Bases de Datos

Procesamiento y Optimización de Consultas

1er. Cuatrimestre 2008

Indice

Procesamiento y Optimización de Consultas

IMPORTANTE:
Este apunte es un complemento de la bibliografía que la materia utiliza como base para el tema.
No contiene una visión exhaustiva ni completa de los contenidos, sólo pretende ser un elemento más de ayuda para la comprensión de los mismos.

1 Introducción.

El presente apunte está orientado a dar una visión práctica del procesamiento de consultas sobre bases de datos relacionales.

Además de dar una idea básica de las principales acciones que realiza un motor de base de datos relacionales para procesar una consulta, se brinda un panorama simplificado de las diferentes maneras de resolver las consultas, del cálculo de sus costos y de la optimización de su ejecución.

Si bien cada motor de base de datos puede tener sus particularidades, entendemos que lo que aquí se plantea da una base razonable para permitir la comprensión de los mismos.

El análisis de costos de la ejecución de determinadas sentencias SQL (sobre todo algunas que sean críticas o de uso muy frecuente) es importante en determinadas situaciones para completar el diseño físico de una base de datos relacional. Si bien este análisis puede abarcar todo tipo de sentencias DML, el enfoque de este apunte está orientado sólo al procesamiento de las consultas SQL.

2 Descripción General del Procesamiento de Consultas

2.1 Procesamiento de Consultas

Los lenguajes de consultas relacionales (como el SQL) nos dan una interfaz declarativa de alto nivel para acceder a los datos almacenados en una base de datos.

El procesamiento de consultas se refiere al conjunto de actividades que realiza un motor de base de datos para la extracción de datos de la base de datos a partir de una sentencia en un lenguaje de consulta.

Los pasos básicos son:

1.- Parsing y traducción

2.- Optimización

3.- Generación de código

4.- Ejecución de la consulta

Estos pasos en general son realizados por diferentes componentes del motor. Los componentes clave son: el optimizador de consultas y el procesador de consultas.

La idea general es que luego del parsing se construya una expresión algebraica equivalente (o podría ser más de una), se analicen diferentes formas de resolver la consulta (estas formas se llaman planes de ejecución) evaluando sus costos, y se seleccione la más eficiente. Esto lo hace el componente generalmente llamado Optimizador de Consultas.

Luego, esta consulta es pasada al otro componente generalmente llamando Procesador de Consultas, que es el que se encarga de ejecutar físicamente la consulta de acuerdo al plan de ejecución, produciendo y devolviendo el resultado.

2.2 Componente Optimizador de Consultas

Mencionamos especialmente este componente ya que es el que más nos interesa para el enfoque de la práctica.

El optimizador de consultas es el responsable de generar el plan, que va a ser el input del motor de ejecución. Debe ser un plan eficiente de ejecución de una consulta SQL, perteneciente al espacio de los posibles planes de ejecución de esa consulta.

La cantidad de posibles planes de ejecución para una consulta puede ser grande, ya sea porque algebraicamente se la puede escribir de diferentes maneras lógicamente equivalentes, o porque hay más de un algoritmo disponible que implemente una expresión algebraica dada.

Como el procesamiento de cada plan de ejecución puede tener un rendimiento diferente, la tarea del optimizador es realmente importante a efectos de encontrar una buena solución.

Típicamente, a partir de una consulta en lenguaje SQL, construye una expresión en álgebra relacional de la siguiente manera:

1. Realiza el producto cartesiano de las tablas del FROM de izquierda a derecha.
2. Realiza un select con las condiciones de la cláusula WHERE.
3. Realiza una proyección con las columnas de la cláusula SELECT.

Una vez obtenida la expresión, pasa a armar un plan de ejecución de la consulta construyendo un árbol, el cual consta de expresiones de álgebra relacional, donde las relaciones son las hojas y los nodos internos, las operaciones algebraicas. Es importante notar que una misma consulta puede tener varios planes de ejecución (es decir, varios árboles pueden producir los mismos resultados).

En general, el objetivo principal es minimizar la cantidad de accesos a disco. El optimizador construye diferentes planes basándose en:

- Los distintos algoritmos implementados en el procesador de consultas y disponibles al optimizador que implementan las operaciones de álgebra relacional (Proyección, Selección, Unión, Intersección, Resta, Join)
- Información acerca de:
  - La estructura física de los datos (ordenamiento, clustering, hashing)

- La existencia de índices e información sobre ellos (ej : nro de niveles que posee).
- Estadísticas guardadas en el catálogo (esta información no está "al día" constantemente por razones de eficiencia, sino que se actualiza periódicamente):
  - Tamaño de archivos y factor de bloqueo.
  - Cantidad de tuplas de la relación
  - Cantidad de bloques que ocupa la relación
  - Cantidad de valores distintos de una columna (esto permite estimar la selectividad de una operación)

- Ciertas heurísticas que le permiten encontrar planes de ejecución sin necesidad de generar en forma completa el espacio de búsquedas (como podría ser tratar de armar planes que realicen cuanto antes las operaciones más restrictivas y maximicen el uso de índices y de las estructuras físicas existentes)

Una vez construidos los diferentes planes alternativos, el optimizador selecciona el que sea más eficiente, el cual es su output.

3 Descripción del modelo utilizado

A continuación describimos el modelo que utilizaremos en la materia para trabajar sobre procesamiento y optimización de consultas. Como ya se dijo, este es un modelo simplificado, en las bases de datos reales por lo general se encontrará mayor complejidad de casos que los que veremos aquí.

3.1 Relaciones

Las relaciones se dividen en bloques o páginas. La longitud de estos bloques se define a nivel general (se lo denota con **LB**), por lo que todas las relaciones poseen bloques del mismo tamaño.

Dada una relación R, se conocen los siguientes datos:

- $\mathbf{B_R}$: Cantidad de bloques que ocupa R
- $\mathbf{FB_R}$: Cantidad de tuplas por bloque de R (factor de bloqueo)
- $\mathbf{L_R}$: Longitud de una tupla de R
- $\mathbf{T_R}$: Cantidad total de tuplas de R
- $\mathbf{I_{R,A}}$: Imagen del atributo A de R. Denota la cantidad de valores distintos de ese atributo en la relación.
- **X**: altura del árbol de búsqueda
- $\mathbf{FB_I}$: Cantidad de entradas por bloque del índice I (factor de bloqueo del índice)
- $\mathbf{BH_I}$: Cantidad de bloques que ocupa el índice i para nodos hoja (el índice debe ser árbol $B^{+)}$. Si no se tiene el dato y se desea calcularlo, se utilizará el criterio de peor caso (cada nodo hoja estará completo en su mínimo, es decir, d/2, donde d es el orden del árbol).
- $\mathbf{MBxB_I}$: Cantidad máxima de bloques que ocupa un bucket del índice i (el índice debe ser basado en hashing)

- **CBu_I:** cantidad de buckets del índice I (el índice debe ser basado en hashing)

Además, asumiremos que:

- en cada bloque de la relación R solamente hay tuplas de R (y no de otras relaciones).
- La longitud de cada tupla es fija para cada tabla.
- La longitud de cada tupla es siempre "bastante" menor que la longitud de un bloque, en el sentido que un bloque puede contener más de una tupla.

3.2 Memoria principal

La memoria principal disponible para la base de datos también se divide en bloques. Llamaremos B a la cantidad de bloques disponibles.

3.3 Costos

Como unidad de medida se utilizarán los accesos a disco (tanto lecturas como escrituras).

Vale la pena mencionar que motores de bases de datos reales pueden tener en cuenta otros factores que también influyen, como el tiempo insumido por cada acceso a disco, el tiempo insumido por cada acceso a memoria, el tiempo de procesamiento de determinadas operaciones (como ser la evaluación de una función de hashing sobre una clave). En nuestro modelo simplificado estos factores no serán tenidos en cuenta.

3.4 Aclaraciones generales

- En algunos casos podría suceder, por ejemplo, que se necesitara utilizar un bloque de una relación y que éste ya se encontrara en memoria principal. Aprovechando esto, se podría evitar el acceso a disco (con el consiguiente ahorro de costos); sin embargo, es difícil predecir cómo la base de datos hará uso de la memoria principal (puede depender de varios factores como la política de reemplazos de bloques de memoria), por lo que se asumirá que siempre se accede a disco.
- Los costos de las búsquedas serán expresados en ***peor caso***.
- Dentro de un bloque sólo se almacenarán tuplas enteras. Esto va a hacer que en algunos casos se desperdicie espacio dentro de algunos bloques.
- Todos los archivos o índices están organizados en bloques.
- Los tamaños de los bloques de disco y de memoria son iguales.

4 Organización de archivos e índices

Recordaremos brevemente algunos conceptos sobre organización de archivos e índices relacionados con el proceso de optimización de consultas, analizando los costos de acceso a los datos.

Para el análisis de costos de acceso sólo consideraremos tres operaciones:

a) la exploración completa de las tuplas

b) la búsqueda de tuplas a través de una clave por igualdad (A=c)

c) la búsqueda de tuplas a través de una clave según un rango de valores (c <= A<=d)

4.1 Organización de archivos y registros

Como ya se vio en la materia, desde un punto de vista lógico una base de datos relacional está compuesta por múltiples relaciones, que son tablas compuestas por tuplas.

En un nivel físico, esas tablas son archivos de registros, los que pueden estar agrupados en páginas o bloques.

Los atributos de las relaciones aquí serán los campos de los registros.

Todo registro de una tabla tiene un identificador único llamado rid (record identifier). Este identificador es independiente del contenido del registro, y es utilizado directamente por el DBMS.

En el alcance de la práctica de la materia, veremos dos tipos de archivos: Heap files y Sorted files.

4.2 Heap files

4.2.1 Descripción

Los archivos Heap son el tipo de archivo más simple, que consiste de una colección desordenada de registros, agrupados en bloques.

4.2.2 Costo de exploración completa

Hay que recorrer linealmente todas las tuplas del archivo (file scan). Para esto hay que leer todos los bloques de datos.

El costo de acceso es:

B_R

4.2.3 Costo de búsqueda por igualdad (A = c)

En este caso no queda más remedio que recorrer linealmente los bloques del archivo de datos.

Como en la materia siempre consideramos peor caso para las búsquedas, tendremos que recorrer siempre todos los bloques de datos del archivo, ya que puede haber más de una tupla, o en caso de haber una sola encontrarse en último lugar.

El costo de acceso es:

$$B_R$$

- *Si no consideráramos peor caso, podríamos pensar que cuando se trata de claves candidatas el costo se reduce a* **(BR / 2)**, *ya que podríamos frenar la búsqueda al encontrarla, y en promedio esto nos daría ese resultado.*

4.2.4 Costo de búsqueda por rango (c <= A <= d)

Aquí también la única alternativa es recorrer linealmente el archivo de datos en forma completa.

El costo de acceso es:

$$B_R$$

4.3 Sorted File

4.3.1 Descripción

Los archivos Sorted contienen los registros ordenados de acuerdo a los valores de determinados campos.

En lo que sigue supondremos que el archivo está ordenado según un atributo A.

4.3.2 Costo de exploración completa

Hay que recorrer linealmente todas las tuplas del archivo (file scan). Para esto hay que leer todos los bloques de datos. Un valor agregado en este caso es que el resultado está ordenado según A.

El costo de acceso es:

$$B_R$$

4.3.3 Costo de búsqueda por igualdad (A = c)

En este caso se realiza una búsqueda binaria sobre los bloques del archivo para encontrar el que contenga la primer tupla que coincida con la clave. En caso de que A no sea clave candidata (i.e.,puede haber más de una tupla con el valor buscado), habrá que seguir recorriendo las tuplas siguientes hasta consumir todas las coincidentes con el valor buscado.

En una primera aproximación, diremos que el costo de acceso es:

$$\log_2(B_R) + B'$$

donde B' es la cantidad de bloques adicionales que ocupan las tuplas que cumplen con el criterio de la búsqueda.

Refinando la fórmula anterior, si suponemos que T' es la cantidad de tuplas que cumplen con el criterio de búsqueda, tendremos que el costo de acceso es:

$$\log_2(B_R) + [\ T' / FB_R\ ]$$

donde la notación de parte entera debe interpretarse como 'parte entera por exceso'.

- *Notar que si A es clave candidata sólo habrá una tupla y no será necesario ningún bloque adicional. En este caso el costo se simplifica a* ***$\log_2(B_R)$***

4.3.4 Costo de búsqueda por rango (c <= A <= d)

Este caso es similar al de la búsqueda por igualdad con una clave no candidata, ya que habrá que encontrar la primer tupla con valor c en el atributo A y luego recorrer secuencialmente.

En una primera aproximación, diremos nuevamente que el costo de acceso es:

$$\log_2(B_R) + B'$$

donde B' es la cantidad de bloques adicionales que ocupan las tuplas que cumplen con el criterio de la búsqueda.

Refinando la fórmula al igual que en el punto 4.3.3, tendremos que el costo de acceso es:

$$\log_2(B_R) + [\ T' / FB_R\ ]$$

donde T' es la cantidad de tuplas que cumplen con el criterio de búsqueda. En la fórmula anterior, la notación de parte entera debe interpretarse como 'parte entera por exceso'.

4.4 Indices

Para permitir accesos a la información de una forma no soportada (o no eficientemente soportada) por la organización básica de un archivo, se suelen mantener estructuras adicionales llamadas índices. Estas estructuras contienen información (entrada o index entry) que permite lograr el acceso deseado.

Los índices mejoran el acceso sobre un campo o conjunto de campos de las tuplas de una tabla.

En general, las entradas de los índices asociadas a una clave k (la llamaremos k*) pueden ser de la siguiente forma:

a) k* = el valor clave k y el registro de datos asociado

b) k* = el valor clave k y el rid del registro asociado

c) k* = el valor clave k y una lista de los rid de los registros asociados.

En la práctica de la materia vamos a trabajar con índices de tipo árbol B^+ (clustered y unclustered) e índices basados en hash estático.

4.5 Propiedades de los índices

4.5.1 Índices Clustered vs. unclustered

Si los datos del archivo están ordenados físicamente en el mismo orden que uno de sus índices, decimos que ese índice es **clustered**. Caso contrario es **unclustered.**

Los archivos de datos a lo sumo pueden tener un índice clustered, en tanto que la cantidad de índices unclustered es ilimitada.

- *Notar que si las entradas de un índice **árbol B$^+$** son de la forma a) mencionada en el punto 4.4, ese índice es naturalmente clustered.*

4.5.2 Índices Densos vs. No Densos

Los índices **densos** son los que tienen una entrada de índice por cada valor de clave de búsqueda del archivo de datos. En cambio, si no todos los valores clave están en el índice, se llaman **no densos**. Por ejemplo, si tenemos un índice clustered ciertas veces se optimiza el uso del espacio de índices manteniendo en un índice no denso sólo el primer valor de clave de cada bloque del archivo de datos.

En la práctica de la materia trabajaremos sólo con índices densos.

4.5.3 Índices Primarios vs. secundarios.

Se llaman índices primarios (no confundir con claves primarias) a aquellos índices que contienen todos los registros completos de los archivos. En caso de tener sólo los rids se los llaman índices secundarios.

- *Notar que los índices con entradas de la forma a) mencionada en el punto 4.4 corresponden a índices primarios, en tanto que las b) y c) corresponden a índices secundarios.*

En la práctica de la materia trabajaremos sólo con índices secundarios.

4.6 Índices Arbol B+ clustered

4.6.1 Descripción

Los índices de tipo **árbol B$^+$** son árboles balanceados con una cantidad de claves por nodo interno dada por un orden d. Cada nodo interno tendrá entre d/2 y d claves (con excepción de la raíz, cuyo mínimo de claves es 1). Los nodos hoja contienen la información k* de todos los registros del archivo. Estos índices son los recomendados para acceder a rangos de claves.

Adicionalmente, los indices **árbol B$^+$ clustered** son aquellos para los cuales el archivo de datos asociado está ordenado en el mismo orden que dicho índice.

- *En la materia trabajaremos con índices clustered secundarios, es decir, con el archivo de datos separado del índice, por lo que nuestros análisis asumirán directamente esto.*

En lo que sigue supondremos que el índice está basado en el atributo A.

4.6.2 Costo de exploración completa

En este caso el índice no nos ayuda, con lo cual hay que recorrer todos los bloques del archivo de datos.

El costo de acceso es

$$B_R$$

Las tuplas resultantes quedan ordenadas según el atributo A.

4.6.3 Costo de búsqueda por igualdad (A = c)

Se deben leer los bloques del árbol B^+ comenzando desde la raíz hasta el nodo hoja correspondiente (son X bloques, uno por nivel del árbol). Se obtiene el rid del primer registro que cumple la igualdad y se lee el bloque correspondiente del archivo de datos. En caso de que haya más de un registro coincidente con la selección, se siguen recorriendo secuencialmente los bloques de datos.

En una primera aproximación, diremos que el costo de acceso es:

$$X + B'$$

donde B' es la cantidad de bloques que ocupan las tuplas que cumplen con el criterio de la búsqueda.

Refinando la fórmula anterior, si suponemos que T' es la cantidad de tuplas que cumplen con el criterio de búsqueda, tendremos que el costo de acceso es:

$$X + 1 + [\ T' / FB_R\ ]$$

ya que al suponer el peor caso de búsqueda consideramos que la segunda tupla está en un bloque distinto de la primera, es decir, la primera está en un bloque y las restantes están contiguas a partir del siguiente bloque. En la fórmula anterior, la notación de parte entera debe interpretarse como 'parte entera por exceso'.

- *Notar que si A es clave candidata sólo habrá una tupla y no será necesario ningún bloque adicional. En este caso el costo se simplifica a* ***X + 1***

4.6.4 Costo de búsqueda por rango (c <= A <= d)

Similar al anterior, pero la primer búsqueda se hace por el valor c, accediendo inicialmente al archivo de datos por el registro con menor clave que sea >= c, es decir, el primer registro que cumpla la condición.

En una primera aproximación, diremos nuevamente que el costo de acceso es:

$$X + B'$$

donde B' es la cantidad de bloques que ocupan las tuplas que cumplen con el criterio de la búsqueda.

Refinando la fórmula al igual que en el punto 4.6.3, tendremos que el costo de acceso es:

$$X + 1 + [\ T' / FB_R\ ]$$

donde T' es la cantidad de tuplas que cumplen con el criterio de búsqueda. En la fórmula anterior, la notación de parte entera debe interpretarse como 'parte entera por exceso'.

Las tuplas resultantes quedan ordenadas según el atributo A.

4.7 Índices Arbol B+ unclustered

4.7.1 Descripción

Los indices de tipo **árbol B^+ unclustered** son índices árbol B^+ para los cuales el archivo de datos asociado no está ordenado según el orden de dicho índice.

En lo que sigue supondremos que el índice I está basado en el atributo A.

4.7.2 Costo de exploración completa

Este caso es similar al índice unclustered, con la diferencia que las tuplas resultantes no quedan ordenadas según el atributo A.

El costo de acceso es

$$B_R$$

4.7.3 Costo de búsqueda por igualdad (A = c)

Se deben leer los bloques del árbol B^+ comenzando desde la raíz hasta el nodo hoja correspondiente (son X bloques, uno por nivel del árbol). Se obtiene el rid del primer registro que cumple la igualdad y se lee el bloque correspondiente del archivo de datos. En caso de que haya más de un registro coincidente con la selección, se resuelve esto dependiendo de la implementación del índice (asumiremos que la recuperación de los punteros para un valor de clave en una búsqueda por igualdad no tiene costo adicional una vez que se llegó al nodo hoja del índice). Para cada uno de los punteros, se debe acceder al bloque de datos que contiene la tupla apuntada (o sea, tenemos que levantar un bloque de datos por cada entrada del índice coincidente).

En una primera aproximación diremos que el costo de acceso es

$$X + T'$$

donde T' es la cantidad de tuplas coincidentes con la búsqueda.

- *Notar que si A es clave candidata sólo habrá una tupla y no será necesario ningún bloque adicional. En este caso el costo se simplifica a* $X + 1$

4.7.4 Costo de búsqueda por rango (c <= A <= d)

Similar al anterior, pero la primer búsqueda se hace por el valor c, accediendo inicialmente al archivo de datos por el registro con menor clave que sea >= c.

En una primera aproximación, diremos nuevamente que el costo de acceso es:

X + BH' + T'

donde BH' es la cantidad de bloques de nodos hoja del índice a recorrer, y T' es la cantidad de tuplas coincidentes con el criterio de búsqueda.

Refinando la fórmula al igual que en el punto 4.7.3, tendremos que el costo de acceso es:

$X + [T' / FB_I] + T'$

donde T' es la cantidad de tuplas que cumplen con el criterio de búsqueda y FB_I el factor de bloqueo del índice I. En la fórmula anterior, la notación de parte entera debe interpretarse como 'parte entera por exceso'.

Las tuplas resultantes quedan ordenadas según el atributo A.

4.8 Índices basados en hash estático

4.8.1 Descripción

Los índices **basados en hash estático** están compuestos por una cantidad determinada (fija) de buckets, donde la información k* es ubicada a través de la utilización de una función de hash. Cada bucket es una lista encadenada de bloques, los que se irán agregando bajo demanda en la medida en que se vayan completando los bloques previos de ese bucket. Estos índices son los ideales para búsquedas por igualdad.

- *En la materia trabajaremos con índices basados en hash secundarios, es decir, con el archivo de datos separado del índice, por lo que nuestros análisis asumirán directamente esto.*

En lo que sigue supondremos que el índice está hasheado por el atributo A.

4.8.2 Costo de exploración completa

En este caso el índice no nos ayuda, con lo cual hay que recorrer todos los bloques del archivo de datos.

El costo de acceso es

B_R

4.8.3 Costo de búsqueda por igualdad (A = c)

Se evalúa la función de hashing sobre la clave (asumimos costo cero para esta operación), luego accedemos al bucket correspondiente. A partir de ahí recorremos los bloques del bucket buscando los rids de las tuplas. En este último caso, por cada entrada del índice que cumpla la condición se necesitará acceder al bloque correspondiente del archivo de datos.

- *Notar que en general la cantidad de registros a buscar será un número relativamente pequeño.*

El costo de acceso es:

$MB x B_I + T'$

donde **$MBxB_I$** es la cantidad máxima de bloques de un bucket del índice I, y T' es la cantidad de tuplas coincidentes con la búsqueda.

4.8.4 Costo de búsqueda por rango (c <= A <= d)

En este caso el índice no nos ayuda, con lo cual hay que recorrer todos los bloques del archivo de datos.

El costo de acceso es

B_R

4.9 Cuadro resumen de costos de acceso

El siguiente cuadro resume los costos de acceso de las operaciones de consulta básicas según los diferentes tipos de archivo e índices que utilizaremos en la práctica de la materia, siempre teniendo en cuenta nuestro modelo simplificado.

Tipo de archivo / índice	Costo de exploración completa	Costo de búsqueda por igualdad (A = k)	Costo de búsqueda por rango (k_1 <= A <= k_2)
Heap file	**B_R**	**B_R**	**B_R**
Sorted file	**B_R**	**$\log_2(B_R) + [T' / FB_R]$**	**$\log_2(B_R) + [T' / FB_R]$**
Índice B+ clustered sobre A	**B_R**	**$X + 1 + [T' / FB_R]$**	**$X + 1 + [T' / FB_R]$**
Índice B+ unclustered sobre A	**B_R**	**$X + T'$**	**$X + [T' / FB_I] + T'$**
Índice hash estático sobre A	**B_R**	**$MBxB_I + T'$**	**B_R**

Donde :

- **[]** debe interpretarse, en esta tabla, como 'parte entera por exceso'
- T' es la cantidad de tuplas que cumplen con el criterio de la búsqueda
- FB_R es el factor de bloqueo del archivo
- FB_I es el factor de bloqueo del índice I
- $MBxB_I$ es la cantidad máxima de bloques de un bucket

5 Evaluación de operaciones relacionales

Como ya hemos mencionado, para procesar consultas se construyen planes de ejecución, que son árboles donde inicialmente los nodos son operadores algebraicos. Cada operación algebraica tiene una o más implementaciones físicas. Llamaremos operador físico a cada implementación.

Las diferentes implementaciones aprovechan o explotan algunas propiedades de las tablas a efectos de lograr una mejor performance, por ejemplo:

- Existencia de índices relevantes en los archivos input
- Ordenamiento interesante de los archivos de input
- Tamaño de los archivos de input
- Tamaño del buffer

Los diversos algoritmos se basan en algunos de los siguientes principios:

- **Iteración:** se examinan todas las tuplas de la relación, aunque a veces se examina el índice.
- **Indexación:** si hay una condición de selección y/o join y a su vez existe un índice sobre los campos involucrados, se utilizará el índice.
- **Particionamiento:** particionando la relación según una clave de búsqueda podremos descomponer la operación en operaciones menos costosas.

Los diferentes métodos para recuperar tuplas se llaman **caminos de acceso**: nosotros veremos el ***file scan*** (recorrido sobre el archivo de datos) y el ***index scan*** (recorrido sobre las entradas de un índice).

Coincidencia con un índice (*index matching*):

- un predicado p coincide con la clave de un índice árbol B^+ si el predicado es una conjunción de términos de la forma (A op c) involucrando a todos los atributos de un prefijo de la clave del índice (o a la clave completa)
- un predicado p coincide con la clave de un índice basado en hash si el predicado es una conjunción de términos de la forma (atributo = valor) para todos los atributos del índice

Llamaremos **selectividad** de un predicado p (y lo denotamos sel(p)) a la proporción de tuplas de una relación que satisfacen ese predicado, es decir, la división entre la cantidad de tuplas que satisfacen el predicado sobre la cantidad total de tuplas de las relación.

Tendremos que 0 <= sel(p) <= 1

- *Notar que la selectividad del predicado (A = c) para una relación R, con A clave candidata de R, es $1/T_R$ (y es la mejor selectividad que se puede obtener para una relación).*

A continuación veremos diversas implementaciones para las operaciones de SELECCIÓN, PROYECCIÓN y JOIN en forma individual, y más adelante analizaremos algunas cuestiones de su composición en planes de ejecución.

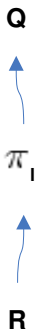
- *En el alcance de la materia no consideraremos cláusulas del estilo ORDER BY , GROUP BY ni funciones de agregación ni sentencias SQL anidadas.*

La idea general es dar una breve explicación de los algoritmos, analizando sus costos de procesamiento (considerando Costo de Input y Costo de Output), la cantidad de tuplas en el resultado, y eventualmente alguna característica del mismo.

- ***IMPORTANTE:** En el marco del proceso de optimización, no calcularemos el costo del output del resultado final (o sea, cuando se trate de operaciones que están en la raíz del árbol del plan de ejecución). Esto es porque el resultado que producen todos los planes es el mismo, por lo que el costo de ese output será igual en todos los casos. De todos modos, en los puntos que siguen hablaremos del costo del output para indicar cómo calcularlo en los casos que se necesite (ver sección 6. Planes de Ejecución).*

5.1 Proyección (π)

La operación de proyección **π (l, R, Q)** procesa las tuplas de una relación de entrada R y produce un resultado Q con todas las tuplas pero sólo conteniendo los atributos indicados en la lista l.



5.1.1 Características del resultado

5.1.1.1Cantidad de tuplas

En la materia no consideraremos el caso de eliminación de duplicados, por lo que la cantidad de tuplas del resultado siempre será igual a la cantidad de tuplas de la relación original.

Por lo tanto la cantidad de tuplas del resultado será

$$T_Q = T_R$$

5.1.1.2Longitud de tuplas

La longitud de las tuplas del resultado será la suma de las longitudes de los atributos incluidos en la selección. En caso de que no se sepa las longitudes de los campos, se asumirá que todos los campos tienen la misma longitud y por lo tanto será proporcional a la cantidad de atributos incluidos.

Si se conocen las longitudes de los atributos

$$L_Q = SUMA_{Ai\ in\ I}\ (L_{Ai})$$

Si no se conocen las longitudes de los atributos

$$L_Q = L_R * (Nº\ atributos\ de\ I\ /\ (Nº\ atributos\ de\ R))$$

5.1.1.3 Cantidad de bloques

La cantidad de bloques del resultado se calcula en función de la longitud de los bloques, la cantidad de tuplas y la longitud de cada tupla.

Calculemos primero el factor de bloqueo del resultado, es decir, cuantas tuplas del resultado entran por bloque:

$$FB_Q = [\ LB\ /\ L_Q\ ]$$ donde [ ] es parte entera.

La cantidad de bloques ocupados será

$$B_Q = [T_Q\ /\ FB_Q\ ]$$ donde [ ] es parte entera por exceso

5.1.1.4Costo del Output

Para el enfoque de la materia, el costo del output (CO) de una operación, en caso de tener que computarse, siempre es la cantidad de bloques necesarios para escribir el resultado en disco.

Por lo tanto, el costo del output será:

$$CO = B_Q$$

5.1.2 Algoritmo de Búsqueda Lineal en Archivo

Precondición: ninguna

Descripción: se recorren todos los bloques de la relación input (file scan) y se seleccionan los atributos de cada tupla.

Costo de input (CI):

El costo del input es $CI = B_R$

5.1.3 Algoritmo de Búsqueda Lineal en Índice B^+

Precondición: todos los atributos a seleccionar forman parte de la clave del índice

Descripción: se accede al primer nodo hoja (el de menor clave), se recorren todos los bloques de nodos hoja del índice y se seleccionan los atributos de cada entrada del índice.

Costo de input (CI):

Se suma el costo de acceder a los nodos no hoja necesarios, más todos los bloques hoja del indice

El costo del input es **$CI = X - 1 + BH_I$**

- *Notar que consideramos X – 1 porque es lo que cuesta encontrar el puntero al primer nodo hoja (hay que "bajar" en el árbol hasta su padre)*

5.1.4 Algoritmo de Búsqueda Lineal en Índice hash

Precondición: todos los atributos a seleccionar forman parte de la clave del índice

Descripción: Se recorren todos los bloques de todos los buckets del índice y se seleccionan los atributos de cada entrada del índice.

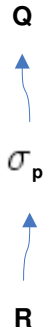
Costo de input (CI):

Se calculan en función de la cantidad máxima de bloques de índice por bucket.

El costo del input es **$CI = CBU_I * MBxB_I$**

5.2 Selección (σ)

La operación de selección **σ (p, R, Q)** procesa las tuplas de una relación de entrada R y genera un resultado Q con todas las tuplas que cumplen la condición p.



5.2.1 Características del resultado

5.2.1.1Cantidad de tuplas del resultado

Para estimar la cantidad de tuplas que satisfacen una condición asumiremos (salvo indicación en contrario) que la distribución de valores es uniforme y que los valores de los atributos son probabilísticamente independientes.

5.2.1.1.1Caso condición del tipo A = a

Si A no es clave candidata, se estimará como **$T_{Q =} T_R / I_{R,A}$**

- *Notar que si A es clave candidata, IR,A = TR, por lo que la cantidad de tuplas resultante es 1*

5.2.1.1.2Caso condición del tipo A = a and B = b

Se estimará como $T_Q = T_R / [I_{R,A} * I_{R,B}]$

En caso de haber más términos de igualdad para otros campos en la conjunción, se seguirá multiplicando el denominador por la imagen de cada campo.

Notar que la cantidad de tuplas del resultado es independiente del algoritmo de implementación.

5.2.1.1.3Caso condición del tipo A > a

Si hay noción de distancia sobre el atributo A, se estimará como:

$$T_Q = T_R * dist(a, A_{max}) / dist(A_{min}, A_{max})$$

donde A_{min} y A_{max} son los valores mínimo y máximo de la Imagen de A.

Si la noción de distancia no es aplicable sobre el atributo A, se estimará como:

$$T_Q = T_R / 2$$

5.2.1.1.4Caso condición del tipo $a_1 < A < a_2$

Si hay noción de distancia sobre el atributo A, se estimará como:

$$T_Q = T_R * dist(a_1, a_2) / dist(A_{min}, A_{max})$$

donde A_{min} y A_{max} son los valores mínimo y máximo de la Imagen de A.

Si la noción de distancia no es aplicable sobre el atributo A, se estimará como:

$$T_Q = T_R / 2$$

5.2.1.1.5Caso condición del tipo A IN $\{a_1, ..., a_i, ..., a_n\}$

Se estimará como $T_Q = T_R * n / I_{R,A}$

5.2.1.1.6

5.2.1.1.7Caso condición del tipo A = a or B = b

Se estimará como $T_Q = (T_R / I_{R,A}) + (T_R / I_{R,B}) - (T_R / (I_{R,A} * I_{R,B}))$

- *Recordar que estamos asumiendo probabilidades independientes y distribución uniforme de los valores de los diferentes atributos.*

- Se deja como ejercicio calcular la cantidad de tuplas de un predicado disyuntivo que tenga más de dos términos

5.2.1.1.8Caso general

En el caso general, estimaremos la cantidad de tuplas a partir de la selectividad, con la siguiente fórmula: $T_Q = sel(p) * T_R$

5.2.1.2Longitud de tuplas

La longitud de las tuplas del resultado será la misma que las de la relación input, por lo que queda

$L_Q = L_R$

5.2.1.3Cantidad de bloques

La cantidad de bloques del resultado se calcula en función de la longitud de los bloques, la cantidad de tuplas y la longitud de cada tupla.

El factor de bloqueo del resultado es el mismo que el de la relación input, ya que las tuplas son iguales:

$FB_Q = FB_R$

La cantidad de bloques ocupados será

$B_Q = [T_Q / FB_Q]$ donde [ ] es parte entera por exceso

5.2.1.4Costo del Output

Para el enfoque de la materia, el costo del output (CO) de una operación, en caso de tener que computarse, siempre es la cantidad de bloques necesarios para escribir el resultado en disco.

Por lo tanto, el costo del output será:

$CO = B_Q$

5.2.2 Algoritmo de Búsqueda Lineal en archivo

Precondición: ninguna

Descripción: se recorren todos los bloques del archivo (file scan) y se comprueba la condición en cada tupla

Costo de input (CI):

El costo del input es $CI = B_R$

- *Notar que este algoritmo , si bien es ineficiente, tiene la ventaja de que es aplicable a cualquier tipo de archivo, no tiene precondiciones*

5.2.3 Algoritmo de Búsqueda binaria en archivo ordenado

Precondición: se deben dar las siguientes condiciones simultáneamente:

- el archivo está ordenado según una clave k
- el predicado p coincide con el índice, o p es un predicado conjuntivo de la forma p1 and p2, donde p1 y p2 son dos predicados válidos, y p1 coincide con el índice
- p (o la subexpresión coincidente de p) determina una búsqueda por igualdad de clave o por un rango.

Descripción: se realiza búsqueda binaria hasta encontrar la primer tupla coincidente, y luego se recorre secuencialmente el archivo mientras las tuplas cumplan la condición de la subexpresión coincidente con el orden. En el caso de que p fuese un predicado conjuntivo según la precondición, para cada tupla obtenida en el paso anterior se debe verificar además que cumpla con el predicado p2. Si lo cumple, se agrega a la relación resultado. Notar que esto último no afecta el costo de input, ya que no involucra ningún acceso adicional a disco.

Costo de input (CI):

Sea p_k la subexpresión de p coincidente con la clave k, utilizada para obtener las tuplas candidatas a partir del archivo ordenado (si toda la expresión coincide, tenemos $p_k = p$).

El costo de la búsqueda binaria es $\log_2(B_R)$.

El costo del recorrido para las tuplas es

$[(sel(p_k) * T_R) / FB_R.]$ donde [ ] es parte entera por exceso

Por lo tanto, queda **$CI = \log_2(B_R) + [(sel(p_k) * T_R) / FB_R]$**

- *Notar que en caso de tratarse de una búsqueda por igualdad de una clave candidata, se simplifica la fórmula anterior, quedando* **$CI = \log_2(B_R)$**

Característica del resultado: la relación del output está ordenada de acuerdo a la misma clave que la relación R del input

5.2.4 Algoritmo de Búsqueda en índice árbol B⁺ clustered

Precondición: se deben dar las siguientes condiciones simultáneamente:

- el archivo tiene un índice árbol B^+ clustered según una clave k
- el predicado p coincide con el índice, o p es un predicado conjuntivo de la forma p1 and p2, donde p1 y p2 son dos predicados válidos, y p1 coincide con el índice
- p (o la subexpresión coincidente de p) determina una búsqueda por igualdad de clave o por un rango.

Descripción: Esto se resuelve accediendo según el índice, como se vio para búsquedas por igualdad o rango de índices arbol B^+ clustered. En el caso de que p fuese un predicado conjuntivo según la precondición, a cada tupla obtenida a partir del índice se debe verificar además que cumpla con el predicado p2. Si lo cumple, se agrega a la relación resultado. Notar que esto último no afecta el costo de input, ya que no involucra ningún acceso adicional a disco.

- *Si se tiene un índice B^+ clustered y el predicado coincide con ese índice y determina un rango, este algoritmo es el indicado para implementar la selección*

Costo de input (CI):

Sea p_I la subexpresión de p coincidente con I, utilizada para obtener las tuplas candidatas a partir del índice (si toda la expresión coincide, tenemos $p_I = p$).

La cantidad de bloques de índice a recorrer está dada por **$sel(p_I) * BH_I$**, queda

$CI = X + 1 + [((sel(p_I) * T_R) - 1) / FB_R]$ ([] por exceso)

- *El motivo de sumar 1 es por considerar el peor caso, asumiendo que la primer tupla está en un bloque y a partir de la segunda están en otro(s).*
- *Notar que en caso de tratarse de una búsqueda por igualdad de una clave candidata, se simplifica la fórmula anterior, quedando* **$CI = X + 1$**

Característica del resultado: la relación del output está ordenada de acuerdo a la misma clave que el índice utilizado

5.2.5 Algoritmo de Búsqueda en índice árbol B⁺ unclustered

Precondición: se deben dar las siguientes condiciones simultáneamente:

- el archivo tiene un índice I de tipo árbol B^+ unclustered según una clave k
- el predicado p coincide con el índice, o p es un predicado conjuntivo de la forma p1 and p2, donde p1 y p2 son dos predicados válidos, y p1 coincide con el índice
- p (o la subexpresión coincidente de p) determina una búsqueda por igualdad de clave o por un rango.

Descripción: Esto se resuelve accediendo según el índice, como se vio para búsquedas por igualdad o rango de índices arbol B^+ unclustered. En el caso de que p fuese un predicado conjuntivo según la precondición, a cada tupla obtenida a partir del índice se debe verificar además que cumpla con el predicado p2. Si lo cumple, se agrega a la relación resultado. Notar que esto último no afecta el costo de input, ya que no involucra ningún acceso adicional a disco.

Costo de input (CI):

Sea p_I la subexpresión de p coincidente con I, utilizada para obtener las tuplas candidatas a partir del índice (si toda la expresión coincide, tenemos $p_I = p$).

La cantidad de bloques de índice a recorrer está dada por

$1 + [((sel(p_I) * T_R) -1) / FB_I]$, ([] por exceso)

Entonces queda :

$CI = X + (1 + [((sel(p_I) * T_R) -1) / FB_I]) + sel(p_I) * T_R$

- *Notar que para el último término de la formula anterior se está asumiendo el peor caso, que es que todas las tuplas están en diferente bloque y siempre se requiera un acceso a disco para obtenerla*

Característica del resultado: la relación del output está ordenada de acuerdo a la misma clave que el índice utilizado

5.2.6 Algoritmo de Búsqueda en índice hash

Precondición: se deben dar las siguientes condiciones simultáneamente:

- el archivo tiene un índice I basado en hashing según una clave k
- el predicado p coincide con el índice, o p es un predicado conjuntivo de la forma p1 and p2, donde p1 y p2 son dos predicados válidos, y p1 coincide con el índice.
  - *Notar que p (o la subexpresión coincidente de p) determina una búsqueda por igualdad de clave.*

Descripción: Esto se resuelve accediendo según el índice, como se vio para búsquedas por igualdad en índices basados en hash. En el caso de que p fuese un predicado conjuntivo según la precondición, para cada tupla obtenida a partir del índice se debe verificar además que cumpla con el predicado p2. Si lo cumple, se agrega a la relación resultado. Notar que esto último no afecta el costo de input, ya que no involucra ningún acceso adicional a disco.

Costo de input (CI):

Sea p_I la subexpresión de p coincidente con I, utilizada para obtener las tuplas candidatas a partir del índice (si toda la expresión coincide, tenemos $p_I = p$).

Como la cantidad de tuplas a recorrer está dada por sel(p_I), tendremos que

$CI = MBxB_I + sel(p_I) * T_R$

5.2.7 Algoritmo de Intersección de Rids basado en hash

Precondición:

Se tiene un predicado conjuntivo $p = p_1 \wedge p_2$, tal que p no coincide con ningún índice, pero p_1 coincide con un índice I_1 y p2 coincide con un índice I_2

Descripción:

Se realiza sobre I1 la búsqueda de las tuplas que cumplen p1. Los rids de las tuplas encontradas se escriben en un archivo hash intermedio. Algo análogo se hace sobre I2 para las tuplas que cumplen p2, utilizando el mismo archivo hash.

Luego se recorre el hash y para cada rid encontrado que aparezca dos veces se accede al archivo de datos para obtener las tuplas

- *Este algoritmo puede extrapolarse a un predicado conjuntivo de la forma p1 ^... ^pi ^ ... ^ pn*

Costo de input (CI):

- Se deja como ejercicio, suponiendo que el hash intermedio entra completo en memoria.
  - *Notar que si el archivo hash entra en memoria, la operación sobre el hash tiene costo 0.*

- *En la materia sólo trabajaremos con expresiones en Forma Normal Conjuntiva*

5.2.8 Algoritmo de Unión de Rids basado en hash

Precondición:

Se tiene un predicado disyuntivo p = p_1 v p_2, tal que p no coincide con ningún índice, pero p_1 coincide con un índice I_1 y p2 coincide con un índice I_2

Descripción:

Se realiza sobre I1 la búsqueda de las tuplas que cumplen p1. Los rids de las tuplas encontradas se escriben en un archivo hash intermedio. Algo análogo se hace sobre I2 para las tuplas que cumplen p2, utilizando el mismo archivo hash. Sólo se agregan las tuplas que no están repetidas.

Luego se recorre el hash y para cada rid encontrado se accede al archivo de datos para obtener las tuplas

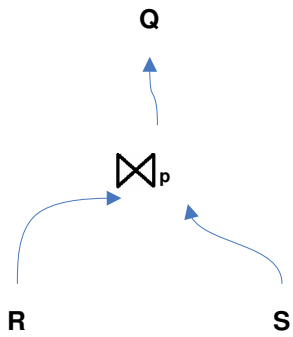
- *Este algoritmo puede extrapolarse a un predicado disyuntivo de la forma p1 v ... v pi v ... v pn*

Costo de input (CI):

- Se deja como ejercicio, suponiendo que el hash intermedio entra completo en memoria.
  - *Notar que si el archivo hash entra en memoria, la operación sobre el hash tiene costo 0.*
  - *En la materia sólo trabajaremos con expresiones en Forma Normal Disyuntiva*

5.3 Join (⋈)

La operación de Join ⋈ **(p, R, S, Q)** procesa las tuplas de dos relaciones de entrada R y S, y produce un resultado Q con todas las tuplas del producto cartesiano R X S que cumplen con la condición p.



En todo lo que sigue asumiremos que se trata siempre de equijoins, y la condición de junta está dada por $R.r_i = S.s_j$.

5.3.1 Características del resultado

5.3.1.1Cantidad de tuplas

Dada una condición de junta del estilo $R.r_i = S.s_j$, la cantidad de tuplas del resultado estará determinada por el atributo de la junta que tenga mayor imagen en la relación a la que pertenece.

Por lo tanto la cantidad de tuplas del resultado será

$$T_Q = (T_R * T_S) / \max(I_{R,ri}, I_{S,sj})$$

- *Notar que cuando ri es clave foránea de sj (i.e. sj es clave primaria de S), tendremos que $I_{R,ri} <= I_{S,sj} = T_S$.. Por lo tanto, en este caso TQ = TR; dicho con un ejemplo más intuitivo, cuando tenemos una relación muchos a uno, la cantidad de tuplas resultante de una junta natural es la cantidad de tuplas de la relación del lado "muchos"*

5.3.1.2Longitud de tuplas

La longitud de las tuplas del resultado será la suma de las longitudes de las tuplas de las relaciones de input. Queda así:

$$L_Q = L_R + L_S$$

- *En rigor de verdad, la longitud de las tuplas es $L_R + L_S - L'$, donde L' es la longitud de los campos comunes entre R y S considerados en la condición p (ya que los mismos no se repiten en el resultado).*

5.3.1.3 Cantidad de bloques

La cantidad de bloques del resultado se calcula en función de la longitud de los bloques, la cantidad de tuplas y la longitud de cada tupla.

Calculemos primero el factor de bloqueo del resultado, es decir, cuantas tuplas del resultado entran por bloque:

$$FB_Q = [LB / L_Q]$$

La cantidad de bloques ocupados será

$$B_Q = [T_Q / FB_Q]$$ ([] por exceso)

5.3.1.4Costo del Output

Para el enfoque de la materia, el costo del output (CO) de una operación, en caso de tener que computarse, siempre es la cantidad de bloques necesarios para escribir el resultado en disco.

Por lo tanto, el costo del output será:

$$CO = B_Q$$

5.3.2 Algoritmo Block Nested Loops Join (BNLJ)

Precondición: se tienen B bloques de memoria disponibles para la operación (B-2 bloques se utilizan para el almacenamiento de la relación R, 1 bloque se utiliza para ir leyendo los bloques de S, y el bloque restante se destina para la salida de la operación, en memoria).

Descripción:

```
Para cada segmento de B-2 bloques de R
    Para cada bloque de S
        Para toda tupla r del segmento de R y tupla s del bloque de S
            Si r_i == s_j ( o más generalmente, vale p(s) )
                Agregar <r,s> al resultado
            Fin si
        Fin para
    Fin para
Fin para
```

Costo de input (CI)

$B_R + B_S * [B_R/(B-2)]$

Explicación: Coloco en memoria B-2 bloques de B_R y recorro B_S entero. Esto lo hago $[B_R/(B-2)]$ veces. A su vez, estoy recorriendo todo R una sola vez, entonces debo sumar B_R. En este caso la notación de parte entera debe interpretarse como 'parte entera por exceso'.

5.3.3 Algoritmo Index Nested Loops Join (INLJ)

Precondición:

- el archivo tiene un índice I según una clave k
- el predicado p del join coincide con el índice I, o p es un predicado conjuntivo de la forma p1 and p2, donde p1 y p2 son dos predicados válidos, y p1 coincide con el índice.

Descripción

```
Para cada tupla r de R
    Para cada tupla s de S | r_i = s_j (obtenida según el índice de S)
        Si no hay condicion adicional o (hay y s la cumple )
            Agregar <r,s> al resultado
        Fin Si
    Fin para
Fin para
```

Costos de input (CI)

$B_R + \mathbf{T_R}$ *("buscar para la tupla t_R index entry/ies en índice de S" + "buscar valor/es apuntados por index entry/ies")

Explicación: Por cada bloque de R y por cada tupla dentro de ese bloque busco los index entries correspondientes en el índice de S y busco los valores apuntados. El costo depende casi íntegramente del tipo de índice.

Debemos tener en cuenta si el índice es clustered o unclustered, ya que en el primer caso accedemos a bloques con tuplas que cumplen la condición de junta, mientras que en el segundo caso tendremos un acceso por cada tupla que cumpla la condición.

5.3.4 Algoritmo Sort Merge Join (SMJ)

Precondición: ninguna

Descripción

Obs: Llamamos partición al conjunto de tuplas que poseen el mismo valor de cierto atributo

```
Si R no está ordenada en el atributo i
        ordenar R
Si S no está ordenada en el atributo j
        ordenar S

r = primera tupla de R     //itera sobre R
s = primera tupla de S     //itera sobre S

Mientras r ≠ null y s ≠ null   //cuando se llega al final de un archivo se obtiene null
        si r_i < s_j
                r = siguiente tupla en R   //No hay partición en S para r_i
        si r_i == s_j
                //Se encontró la partición en S para r_i...
                Mientras r_i == s_j    //Se procesa partición actual de R
                        inicio_part_s = s    //Marca el inicio de la partición en S para r_i
                        Mientras s_j == r_i    //Se procesa partición actual de S
                                Agregar <r,s> al resultado
                                s = siguiente tupla en S
                        Fin mientras

                      s = inicio_part_s
                        r = siguiente tupla en R
                Fin mientras
        si r_i > s_j
                s = siguiente tupla en S   //s no apuntaba al inicio de la partición en S de
```

r_i entonces sigo buscando Fin mientras

Costo de input (CI)

El costo de input para SMJ involucra el costo de ordenar cada una de las relaciones participantes por el atributo de junta (podría darse el caso que alguna de las relaciones ya estuviera ordenada). A esto debemos adicionarle el costo del *merge*, proceso en el cual se buscan en ambas relaciones ordenadas las tuplas que cumplen la condición de junta (es decir, las tuplas que pertenecen a la misma partición), luego de lo cual se realiza la junta de estas tuplas. El costo de merge es $B_R + B_S$, dado que se recorre una vez cada relación (consideramos que las particiones de S se recorren una sola vez para los casos en que el valor de R_i no se repite en R).

El costo de ordenar una relación (en este caso R) es: $([\log_{B-1}[B_R/B]]+1)*2B_R$

donde [] debe entenderse como parte entera por exceso;_

$[\log_{B-1}[B_R/B]]+1$ representa la cantidad de pasadas y esta cantidad se

multiplica por $2B_R$ porque en cada pasada leo y vuelvo a escribir cada bloque de R.

6 Planes de Ejecución

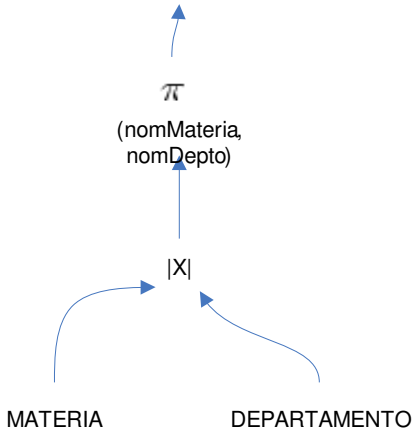
Hasta ahora vimos como evaluar el procesamiento de operaciones algebraicas individualmente, estimando sus costos de input y de output.

Como ya se dijo, en el marco del procesamiento y optimización de consultas primero se construyen representaciones algebraicas de las consultas a procesar. Estas representaciones tienen la forma de árbol de operaciones algebraicas.

Por ejemplo, la consulta

```
SELECT m.nomMateria,  d.nomDepto
FROM MATERIA m, DEPARTAMENTO d
WHERE m.cod_depto=d.cod
```

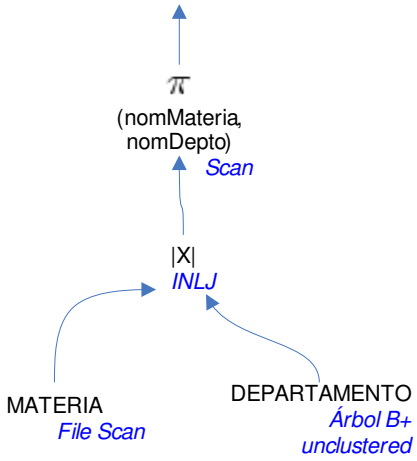
Podría representarse por el siguiente árbol (no es la única representación posible):



- Las hojas del árbol representan las relaciones origen de la consulta, en este caso MATERIA y DEPARTAMENTO
- Cada nodo no hoja representa las diferentes operaciones algebraicas que se van a realizar para arribar al resultado final, en este caso |X| y **π** .
- Los arcos de un nodo hacia sus sucesores representan las relaciones de input de esa operación, en este caso |X| tiene dos inputs, MATERIA y DEPARTAMENTO, en tanto que **π** tiene un input, MATERIA |X| DEPARTAMENTO.
- Los arcos de un nodo hacia sus antecesores representan la relación output de esa operación, en este caso, |X| tiene un output que es MATERIA |X| DEPARTAMENTO.
- El arco saliente del nodo raiz representa el resultado final. En este caso, el arco saliente de **π** representa la relación resultante de realizar **π** (nomMateria, nomDepto) (MATERIA |X| DEPARTAMENTO)

Cada operación algebraica tiene uno o más algoritmos u operadores físicos que la resuelven. Asimismo, cada tabla tiene uno o más métodos de acceso disponibles. Para cada nodo del árbol debemos indicar una de esas formas físicas concretas de resolverlo.

En nuestro ejemplo, suponiendo que tenemos disponible un indice arbol B^+ unclustered sobre DEPARTAMENTO.cod, un posible plan de ejecución sería el siguiente:

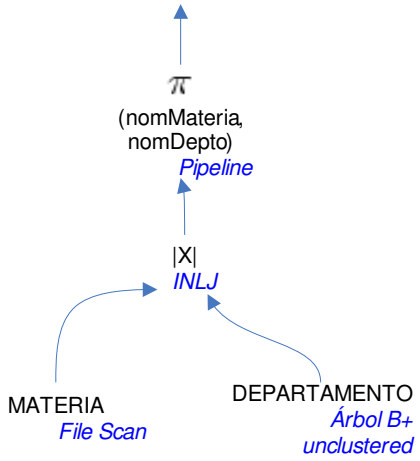


Para obtener los costos totales de ejecución de un plan, tenemos que sumar todos los costos de input y output de los diferentes nodos del mismo (con excepción, como ya se dijo, del output del nodo raíz, ya que es siempre el mismo para cualquier plan).

Para realizar el análisis de costos de un plan es importante tener en cuenta qué es lo que pasa con los resultados intermedios entre los nodos de cada árbol. Una posibilidad es **materializar** estos resultados, escribiéndolos a disco; en este caso tendremos que computar el costo de output de la operación previa y el costo de input de la siguiente operación. Otra posibilidad es intentar realizar la siguiente operación en forma *pipelinizada*, realizando ambas operaciones y recién luego escribiendo a disco; la primer operación le va pasando las tuplas resultados a la segunda a medida que las calcula, y la segunda las procesa directamente, sin necesidad de darles persistencia temporaria; en este caso, no tendremos que computar costo de output de la primer operación ni de input de la segunda, ya que no se realizan accesos a disco.

Un ejemplo donde esta mejora se puede ver en forma clara es en una selección seguida de una proyección. Tiene sentido que a medida que se seleccionan las tuplas de una de las relaciones, se vayan proyectando algunos atributos sin necesidad de escribir a disco el resultado intermedio.

En nuestro ejemplo anterior podríamos optar por “pipelinizar” el input de la raíz. Quedaría así:



La posibilidad de utilizar o no pipeline en la composición de operaciones la da el procesador de consultas (al igual que para cualquier operación física). Para poder realizar pipeline en una composición necesitamos que el procesador tenga implementada una operación física pipelinizada para dicha composición.

En el proceso de optimización siempre debemos analizar los distintos algoritmos que tenemos disponibles en cada operación. Es importante notar en este punto que los índices sólo están disponibles cuando se trata de la relación original. Es decir, una vez que realizamos una operación sobre una relación, el índice no está disponible sobre la relación resultado, ya que ésta queda en memoria o grabada en disco en un nuevo archivo temporario que no tiene índices.

Otra cosa a tener en cuenta es lo que se llama “órdenes interesantes”.

Un resultado intermedio está en un orden interesante si está ordenado de forma tal que:

- Alguna cláusula ORDER BY de un nivel superior del árbol coincide con ese orden
- Alguna cláusula GROUP BY de un nivel superior del árbol coincide con ese orden
- Alguna cláusula DISTINCT de un nivel superior está aplicada a atributos que coinciden con ese orden
- Algún join de un nivel superior puede hacer uso de ese orden para disminuir sus costos
  - *Recordar que en la práctica de la materia no trabajaremos con ORDER BY, ni con GROUP BY ni con DISTINCT*

En muchos casos, dentro de un plan global, puede ser más conveniente mantener un algoritmo que deje el resultado en un orden interesante (ya que será aprovechado por una operación posterior), aún cuando el costo local de esa operación no sea el mejor.

7 Optimización de Consultas

7.1 Optimizaciones Algebraicas

Las optimizaciones de este tipo son aquellas que buscan mejorar la performance de la consulta independientemente de la organización física. Involucran propiedades algebraicas que permiten construir una consulta equivalente a la original.

Algunas propiedades

Cascada de σ

$\sigma_{C1 \text{ and } C2 \ldots\ldots \text{and } Cn}(R) \equiv \sigma_{C1}(\sigma_{C2}(\ldots\sigma_{Cn}(R)\ldots))$

Conmutatividad de σ

$\sigma_{C1}(\sigma_{C2}(R)) \equiv \sigma_{C2}(\sigma_{C1}(R))$

Cascada de π

$\pi_{list1}(\pi_{list2}(R)) \equiv \pi_{list1 \cap list2}(R)$

Conmutatividad de σ con respecto a π

$\pi_{A1, A2, \ldots\ldots An}(\sigma_{C}(R)) \equiv \sigma_{C}(\pi_{A1, A2, \ldots\ldots, An}(R))$

Si C referencia solamente a atributos dentro de $A_1 \ldots A_n$

Comutatividad del Producto Cartesiano (o Junta)

$R \times S \equiv S \times R$

Conmutatividad del σ con respecto al Producto Cartesiano (o Junta)

$\sigma_{C}(R \times S) \equiv (\sigma_{Cr}(R)) \times (\sigma_{Cs}(S))$ donde $C = C_r \cup C_s$

Conmutatividad de la π con respecto al Producto Cartesiano (o Junta)

$\pi_{L}(R \times S) \equiv (\pi_{L1}(R)) \times (\pi_{L2}(S))$ donde $L = L_1 \cup L_2$

Conmutatividad de operaciones de conjuntos ($\cup$ e $\cap$)

$R \theta S \equiv S \theta R$

Si $\theta = \{\cup \text{ e } \cap\}$

Asociatividad del Producto Cartesiano, Junta, $\cup$ e $\cap$

$(R \theta S) \theta T \equiv R \theta (S \theta T)$

Si $\theta = \{x, |X|, \cup \text{ e } \cap\}$

7.2 Algunas heurísticas aplicables

Los optimizadores de consultas utilizan las reglas de equivalencia del álgebra relacional para mejorar, en la medida de lo posible, el rendimiento esperado de una consulta dada.

A continuación enumeramos algunas heurísticas aplicables:

a) **Considerar sólo árboles sesgados a izquierda**

Al analizar los posibles árboles candidatos, reducir el espacio de búsqueda considerando sólo árboles sesgados a izquierda, es decir, árboles donde los sucesores derechos de cualquier nodo sean hojas.

b) **Descomponer las selecciones conjuntivas** en una secuencia de selecciones simples formadas por cada uno de los términos de la selección original.

c) **Llevar las selecciones lo más cercano posible a las hojas del árbol**, de manera de lograr la ejecución temprana reduciendo así el número de tuplas que se propagan hacia niveles superiores.

d) **Reemplazar los productos cartesianos seguidos de selecciones por joins. Evitar en la medida de lo posible los productos cartesianos.**

Si se tiene un producto cartesiano seguido de una selección con un predicado p sobre atributos que determinan un join sobre ese predicado p, se descarta el árbol con las dos operaciones por separado. De esta manera se evita propagar resultados intermedios muy voluminosos.

e) **Descomponer las listas de atributos de las proyecciones y llevarlas lo más cercano posible a las hojas del árbol**, creando nuevas proyecciones cuando sea posible de manera de no propagar hacia niveles superiores atributos innecesarios. De esta manera se logra una reducción temprana del tamaño de las tuplas, y se reduce la cantidad de bloques necesaria para almacenamiento intermedio .

f) **Realizar primero los joins más selectivos**, de manera de reducir el tamaño de los resultados intermedios.

g) **Utilizar como outer (externas) las relaciones más selectivas**

Al planear los árboles, tender a utilizar como relación outer de los joins a aquellas que sean más selectivas, es decir, aquellas en que su selectividad sea menor.

h) **En cada nodo, retener los planes menos costosos, pero considerar también los órdenes interesantes**

Al analizar el costo de un nodo intermedio, retener para niveles superiores del árbol los subplanes de menor costo. En caso de que haya alguno cuyo resultado intermedio esté en algún orden interesante, retener además el de menor costo para ese orden.

i) **Tener en cuenta los índices interesantes al momento de generar los planes de ejecución.**

En muchas ocasiones puede ser importante no “bajar” al máximo las proyecciones y/o selecciones sobre una relación, de manera de poder aprovechar el índice de la relación en alguna operación de un nivel superior, y aplicar recién el filtro y/o la proyección luego de esta operación.

j) **Utilizar el pipeline entre las operaciones siempre que sea posible**, de manera de evitar los costos adicionales a causa de dar persistencia a disco a resultados intermedios en forma innecesaria.

- *En la práctica de la materia no trabajaremos con órdenes interesantes.*

7.3 Pasos para la optimización

Los pasos que seguiremos para la optimización de una consulta son los siguientes:

1. Construir el árbol canónico
2. Construir árboles equivalentes alternativos, utilizando alguna heurística para que no se sobredimensione innecesariamente el espacio de búsqueda del mejor plan
3. Para cada árbol construido, hacer tantos planes de ejecución como surjan de las diferentes combinaciones « interesantes » de reemplazar los operadores lógicos por operadores físicos o algoritmos, indicando en qué casos los resultados intermedios son pipelinizados y si algún resultado queda en un orden interesante
4. Para cada plan concreto de ejecución, evaluar sus costos totales
5. Elegir el plan que haya resultado con menor costo

7.4 Estrategia de Programación Dinámica

La estrategia según los pasos mencionados anteriormente puede ocasionar que se tengan que evaluar varias veces las mismas operaciones en el contexto de distintos planes de ejecución.

Se han desarrollado estrategias alternativas que apuntan a superar este problema, la más utilizada es la llamada estrategia de Programación Dinámica.

La idea básica de esta estrategia es, dado una consulta de n relaciones vinculadas por la operación de junta, dividir el análisis en n pasadas. La pasada i-ésima computará los costos de los planes de i relaciones. Cada pasada trabajará incrementalmente aprovechando los resultados generados por la pasada anterior, siguiendo las heurísticas correspondientes.

- *En la práctica de la materia no trabajaremos con la estrategia de programación dinámica.*

8 Ejemplos

8.1 Ejemplo 1

Dados los siguientes esquemas de relación,

Empleado (nro-e (4), nombre-e (30), domicilio (24), ciudad (10), dni (8), tel (12), sueldo (8))

Trabaja-en (nro-e, nro-d)

Departamento (nro-d (4), nombre-d (20), ubicación (20), ciudad (10)),

donde el número asociado a cada atributo representa la cantidad de bytes que ocupa, y considerando que,

Tamaño de bloque = 4.000 bytes

$T_{Empleado}$ = 100.000 tuplas

$T_{Trabaja\_en}$ = 130.000 tuplas

$T_{Departamento}$ = 50 tuplas

$I_{Empleado.ciudad}$ = 100

$I_{Departamento.ciudad}$ = 8

Existe un índice B+ clustered sobre nro-d en Departamento

Existe un índice B+ unclustered sobre ciudad en Departamento

Existe un índice B+ clustered sobre nro-e en Empleado

Existe un índice B+ clustered sobre nro-d en Trabaja-en

(en este ejemplo consideramos despreciable el costo de acceso a índices),

Bloques disponibles en memoria principal = 3,

optimizar la siguiente consulta en SQL y calcular el costo de su ejecución, analizando distintos planes de evaluación. Es necesario, además, conocer el costo de escribir el resultado de la consulta en disco.

```
SELECT    nro-e, nombre-e
FROM      Departamento D, Trabaja-en T, Empleado E
WHERE     D.nro-d = T.nro-d  and
          T.nro-e  = E.nro-e  and
          D.ciudad = “Mendoza”;
```

Resolución:

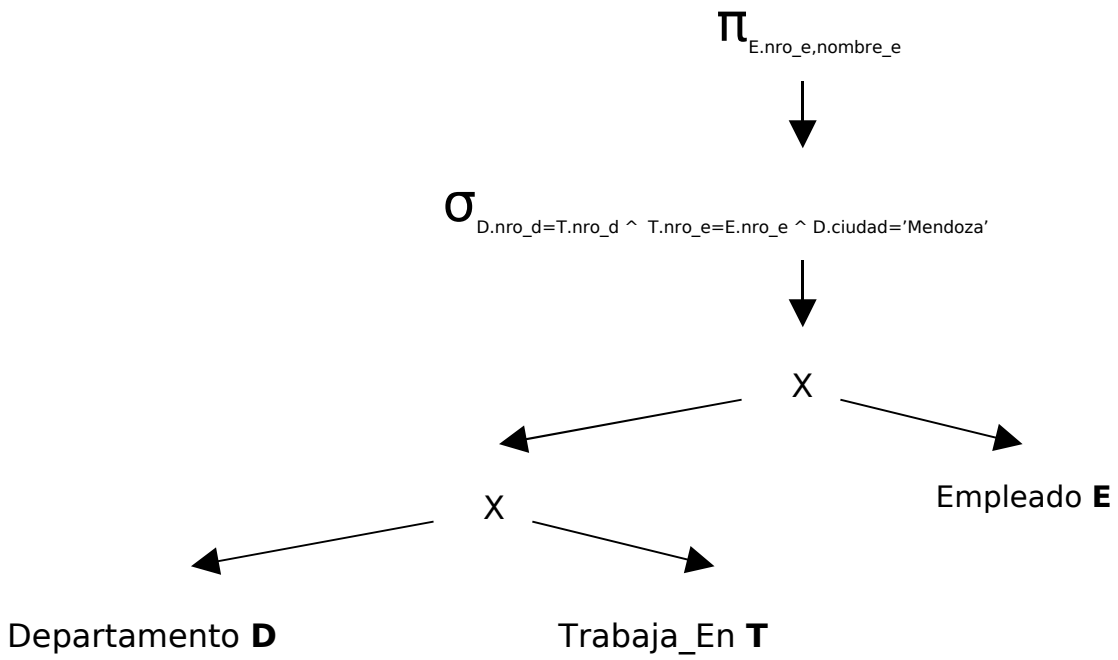
En primer lugar, optimizamos algebraicamente la consulta, y para esto, convertimos la expresión en SQL al Álgebra Relacional.

$$\pi_{E.nro\_e,nombre\_e}\left(\sigma_{\substack{D.nro\_d=T.nro\_d \wedge \\ T.nro\_e=E.nro\_e \wedge \\ D.ciudad='Mendoza'}}\left(D \;|X|\; T \;|X|\; E\right)\right)$$

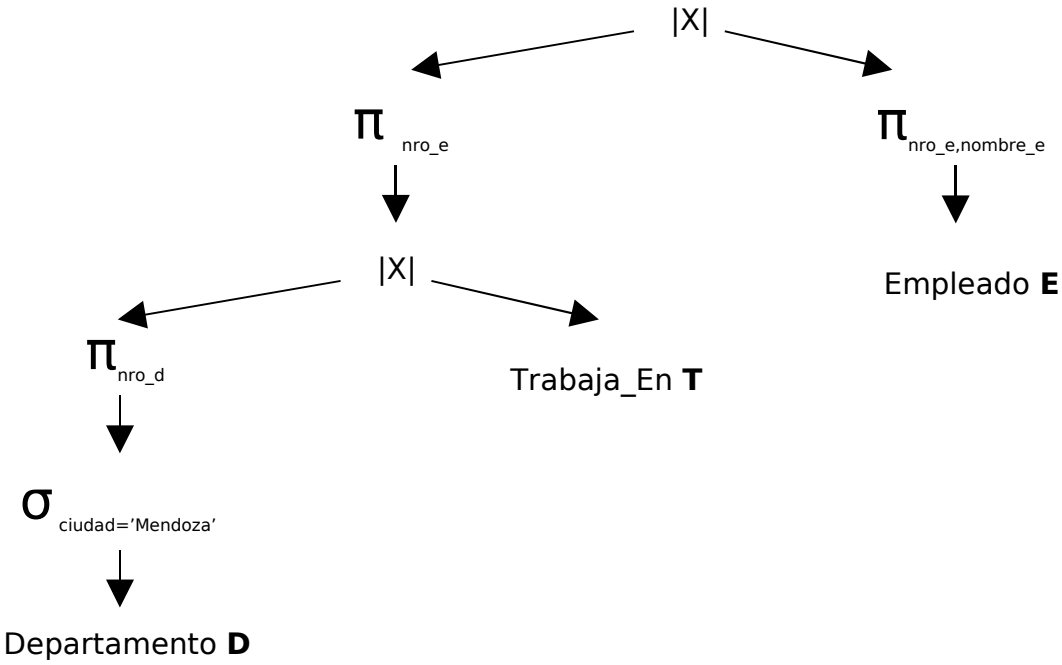
donde D representa Departamento, T representa Trabaja_En, y E Empleado.

A continuación se presenta el árbol canónico de la consulta y un árbol optimizado, que surge de aplicar las reglas de equivalencia algebraicas (se pueden obtener varios árboles optimizados a partir del árbol inicial).

Árbol Canónico



Árbol Optimizado



Una vez obtenidos los pasos para resolver la consulta (surgen del árbol optimizado), calcularemos el costo de ejecutarla, considerando distintos planes de resolución.

8.1.1.1Cálculos adicionales

- Longitud tupla Departamento:
  L_D = 4 by + 20 by +20 by + 10 by
  L_D = 54 by

- Bloques que ocupa Departamento:
  FB_D = [tam_bloque / L_D] = [4000 by/bloque / 54 by/tupla] =
  74 tuplas/bloque
  B_D = [T_D / FB_D] = [50 / 74] ([] por exceso)
  B_D = 0,67 ▪ **B_D = 1 Bloque**

- Longitud tupla Trabaja_En:
  L_T = 4 by + 4 by
  L_T = 8 by

- Bloques que ocupa Trabaja_En:

FB_T = [tam_bloque / L_T] = [4000 by/bloque / 8 by/tupla] = 500 tuplas/bloque

B_T = [T_T / FB_T] = [130000/500] ([] por exceso)

B_T = 260 Bloques

- Longitud tupla Empleado:

  L_E = 4 by +30 by + 24 by +10 by + 8 by +12 by + 8 by

  L_E = 96 by

- Bloques que ocupa Empleado:

  FB_E = [tam_bloque / L_E] = [4000 by/bl / 96 by/t]= 41 tuplas/bl

  B_E = [T_E / FB_E] = [100000 / 41] ([] por exceso)

  B_E = 2440 Bloques

8.1.1.2Costo de ejecución

1) Cálculo del costo $\sigma_{ciudad='Mendoza'}$(Departamento) [Cσ]

   a) Sin índice (file scan):

      Cσ = B_D

      Cσ = 1 acceso

   b) Utilizando el índice unclustered sobre ciudad en Departamento

      Veamos cuántas tuplas corresponderán a la ciudad de Mendoza.

      Considerando una distribución uniforme de los datos,

      $\mathbf{T}_{Departamento,\ ciudad='Mendoza'}$ = T_D / $I_{Departamento.ciudad}$

      = 50 tuplas / 8 = 6,25

      $T_{Departamento,\ ciudad='Mendoza'}$ = 7 tuplas

      Como estamos utilizando un índice unclustered, tendremos un acceso por cada tupla.

      Cσ = 7 accesos

Observamos claramente en este caso, que el uso del índice no nos es útil, y por lo tanto no lo utilizamos. Entonces, tomamos como costo de la selección el encontrado en a),

$C\sigma$ = 1 acceso

que constituye el costo de lectura. De esta manera queda definido *file scan* como camino o método de acceso a Departamento.

Como en memoria principal tenemos disponibles 3 bloques, uno de ellos lo utilizamos para la carga de la relación Departamento (recordar que ocupa 1 bloque).

2) Cálculo del costo de la junta

$\sigma_{ciudad='Mendoza'}$(Departamento) |X| Trabaja_En) [$C\sigma$ |X| T]

La junta se realizará por el atributo clave de la tabla resultante de la selección.

Como hay 3 bloques disponibles en memoria principal, utilizaremos:

- 1 bloque para Departamento que ya está en memoria
- 1 bloque para ir leyendo las tuplas de Trabaja_En
- 1 bloque para ir guardando el resultado de la junta

a) Sin índice: $C\sigma$ |X| T = B_D (ya leído) + B_T (a leer),

pues se compara cada bloque de Departamento con cada bloque de Trabaja_En. En este caso el método de junta utilizado es Block Nested Loops Join (BNLJ), cuando una de las relaciones participantes entra toda en memoria.

$C\sigma_{|X|\ T}$ = 260 accesos

b) Utilizando el índice de agrupamiento sobre nro_d en Trabaja_En

Estimemos la cantidad de empleados por departamento en Trabaja_En, como

$T_{Tdepto} = T_T / I_{Trabaja\_en.nro\_d}$ = 130000 tuplas / 50

T_{Tdepto} = 2600 tuplas/depto

La cantidad de valores distintos para el atributo nro_d en Trabaja_En es, como máximo, igual a la cantidad de tuplas en Departamento, ya que nro_d es clave de esta relación y clave foránea en Trabaja_en.

No olvidar que para el cálculo se consideró una distribución uniforme de los datos.

De esta forma, tendremos aproximadamente 2600 tuplas por cada departamento, en la relación Trabaja_En, agrupadas en

$B_{Tdepto} = [T_{Tdepto} / FB_T] =$ ([] por exceso)

$= [2600 \text{ tuplas/depto} / 500 \text{ tuplas/Bloque}] = [5,2]$

$B_{Tdepto} = 6 \text{ Bloque/depto}$

Luego, por cada departamento de la ciudad de Mendoza (resultado de la selección), se deberán realizar 6 accesos sobre Trabaja_En, para recuperar los empleados de ese departamento, con lo cual, utilizando el índice clustered, tendremos un costo de

$C\sigma_{|X|\,T} = T_{Departamento,\ ciudad='Mendoza'} * B_{Tdepto}$

$= 7 \text{ depto} * 6 \text{ Bloque/depto}$

$C\sigma_{|X|\,T} = 42$ accesos

Para resolver esta junta hemos utilizado el método Index Nested Loops Join (INLJ), descontando el costo de lectura de la relación externa (selección por ciudad='Mendoza'), pues ya estaba en memoria.

Finalmente elegimos la utilización del índice clustered sobre nro_d en Trabaja_En (INLJ) para resolver la junta, ya que disminuye la cantidad de accesos notablemente en comparación con la estrategia sin índices (BNLJ).

Debemos tener en cuenta, que en los puntos a) y b) el costo calculado fue el de lectura, con lo cual

$C\sigma_{|X|\,T\,(Lectura)} = 42$ accesos

3) Cálculo de la proyección $\pi_{nro\_e}(C\sigma\ |X|\ T)$

Si tuviéramos los suficientes bloques en memoria como para guardar el resultado de la junta, no habría costo de escritura. En nuestro caso, sólo

tenemos disponible 1 bloque de memoria. Veamos cuánto ocupa el resultado de la operación.

$$T_{\sigma |X| T} = T_{Departamento,\ ciudad='Mendoza'} * T_{Tdepto}$$
$$= 7 \text{ depto} * 2600 \text{ tuplas/depto}$$
$$T_{\sigma |X| T} = 18200 \text{ tuplas}$$

Observamos en el árbol optimizado que al resultado de la junta se le aplica una operación de proyección. Podríamos realizar esta operación a medida que se calculan las tuplas de la junta (pipeline). De esta forma, cada una de las tuplas en el resultado, estará formada por un único atributo, nro_e, y por lo tanto, la longitud de cada tupla será

$$L_{\pi(\sigma |X| T)} = 4 \text{ by}$$

y la cantidad de bloques que éstas ocupan será

$$FB\pi = [tam\_bloque / L_{\pi(\sigma |X| T)}] = [4000/4] = 1000 \text{ tuplas/bloque}$$
$$B_{\pi(\sigma |X| T)} = [T_{\sigma |X| T} / FB\pi] \quad ([] \text{ por exceso})$$
$$= [18200 \text{ tuplas} / 1000 \text{ tuplas/Bloque}] = [18,2]$$
$$B_{\pi(\sigma |X| T)} = 19 \text{ bloques}$$

con lo cual, guardaremos el resultado en disco (materialización) para ambas estrategias a) y b), con un costo de escritura asociado de

$$C_{\pi(\sigma |X| T)\ (Escritura)} = B_{\pi(\sigma |X| T)}$$

$$\mathbf{C_{\pi(\sigma |X| T)}\ (Escritura) = 19 \text{ accesos}}$$

Cabe aclarar que no estamos considerando valores duplicados en el resultado. Teniendo en cuenta que la imagen de nro_e en Empleado es 100000 (igual a la cantidad de tuplas ya que es atributo clave), y que la cantidad de tuplas resultado de la primera junta es 18200, en el peor caso corresponderán a valores distintos para el atributo nro_e.

Finalmente, el costo total de la primera junta es,

$$C_{\sigma |X| T} = C_{\sigma |X| T\ (Lectura)} + C_{\pi(\sigma |X| T)\ (Escritura)}$$
$$C_{\sigma |X| T} = 42 \text{ accesos} + 19 \text{ accesos}$$
$$\mathbf{C_{\sigma |X| T} = 61 \text{ accesos}}$$

4) Cálculo del costo de la segunda junta (C π(σ |X| T) |X| E)

Veamos cuánto ocupa cada una de las relaciones participantes:

- Resultado de la primera junta

  $B_{\pi(\sigma |X| T)}$ = 19 bloques

- Tabla Empleado

  B_E = 2440 bloques

Como hay 3 bloques disponibles en memoria principal, utilizaremos:
- 1 bloque para ir leyendo el resultado de la primera junta (que ha sido materializada)
- 1 bloque para ir leyendo las tuplas de Empleado
- 1 bloque para ir guardando el resultado de la junta

a) Sin índice: $C_{\pi(\sigma |X| T) |X| E} = B_{\pi(\sigma |X| T)} + B_{\pi(\sigma |X| T)} * B_E$,

pues se compara cada bloque resultado de la primera junta con cada bloque de Empleado. Utilizamos aquí la estrategia BNLJ, eligiendo como relación externa (es decir, se lee una sola vez) al resultado intermedio materializado $\pi(\sigma |X| T)$.

$C_{\pi(\sigma |X| T) |X| E}$ = 46379 accesos

b) Utilizando el índice de agrupamiento sobre nro_e en Empleado

Por cada tupla resultado de la primera junta, tendremos un acceso a la tabla Empleado, a través de su atributo clave nro-e, y del índice de agrupamiento (tener en cuenta que los nro-e en la tabla resultado de la primera junta no tienen por qué estar ordenados). En este caso estaríamos utilizando una estrategia INLJ.

$$C\ \pi(\sigma\ |X|\ T)\ |X|\ E = T\ \pi(\sigma\ |X|\ T)$$

$$\mathbf{C\pi(\sigma\ |X|\ T)\ |X|\ E = 18200\ accesos}$$

Observamos que la utilización del índice ha dado mejores resultados para la ejecución de la segunda junta.

Debemos tener en cuenta, que en los puntos a) y b) el costo calculado fue el de lectura, con lo cual

$$\mathbf{C\ \pi(\sigma\ |X|\ T)\ |X|\ E}\ (\text{Lectura}) = \mathbf{18200\ accesos}$$

que es el costo asociado a la segunda junta.

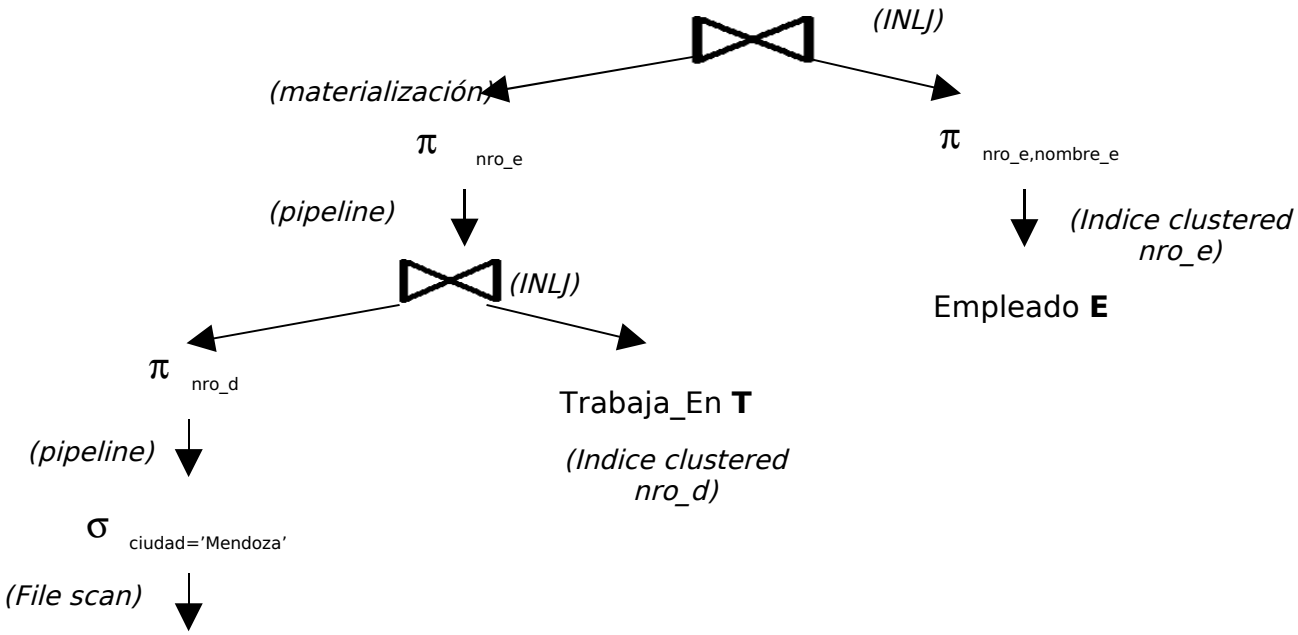
5) Cálculo del costo total de la consulta ($C_{consulta}$)

$$\mathbf{C_{consulta}} = C\sigma + C\sigma\ |X|\ T\ (\text{Lectura}) + C\ \pi(\sigma\ |X|\ T)\ (\text{Escritura}) + C\ \pi(\sigma\ |X|\ T)\ |X|\ E\ (\text{Lectura})$$

$$\mathbf{C_{consulta}} = 1\ \text{acceso} + 42\ \text{accesos} + 19\ \text{accesos} + 18200\ \text{accesos}$$

$$\mathbf{C_{consulta} = 18262\ accesos}$$

De esta manera, el plan de evaluación seleccionado para resolver la consulta es el siguiente:



6) Cálculo del costo de escritura del resultado final

Veamos cuánto ocupa el resultado de la consulta, para calcular el costo de escritura.

Como cada tupla de la tabla resultado de la primera junta, tendrá asociada una única tupla en Empleado, la cantidad de tuplas que resultan de la segunda junta serán

$$T_{\pi(\sigma_{|X|\ T)\ |X|\ E} = T_{\pi(\sigma_{|X|\ T)} = 18200 \text{ tuplas}$$

Esta vez, cada tupla estará formada por los atributos nro-e y nombre-e, con lo cual

$$L_{\pi(\sigma_{|X|\ T)\ |X|\ E} = 4 \text{ by} + 30 \text{ by}$$

$$L_{\pi(\sigma_{|X|\ T)\ |X|\ E} = 34 \text{ by}$$

ocupando, en total,

$$FB_{RES} = [tam\_bloque / L_{\pi(\sigma_{|X|T)|X|E}}] = [4000/34] = 117 \text{ tuplas/bloque}$$

$B\ \pi(\sigma\ |X|\ T)\ |X|\ E = [\ T\ \pi(\sigma\ |X|\ T)\ |X|\ E\ /\ FB_{RES}]$ ([] por exceso)

$= [18200 \text{ tuplas} / 117 \text{ tuplas/Bloque}] =$

$B\ \pi(\sigma\ |X|\ T)\ |X|\ E = 156$ bloques

Entonces, el costo de escritura asociado es de

$C\ \pi(\sigma\ |X|\ T)\ |X|\ E\ (\text{Escritura}) = B\ \pi(\sigma\ |X|\ T)\ |X|\ E$

$\mathbf{C_{\pi(\sigma\ |X|\ T)\ |X|\ E\ (Escritura)} = 156 \text{ accesos}}$

- PARA RESOLVER: Definir otro plan de evaluación de la consulta, tratando de disminuir el costo calculado en el ejemplo (si es necesario puede considerar otros índices).

8.2 Ejemplo 2

Esquema

Libros (ISBN, Titulo, tipo, precio)

Ventas (cod_lib, nro_venta, fecha_venta, ISBN)

Librerias(cod_lib, nombre_lib, telefono, dir_lib, ciudad, provincia)

Datos

Todos los campos de todas las tablas tienen una longitud de 128 bytes.

La longitud del bloque (LB) es 2048 bytes.

Hay 3 bloques de memoria.

Las tablas contienen la siguiente cantidad de registros:

- Librerías: 200 registros
- Libros: 2.000 registros
- Ventas: 200.000 registros

Se tiene un índice secundario **I1** de tres niveles para la tabla Librerías (x=3) según la clave primaria

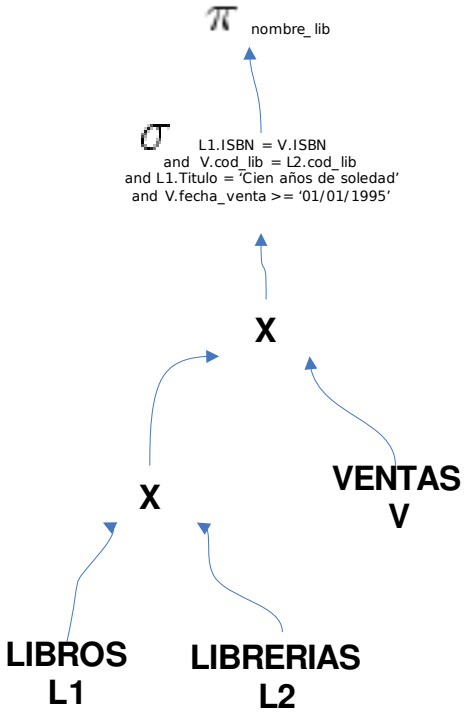
Se asume que:
- Los títulos de los libros no se repiten
- El 50% de los registros corresponden a ventas desde año 1995
- Se asume distribución uniforme de los datos

Consulta

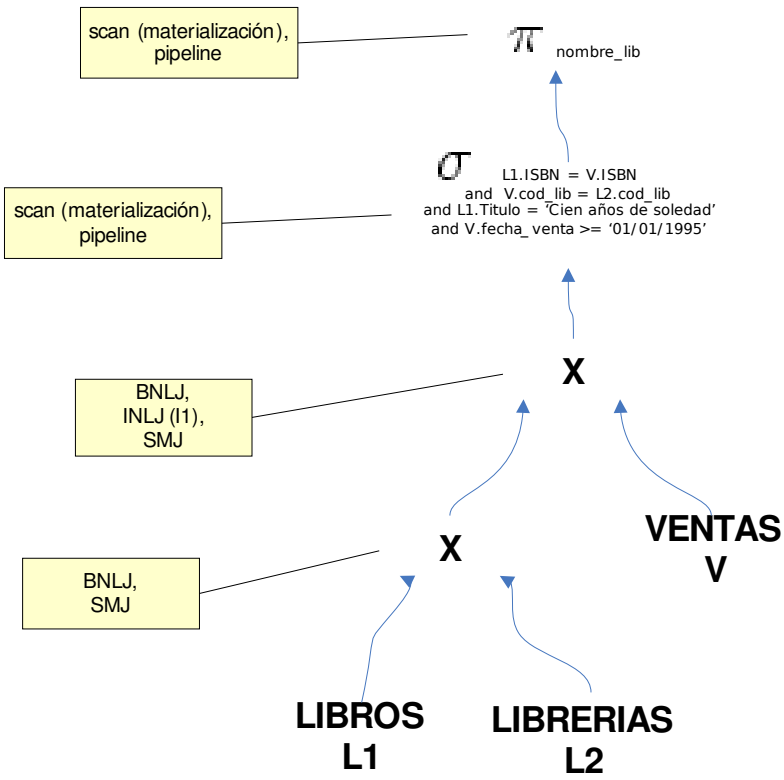
Las librerías que vendieron la publicación 'Cien años de soledad' desde el año 95.

```
SELECT  nombre_lib
FROM  Libros L1, Librerias L2, Ventas V
WHERE       L1.ISBN = V.ISBN  AND
            V.cod_lib = L2.cod_lib AND
           L1.Titulo = 'Cien años de soledad'  AND
            V.fecha_venta >= '01/01/1995'
```

Arbol canónico

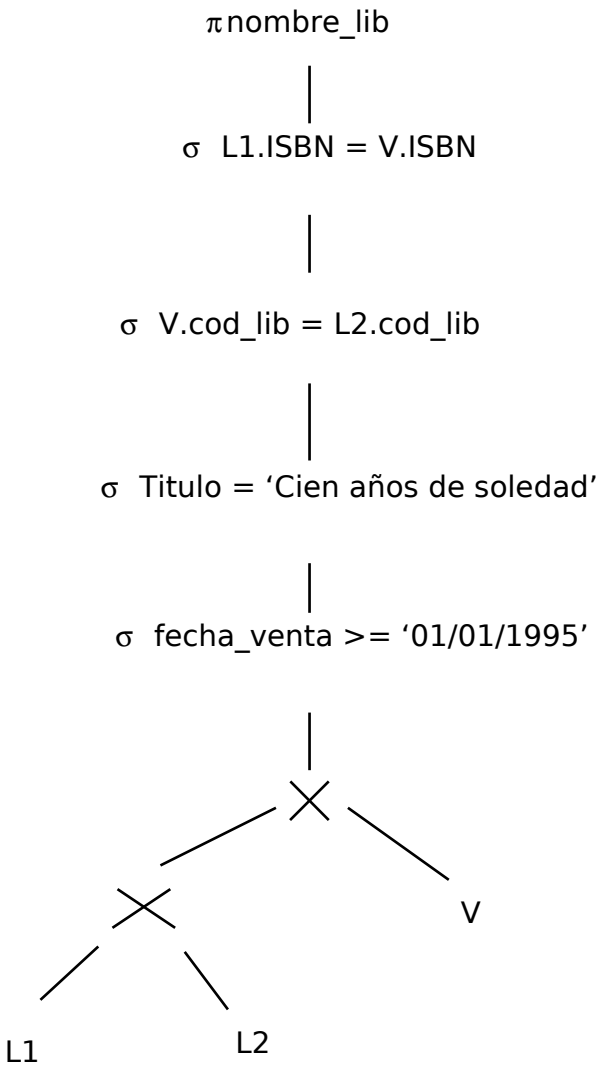


Operaciones físicas disponibles:

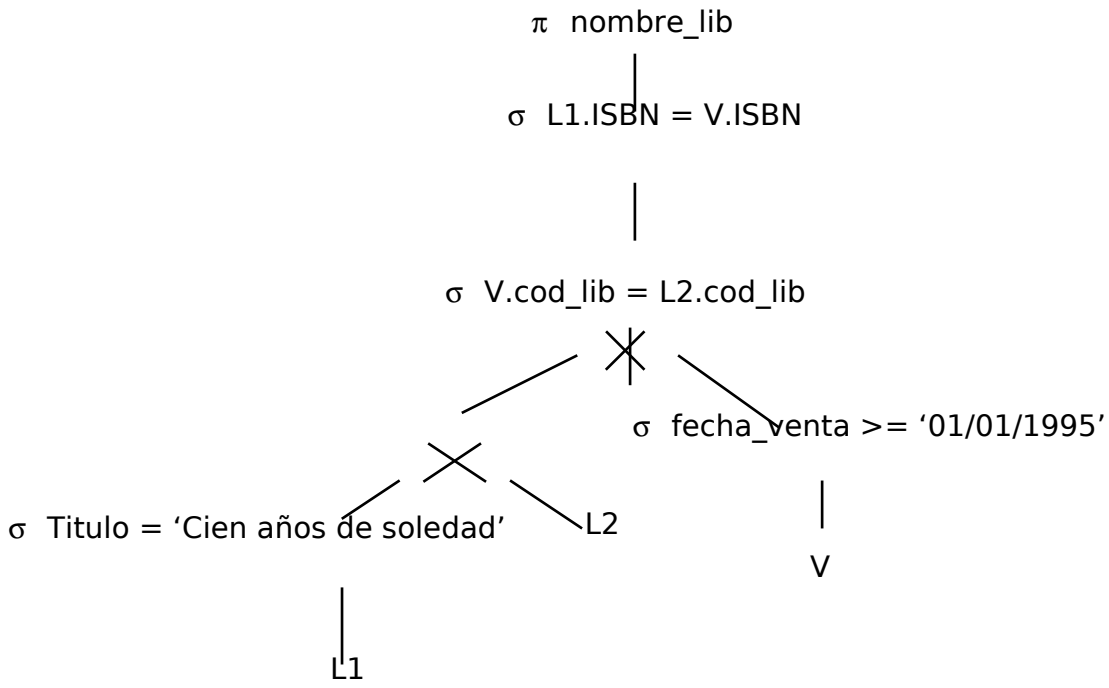


Optimización del árbol

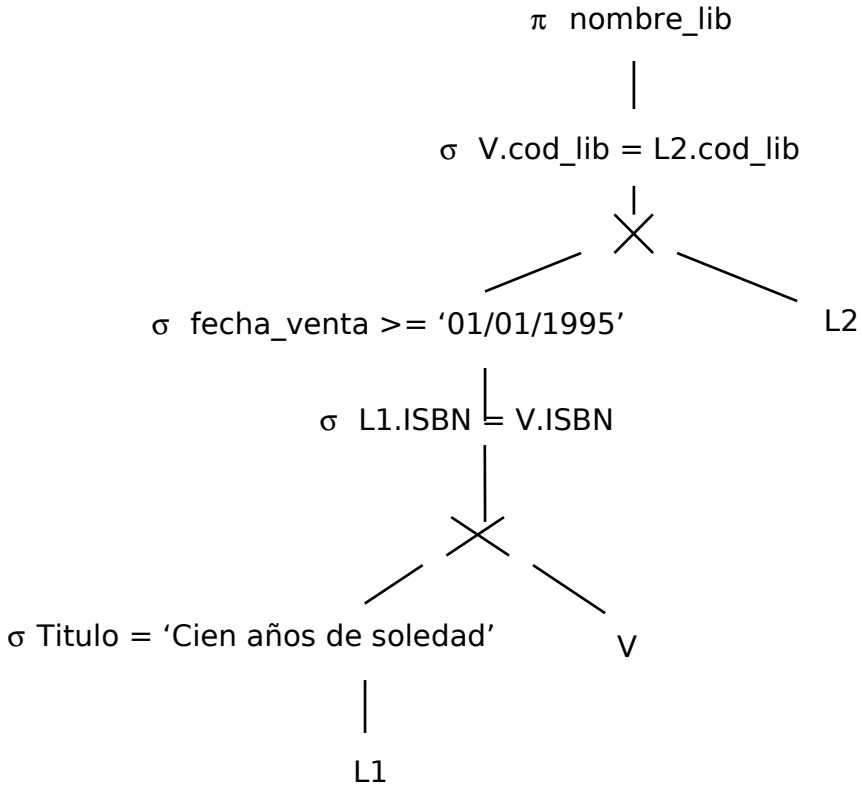
Paso 1) Descomponer las selecciones



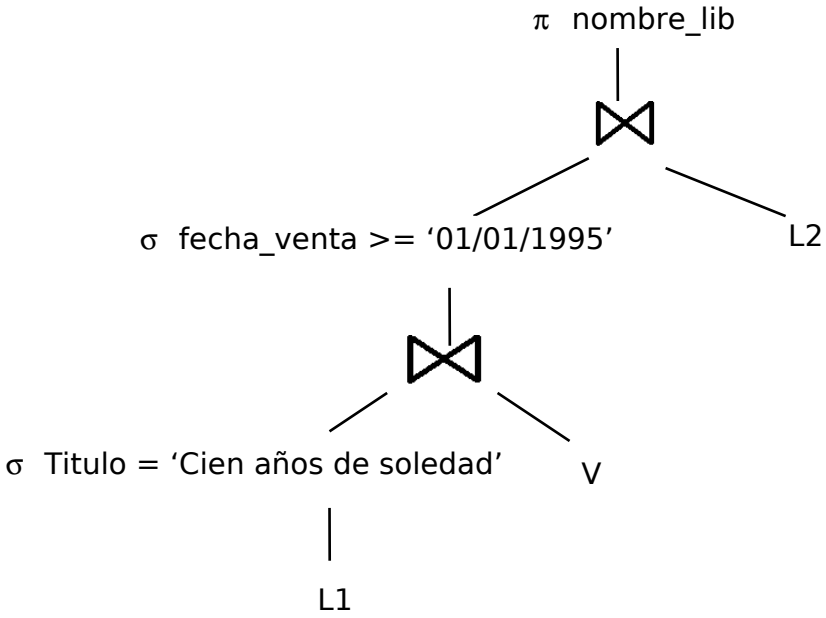
Paso 2) Llevar las selecciones hacia las hojas



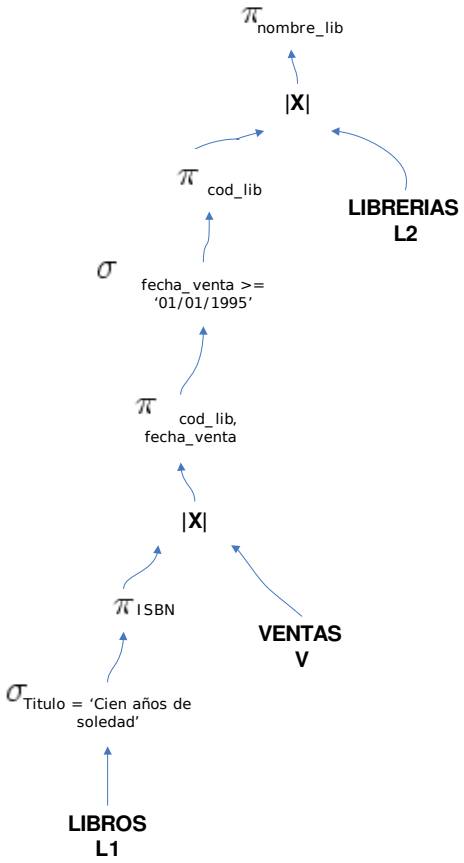
Paso 3) Permutar las relaciones de las hojas para evitar productos cartesianos, tratando de realizar primero las operaciones más selectivas.



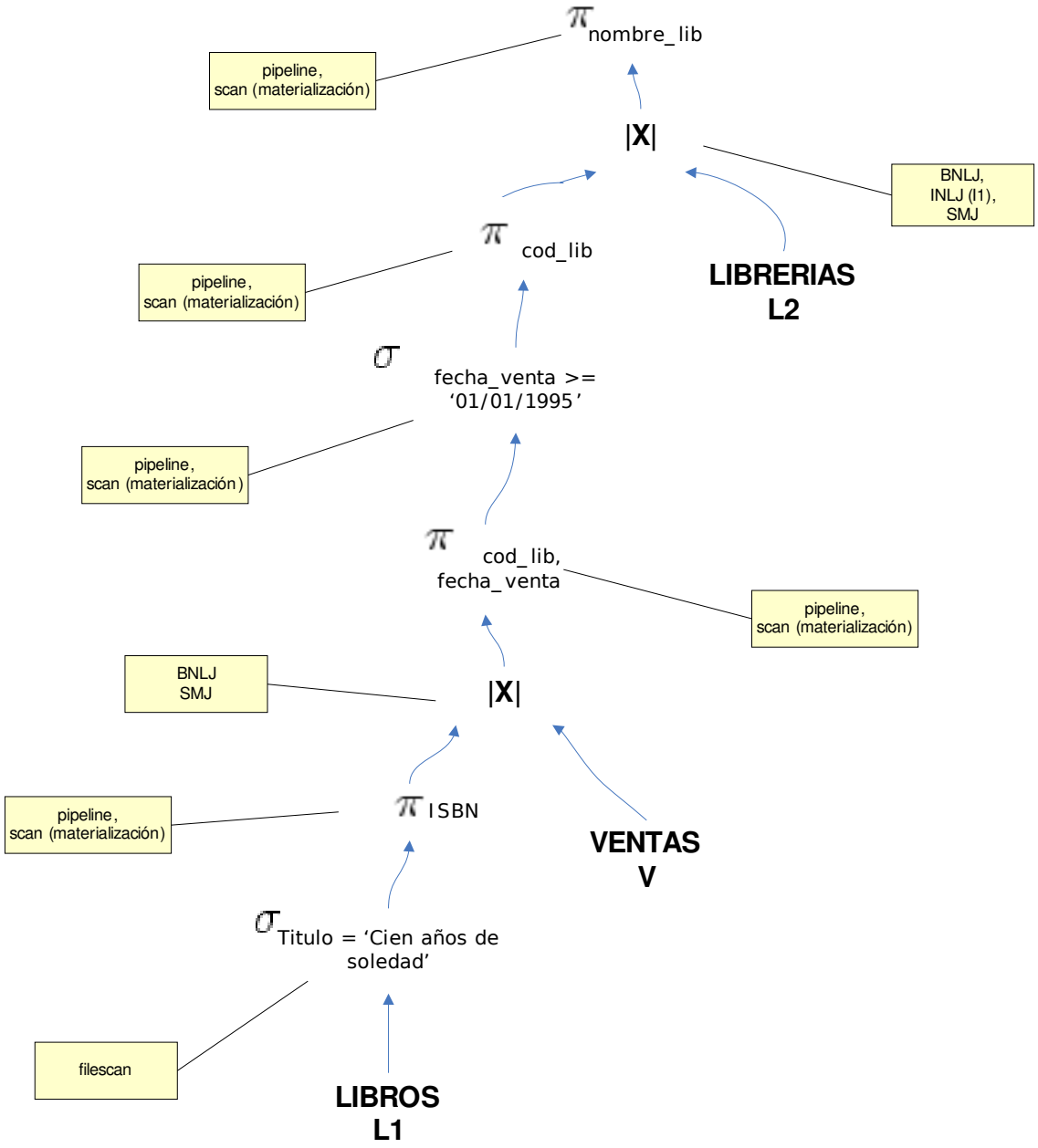
Paso 4) Fusionar en join los productos cartesianos seguidos de selecciones que determinan condición de join



Paso 5) Agregar proyecciones que restrinjan los atributos que se propagan a la mínima expresión necesaria



Operaciones físicas disponibles:

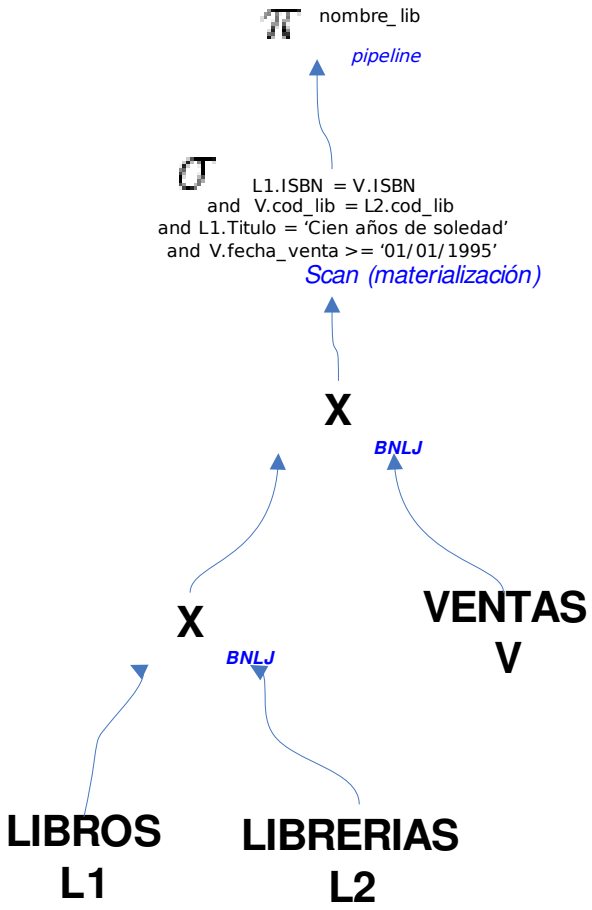


Como vimos, hay varios árboles posibles, nosotros generamos sólo dos en este caso, el canónico y uno optimizado.

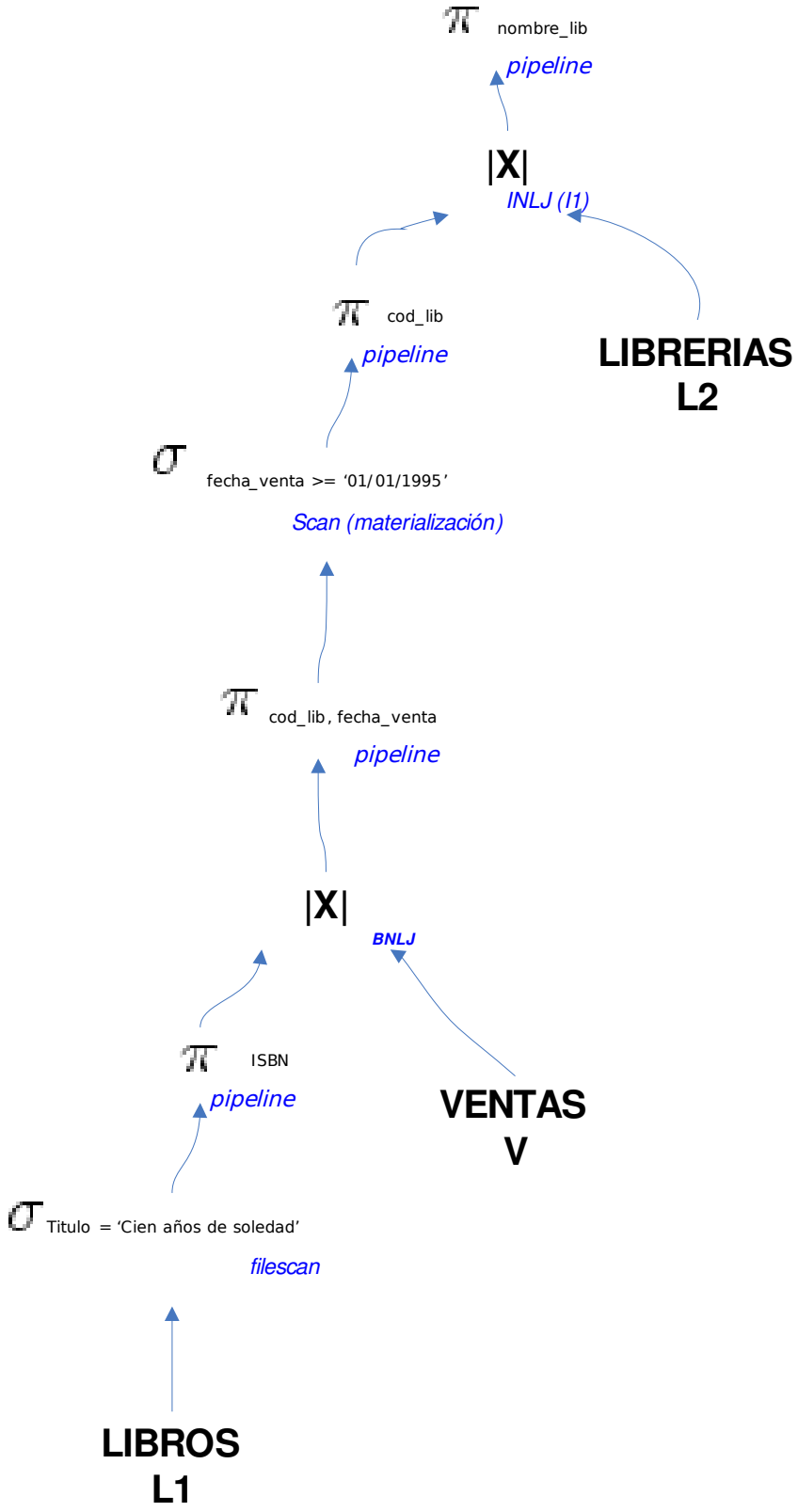
Para cada uno de esos dos árboles, podemos tener varias combinaciones de operaciones físicas disponibles para cada operador algebraico. Las distintas combinaciones válidas de las distintas operaciones físicas disponibles, nos permitirían generar los diferentes planes de ejecución a fin de calcular los costos.

Aquí vamos a elegir dos planes de ejecución, uno basado en el árbol canónico, y otro basado en nuestro árbol optimizado.

Llamaremos **Plan de Ejecución 1** (PE1) al siguiente:



Llamaremos **Plan de Ejecución 2** (PE2) al siguiente:



Cálculo de costos

Cálculo de datos generales:

Libros(L1):

L_{L1} = cantidad_campos * tam_campo = 128 * 4 = 512

FB_{L1} = [tam_bloque / longitud_tupla] = 2048 / 512 = 4

T_{L1} = 2.000

B_{L1} = [cantidad_registros / factor_bloqueo_relación] = 2.000 / 4 = 500

Librerías(L2):

L_{L2} = 128 * 6 = 768

FB_{L2} = [2048 / 768] = 2

T_{L2} = 200

B_{L2} = 200 / 2 = 100

Ventas(V):

L_V = 128 * 4 = 512

FB_V = 2048 / 512 = 4

T_V = 200.000

B_V = 200.000 / 4 = 50.000

Características del resultado

Sea Q la relación resultado

$\mathbf{L_Q}$ = 128 * 1 = 128

$\mathbf{FB_Q}$ = LB / L_Q = [2048 / 128] = 16

Ahora, estimemos los registros devueltos. Para ello usaremos algunos cálculos auxiliares:

Llamemos p al predicado de la selección. Podemos observar que p es de la forma (p_{L1} ^ p_V ^ pL1,V ^pL2,V), donde p_R denota un predicado que afecta sólo a atributos de R y $p_{R,S}$ sólo a atributos de R y de S.

Utilizando operaciones algebraicas podremos ver que

$\sigma_{P}(L1 \mathbf{X} L2 \mathbf{X} V) = (\sigma_{PL1}(L1) \mathbf{|X|}_{PL1,V} \sigma_{PV}(V)) \mathbf{|X|}_{PL2,V} L2$

Calculemos por partes:

Sea Q1 = $\sigma_{PL1}(L1)$

$T_{Q1} = T_{L1}/I_{L1,TITULO}$ = 2000 / 2000 = 1

Sea Q2 = $\sigma_{PV}(V)$

TQ2 = sel(p_v) * TV = 0.5 * 200000 = 100000

Sea Q3 = Q1 $|\mathbf{X}|_{\mathbf{PL1,V}}$ Q2

T_{Q3} = (T_{Q1} * T_{Q2}) / max ($I_{Q1,ISBN}$, $I_{Q2.ISBN}$)

= (1 * 100000) / max (1, 2000) = 50

Ahora queda Q = Q_3 $|\mathbf{X}|_{\mathbf{PL2,V}}$ L_2

T_Q = (T_{Q3} * T_{L2}) / max(I_{Q3,cod_lib}, I_{L2,cod_lib})

{como Q3.cod_lib es FK ▪ $I_{L2.codlib}$ >= $I_{Q3.cod\_lib}$}

= (50 * 200) / 200 = 50

Veámoslo informalmente:

Usando que están distribuidos uniformemente, asumimos #ventas / #libros da la cantidad de ventas que corresponden a cada libro (por lo tanto, 'Cien años de soledad' tiene esa cantidad de ventas).

Como se supone que la mitad de las ventas son desde el 95 (dato), la cantidad anterior la divido por 2.

(#ventas/#libros) / 2 = (200.000 / 2.000) / 2 = 50

Por lo tanto

$\mathbf{T_Q}$ = 50

$\mathbf{B_Q}$ = T_Q / FB_Q = 50 / 16 = 4

Calcularemos los costos sobre los dos planes de ejecución elegidos

Costo del plan elegido sobre el árbol canónico:

Paso 1) (L1 x L2)

Block Nested Loops Join

CI (costo de input) = B_{L1} + B_{L2} * B_{L1} = 500 + 100*500 = 50.500

Llamemos L' = L1 x L2 y calculemos sus datos:

$L_{L'}$ = 128 * (4+6) = 1280

$FB_{L'}$ = [2048 / 1280] = 1

$T_{L'}$ = 2.000 * 200 = 400.000

$B_{L'}$ = 400.000 / 1 = 400.000

CO (costo de output) = $B_{L'}$

CT_{paso1} = CI + CO = 450.500 accesos

Paso 2) L' x Ventas

Block Nested Loops Join

CI = $B_{L'}$ + B_V * $B_{L'}$ = 400.000 + 400.000 * 50.000 = 2.000.400.000

Llamemos L'' = L' x V y calculemos sus datos:

$L_{L''}$ = 128 * (10+4) = 1792

$FB_{L''}$ = [2048 / 1792] = 1

$T_{L''}$ = 400.000 * 200.000 = 80.000.000.000

$B_{L''}$ = 80.000.000.000/ 1 = 80.000.000.000

CO = $B_{L''}$

CT_{paso2} = CI + CO = 82.000.400.000 accesos

Paso 3) Selección

File scan

CI = $B_{L''}$

Llamemos L''' a $\pi_{nombre\_lib}(\sigma_{...}(L''))$ (aquí la proyección la hacemos "on the fly" y no como otro paso separado).

Calculemos los datos de L''':

$L_{L'''}$ = 128 * 1 = 128

$FB_{L'''}$ = 2048 / 128 = 16

$T_{L'''}$ = 50 (es el valor que calculamos antes)

$B_{L'''}$ = [50/16] = 4

CO = 4

Como es la operación de la raíz, no computaremos el costo del output

CT_{paso3} = CI + CO = 80.000.000.000 accesos

Si sumamos los costos de los 3 pasos obtenemos

COSTO TOTAL = 162.000.850.504 accesos

Costo del plan elegido sobre el árbol optimizado

Paso 1) Selección por título

File scan

CI = B_{L1} = 500

Devuelve un sólo registro. Además, proyecto el ISBN "on the fly".

Llamemos L' = $\pi_{ISBN}(\sigma_{...}(L1))$ y calculemos sus datos:

$L_{L'}$ = 128 * 1 = 128

$FB_{L'}$ = 2048 / 128 = 16

$T_{L'}$ = 1

$B_{L'}$ = 1

CO $= B_{L'} = 1$

CT_{paso1} = CI + CO = 501 accesos

Paso 2) L' x Ventas

Block Nested Loops Join. Luego se proyectan cod_lib y fecha_venta "on the fly", ya que elegimos pipeline para esa operación.

Tal como hicimos en el cálculo de costos del árbol canónico, usamos que están distribuidos uniformemente y asumimos que #ventas / #libros da la cantidad de ventas que corresponden a cada libro.

CI **=** $B_{L'} + B_V * B_{L'} = 1 + 1* 50.000 = 50.001$

Llamemos $L'' = \pi_{cod\_lib, fecha\_venta}(L' \times V)$ y calculemos sus datos:

$L_{L''} = 128 * 2 = 256$

$FB_{L''} = 2048 / 256 = 8$

$T_{L''}$ = #ventas / #libros = 200.000/2.000 = 100

$B_{L''} = [100/8] = 13$

CO $= B_{L''} = 13$

CT_{paso2} = CI + CO = 50.014 accesos

Paso 3) Selección por fecha

File scan. Se proyecta sólo cod_lib.

Se queda con la mitad de los registros del paso anterior (dato)

CI **=** $B_{L''} = 13$

Llamemos $L''' = \pi_{cod\_lib}(\sigma_{...}(L''))$ y calculemos sus datos:

$L_{L'''} = 128 * 1 = 128$

$FB_{L'''} = 2048 / 128 = 16$

$T_{L'''} = T_{L'''} /2 = 100/2 = 50$

$B_{L'''} = [50/16] = 4$

CO $= B_{L'''} = 4$

CT_{paso3} = CI + CO = 17 accesos

Paso 4) L''' x Librería

Index Nested Loops Join, usando el índice de la clave primaria de librería. Luego, se vuelve a proyectar un atributo "on the fly".

La cantidad de tuplas resultado se mantiene ya que se hace el join sólo para recuperar el nombre de la librería.

CI = $B_{L'''} + T_{L'''}(X_{I1}+1) = 4 + 50\ (3 + 1) = 204$

Por cada tupla de L''' se va a buscar un index entry dentro del índice I1 de L2. Como es un índice primario, habrá sólo un index entry y se deberá buscar un solo bloque de L2 para traer la librería correspondiente.

Llamemos L'''' = $\pi_{nombre\_lib}(L''' \times L2)$ y calculemos sus datos:

$L_{L''''} = 128 * 1 = 128$

$FB_{L''''} = 2048 / 128 = 16$

$T_{L''''} = T_{L'''} = 50$

$B_{L''''} = [50/16] = 4$

CO $= B_{L''''} = 4$

CT_{paso4} = CI + CO = 208 accesos

Si sumamos los costos de los 3 pasos obtenemos

COSTO TOTAL = 50740 accesos

Como vemos, el costo total del plan elegido del árbol canónico es más de 3 millones de veces más costoso que el del plan elegido del árbol optimizado.

Para hacerlo más gráfico, si suponemos que tenemos capacidad para realizar 1 acceso a disco cada microsegundo, y que toda otra operación consume tiempo 0, la ejecución de la consulta con el plan elegido sobre el árbol optimizado tardaría **0.05 segundos**, en tanto que con el plan elegido sobre el árbol canónico tardaría **45 horas!!!**

9 Bibliografía

- Database Management Systems – R. Ramakrishnan; J. Gehrke – Mc Graw-Hill - 3ª edición
- Principles of Database and Knowledge Base Systems, J. Ullman; Computer Science Press, 1988
- Fundamentos de Bases de Datos – A. Silberschatz; H. Korth; S. Sudarshan – Mc Graw-Hill – 3ª Edición
- Material del curso “Architecture and Implementarion of Database Management Systems” – Prof Dr. Marc . Scholl – University of Konstantz – Winter 2006-2007
- Material del curso “15-415 - Database Applications” – C. Faloutsos – Carnegie Mellon University
- An Overview of Query Optimization in Relational Systems – Surajit Chaudhuri - Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - 1998

Este apunte fue redactado por Diego Castro, Sergio D’Arrigo y Leticia Seijas, con la colaboración de Cecilia Briozzo y Alejandro Eidelsztein