Operación Global de Join es: **R ⋈ S ⋈ T ⋈ W.** Para este ejemplo se hicieron las pruebas con 2 tipos de operaciones Join: Sort Merge y Nested Loop, para cada par de relaciones, así:

Primer Caso:
R ⋈ S → Sort Merge Join; **T ⋈ W** → Sort Merge Join; **R ⋈ S ⋈ T ⋈ W** → Merge Join.
Los costos para cada Join son:
$*\mathbf{R}\bowtie\mathbf{S}_{(SMJ)}$ = 2 [(pr) $\log_{B-1}$(pr) +(ps) $\log_{B-1}$ (ps)] + (pr + ps) = 2[11.60+ 43.1]+ (17) = **126.4**
$*\mathbf{T}\bowtie\mathbf{W}_{(SMJ)}$ = 2[(pt) $\log_{B-1}$(pt)+(pw)$\log_{B-1}$(pw)]+ (pt+pw) = 2[86.5 + 33.3]+(30) = **270**
$*\mathbf{R}\bowtie\mathbf{S}\bowtie\mathbf{T}\bowtie\mathbf{W}_{(MJ)}$ = [prs + ptw] = [126.4 + 270] = **396.4**

Segundo Caso:
R ⋈ S → Nested Loop Join; **T ⋈ W** → Nested Loop Join; **R ⋈ S ⋈ T ⋈ W** → Nested Loop Join
Los costos para cada Join son:
$*\mathbf{R}\bowtie\mathbf{S}_{(NLJ)}$ = [pr + (pr * ps)] = [5 + (5*12)] = **65**

$*T \bowtie W_{(NLJ)} = [pt + (pt * pw)] = [20 + (20*10)] =$ **220**
$*R \bowtie S \bowtie T \bowtie W_{(NLJ)} = [prs + (prs * ptw)] = [65 + (65*220)] =$ **14365**

Como se observa en estos casos, cuando se evalúan los costos asociados a cada posible operación de Join, **los óptimos locales no siempre son los óptimos globales**. Para las dos primeras operaciones de Join $R \bowtie S \bowtie T \bowtie W$, entre Nested Loop y Sort Merge, es mejor escoger Nested Loop, pues su costo en número de operaciones I/O de páginas es menor; pero para la operación global $R \bowtie S \bowtie T \bowtie W$, entre Merge y Nested Loop, es más óptimo escoger Merge Join, pues se cuenta con un *interesting order* que viene del ordenamiento producido por los join anteriores (Sort Merge). Incluso localmente no siempre es el Nested Loop mejor que el Sort Merge, esto dependerá del tamaño de las relaciones.

6. CONCLUSIONES Y TRABAJOS FUTUROS

Se han analizado los principales algoritmos para la realización de un Join. El Nested Loop, el Merge y el Hash. El *Nested Loop* no necesita índices y puede ser utilizado con cualquier clase de operador (=, <, >) para realizar el Join. Si la más pequeña de ambas relaciones cabe completamente en la memoria, es conveniente utilizarla como la relación externa en dicho algoritmo, ya que esto reduce el costo de acceso a disco. Este método en general es costoso debido a que examina cada combinación de tuplas en las dos relaciones. 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. También se puede disminuir la cantidad de accesos sobre la relación interna indexando su atributo de Join.

El *Merge* permite calcular solamente Joins naturales y Equi-Joins. Es muy eficiente cuando las relaciones están ordenadas por el atributo de Join, pues solo se hace un ciclo en las dos tablas. El *Hash* permite calcular Joins naturales y Equi-Joins. Es mejor escoger la relación más pequeña como la relación de construcción y la más grande como la relación de prueba. Un Join entre más de dos relaciones es más complejo de analizar. Como muestra la Gráfica 1 en el caso de las tres relaciones analizadas, aunque inicialmente el Nested Loop es mejor que el Merge, en el resultado final resulta ser mejor el Merge, pues su costo baja debido al orden interesante que se genera.

Se planea realizar un análisis más exhaustivo entre diferentes combinaciones de métodos de Join para más de tres relaciones. Igualmente se pretende modificar dichos métodos para mejorar su rendimiento en situaciones particulares, como por ejemplo en el contexto de las bodegas de datos (Datawarehouse), donde se suelen realizar operaciones de Join entre la *tabla de hechos* y las *tablas de dimensiones*. Debido a que éstas no están normalizadas y aunque los métodos tradicionales de Join funcionan, debe ser posible plantear variantes que aprovechen este aspecto y mejoren su rendimiento.

RECONOCIMIENTOS

El trabajo descrito en este artículo ha sido apoyado por los siguientes proyectos de investigación:

Tesis de Maestría "Modelo de Planificación de Consultas con Control de Calidad en Sistemas de Información Basados en Mediadores", apoyado por la Escuela de Sistemas de la Universidad Nacional de Colombia, Sede Medellín.

Tesis de Doctorado "Modelo Distribuido y Cooperativo Basado en Agentes Ontológicos y de Planificación, para la composición Automática de Servicios Web Semánticos**",** auspiciada por Colciencias, ICFES, ICETEX, Universidad Nacional de Colombia, Sede Medellín y el Banco Mundial, enmarcado en el programa de apoyo a la comunidad científica nacional en programas de Doctorado 2004.

REFERENCIAS

Arroyo, J. (2000) Diseño de Sistemas de Bases de Datos. Lección No. 12: *Procesamiento de preguntas*. J. ICOM6005.

Bernstein, A. (Enero, 1980). Lecture 12: *Query Processing*. Relational Algebra and SQL. Lecturas de Clase.

Cisterna Neira, M. (2002). *Métodos de Optimización de Consultas para el Lenguaje SQL*. Dpto. de Matemática y Ciencia de la Computación, Universidad de Santiago de Chile. Santiago, Chile.

Graefe, G. (1993). *Query Evaluation Techniques for Large Databases*. Revision of November 10, 1993. Portland State University.

Ramakrishnan, R. y Gehrke, J. (1998). *Evaluation of Relational Operations.* Chapter 14, Part A (Joins). Database Management Systems 3ed.

Ramakrishnan, R. y Gehrke, J. (1997). *Implementing Natural Joins*. Implementation of Relational Operators (Joins). Chapter 12, Part A.

Ramez A. Elmasri, y Shamkant B. Navathe. (2000). CS143: *Join Algorithms*. Extraído de: www.cs.ucla.edu/classes/

Selinger, P. Griffiths; Astrahan M.M.; Chamberlin, D.D.; Lorie, R.A. y Price, T.G. (1979*) Access Path Selection in a Relational Database Management System*. *ACM SIGMOD.*

Silberschatz, Korth y Sudarshan. (1997). *Database Systems Concepts*.

Zhang, P. (Octubre, 2004). CSCI 5708: *Query Processing II*. University of Minnesota. Curso.

udlap.mx (Octubre, 2004) http://ict.udlap.mx/people/carlos/bases/notes/proc-queries.pdf

cesma.usb.ve (Octubre, 2004) http://www.bd.cesma.usb.ve/ci5313/docs/optimiza.pdf