FCEyN UBA

Bases de Datos

Procesamiento y Optimización de Consultas

Índice

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)

- **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).
- **$MBxB_I$**: 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 Indices

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

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

4.2.1. 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:

$\mathbf{B_r}$

4.2.2. 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:

$\mathbf{B_r}$

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

4.2.3. 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:

$\mathbf{B_r}$

4.3. Sorted File

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.

4.3.1. 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:

$\mathbf{B_r}$

4.3.2. 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:

$\boldsymbol{log_2}(\mathbf{B_r}) + \mathbf{B'}$

donde $\mathbf{B'}$ 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:

$$\boldsymbol{log_2(B_r) + \lceil T'/FB_r \rceil}$$

donde la notación de parte entera significa '*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*** $\boldsymbol{log_2(B_r)}$

4.3.3. 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. Por lo cual el costo de búsqueda es:

$$\boldsymbol{log_2(B_r) + \lceil T'/FB_r \rceil}$$

donde T' es la cantidad de tuplas que cumplen con el criterio de búsqueda.

4.4. Índices

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***

4.6. Índices Arbol B^+ clustered

Los índices de tipo árbol $\boldsymbol{B^+}$ 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 $\boldsymbol{B^+}$ **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.

4.6.1. Costo de exploración completa

En este caso el índice no nos ayuda, con lo cual tendremos que usar otra forma de acceder al archivo (por ejemplo, file scan).

4.6.2. 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 $\mathbf{X_I}$ 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:

$\mathbf{X_I + B'}$

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:

$$\mathbf{X_I} + \lceil \mathbf{T'/FB_R} \rceil$$

> ✍ ***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*** $\mathbf{X_I} + 1$.

4.6.3. 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. Entonces la fórmula al igual que en el punto 4.6.2 tendremos que el costo de acceso es:

$$\mathbf{X_I} + \lceil \mathbf{T'/FB_R} \rceil$$

donde T' es la cantidad de tuplas que cumplen con el criterio de búsqueda. Las tuplas resultantes quedan ordenadas según el atributo A.

4.7. Índices Arbol B^+ unclustered

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.

4.7.1. Costo de exploración completa

Al igual que en el índice clustered, el índice no nos ayuda para este caso.

4.7.2. 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 seguirán leyendo los *rids* del nodo hoja y, eventualmente si se consumen todos los rids de ese nodo, se pasará a leer los del próximo nodo hoja (aprovechando que están doblemente enlazados). 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). Si bien puede suceder que 2 tuplas a recuperar se encuentren en el mismo bloque, los cálculos se hacen con la peor situación posible (o sea que cada tupla esté en un bloque distinto).

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

$$\mathbf{X_I} - 1 + \mathbf{BH'_I} + \mathbf{T'}$$

donde $\mathbf{BH'_I}$ es la cantidad de bloques de nodos hoja del índice a recorrer y $\mathbf{T'}$ es la cantidad de tuplas coincidentes con el criterio de búsqueda.

✍ ***Notar que el -1 de la fórmula anterior se debe a que dentro de los bloques de nodos hoja está incluido el último nivel del árbol.***

Refinando la fórmula, tendremos que el costo de acceso es:

$$\mathbf{X_I} - \mathbf{1} + \lceil \mathbf{T'}/\mathbf{FB_I} \rceil + \mathbf{T'}$$

donde $\mathbf{FB_I}$ es el factor de bloqueo del índice I

✍ ***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*** $\mathbf{X_I} + \mathbf{1}$

✍ ***Además, notar que si todos los punteros a tuplas que coinciden con la búsqueda entran en una sola hoja, el costo queda*** $\mathbf{X_I} + \mathbf{T'}$

4.7.3. 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 $>= c$.

$$\mathbf{X_I - 1 + \lceil T'/FB_I \rceil + T'}$$

donde **T'** es la cantidad de tuplas que cumplen con el criterio de búsqueda y $\mathbf{FB_I}$ el factor de bloqueo del índice I.

Las tuplas resultantes quedan ordenadas según el atributo A.

4.8. Índices basados en hash estático

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 I está hasheado por el atributo A.

4.8.1. Costo de exploración completa

En este caso el índice no nos ayuda.

4.8.2. 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.***

$$\mathbf{MBxB_I + T'}$$

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

4.8.3. Costo de búsqueda por rango ($c <= A <= d$)

Nuevamente, en este caso el índice no nos ayuda. Notar que para poder utilizar un índice hash es necesario proveer los valores de búsqueda, con lo cual no será posible efectuar búsquedas por rango, dado que sería necesario conocer todos los valores posibles comprendidos en el rango y en algunos casos puede haber un número infinito de valores en un rango.

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	$\log_2(B_R) + \lceil T' / FB_R \rceil$	$\log_2(B_R) + \lceil T' / FB_R \rceil$
Índice B+ clustered sobre A	-	$X_I + \lceil T' / FB_R \rceil$	$X_I + \lceil T' / FB_R \rceil$
Índice B+ unclustered sobre A	-	$X - 1 + \lceil T' / FB_I \rceil + T'$	$X - 1 + \lceil T' / FB_I \rceil + T'$
Índice hash estático sobre A	-	$MBxB_I + T'$	-

Donde:

- **T'** es la cantidad de tuplas que cumplen con el criterio de la búsqueda
- $\mathbf{FB_R I}$ es el factor de bloqueo del archivo
- $\mathbf{FB_I}$ es el factor de bloqueo del índice I
- $\mathbf{MBxB_I}$ 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.

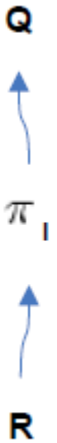
✍ ***En el alcance de la materia no consideraremos cláusulas del estilo ORDER BY , GROUP BY, ni funciones de agregación, ni sentencias SQL anidadas.***

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 $\pi(l, R, Q)$ procesa las tuplas de una relación de entrada R y produce unresultado Q con todas las tuplas pero sólo conteniendo los atributos indicados en la lista l, equivalente a $Q = \pi_l(R)$



5.1.1. Características del resultado

5.1.1.1 Cantidad 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:

$$\mathbf{T_Q = T_R}$$

5.1.1.2 Longitud de tuplas

La longitud de las tuplas del resultado será la suma de las longitudes de los atributos incluidos en la proyección. 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.

$$\mathbf{L_Q = \sum_{Ai \in l}(L_{Ai})}$$

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, cuántas tuplas del resultado entran por bloque:

$$\mathbf{FB_Q = [L_B / L_Q]}$$

(se considera parte entera porque sólo se guardan tuplas enteras en los bloques)
La cantidad de bloques ocupados será

$$\mathbf{B_Q = \lceil T_Q / FB_Q \rceil}$$

(se considera parte entera por exceso ya que por más que no se llene un bloque, el resto se desperdiciará)

5.1.1.4 Costo 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:

$$\mathbf{CO = B_Q}$$

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 del Input: El costo del input es $\mathbf{CI = B_R}$

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 $\mathbf{CI = X–1 + BH_I}$

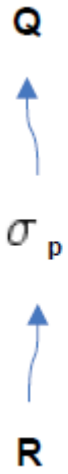
✍*Notar que consideramos* X–1 *porque es lo que cuesta encontrar el puntero al primer nodo hoja (hay que "bajar" en el árbol hasta su padre)*

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: $\mathbf{CI = CBU_I * MBxB_I}$

5.2. Selección (σ)

La operación de selección $\sigma(p, R, Q)$ procesa las tuplas de una relación de entrada R y genera un resultado Q con todas las tuplas que cumplen la condición p. Equivalente a $Q = \sigma_p(R)$



5.2.1. Características del resultado

5.2.1.1 Cantidad de tuplas

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.

5.2.1.1.1 Caso condición del tipo $A = a$

Si A no es clave candidata, se estimará como $\mathbf{T_Q = T_R / I_{R,A}}$

✍ ***Notar que si A es clave candidata,*** $\mathbf{I_{R,A} = T_R}$***, por lo que la cantidad de tuplas resultante es*** 1

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

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

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.***

5.2.1.1.3 Caso condición del tipo $(A > a)$

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

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

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:

$$\mathbf{T_Q = T_R/2}$$

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

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

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

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:

$$\mathbf{T_Q = T_R/2}$$

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

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

5.2.1.1.6 Caso condición del tipo $A = a$ *or* $B = b$

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

✍ ***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

5.2.1.1.7 Caso general

En el caso general, estimaremos la cantidad de tuplas a partir de la *selectividad*, con la siguiente fórmula:

$$\mathbf{T_Q = sel(p) * T_R}$$

5.2.1.2 Longitud de tuplas

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

$$\mathbf{L_Q = L_R}$$

5.2.1.3 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.
El factor de bloqueo del resultado es el mismo que el de la relación input, ya que las tuplas son iguales:

$$\mathbf{FB_Q = FB_R}$$

La cantidad de bloques ocupados será:

$$\mathbf{B_Q = \lceil T_Q / FB_Q \rceil}$$

5.2.1.4 Costo 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:

$$\mathbf{CO = B_Q}$$

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): $\mathbf{CI = B_R}$

✍ ***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 p_1 *and* p_2, donde p_1 y p_2 son dos predicados válidos, y p_1 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 p_2. 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: $\lceil (sel(p_k) * T_R)/FB_R. \rceil$ (parte entera por exceso)

Por lo tanto nos queda:

$$\mathbf{CI = \log_2(B_R) + \lceil \lceil (sel(p_k) * T_R) \rceil / FB_R. \rceil}$$

✍ ***Notar que en caso de una búsqueda por igualdad de una clave candidata, se simplifica la fórmula anterior,quedando:*** $\mathbf{CI = \log_2(B_R)}$

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 **I** árbol $\boldsymbol{B^+}$ clustered según una clave k y el predicado p coincide con k.

Descripción: Esto se resuelve accediendo según el índice, como se vio para búsquedas por igualdad o rango de índices árbol $\boldsymbol{B^+}$ clustered.

Costo de input:

$$\mathbf{CI = X_I + \lceil \lceil (sel(p_k) * T_R) \rceil / FB_R. \rceil}$$

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** árbol $\boldsymbol{B^+}$ unclustered según una clave k y el predicado p coincide con k.

Descripción: Esto se resuelve accediendo según el índice, como se vio para búsquedas por igualdad o rango de índices árbol B^+ unclustered.

Costo de input:

$$\mathbf{CI = X_I - 1 + \lceil\lceil(sel(p_k) * T_R)\rceil / FB_R.\rceil + \lceil\lceil(sel(p_k) * T_R)\rceil}$$

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 y el predicado p coincide con k.

Descripción: Esto se resuelve accediendo según el índice, como se vio para búsquedas por igualdad en índices basados en hash.

Costo de input:

$$\mathbf{CI = MBxB_I + \lceil(sel(p_k) * T_R)\rceil}$$

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 p_2 coincide con un índice I_2

Descripción:

Se realiza sobre I_1la búsqueda de las tuplas que cumplen p_1. Los **rids** de las tuplas encontradas se escriben en un archivo hash intermedio. Algo análogo se hace sobre I_2 para las tuplas que cumplen p_2, 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 $p_1 \wedge ... \wedge p_i \wedge ... \wedge p_n$***

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 conjuntivo $p = p_1 \vee p_2$, tal que p no coincide con ningún índice, pero p_1 coincide con un índice I_1 y p_2 coincide con un índice I_2

Descripción:

Se realiza sobre I_1la búsqueda de las tuplas que cumplen p_1. Los **rids** de las tuplas encontradas se escriben en un archivo hash intermedio. Algo análogo se hace sobre I_2 para las tuplas que cumplen p_2, 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 conjuntivo de la forma*** $p_1 \vee ... \vee p_i \vee ... \vee p_n$

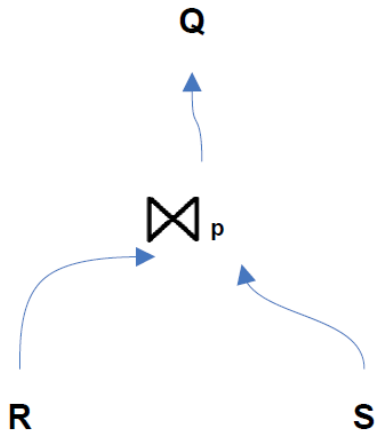
Costo de input (CI): Se deja como ejercicio, suponiendo que el *hash intermedio* entra completo en memoria.

✍ ***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. Junta (Join) $\bowtie$.

La operación de Junta $\bowtie$ (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 $R \times S$ que cumplen con la condición p. Equivalente a $Q = R \bowtie_p S$.



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

5.3.1.1 Cantidad de tuplas

Dada una condición de junta del estilo R.ri = S.sj, 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:

$\mathbf{T_Q = (T_R * T_S)/max(I_{R.r_i}, I_{S.s_j})}$

✍ ***Notar que cuando r_i es clave foránea de s_j (i.e. s_j es clave primaria de S), tendremos que $I_{R.r_i} \leq I_{S.s_j} = T_S$. Por lo tanto, en este caso $T_Q = T_R$; 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"***

5.3.1.2 Longitud 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: $\mathbf{L_Q = L_R + L_S}$

✍ ***Tener en cuenta que para la junta natural (natural joins), la longitud de las tuplas es $L_R + L_S - L'$, donde L' es la longitud de los campos comunes entre R y S (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:

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

La cantidad de bloques ocupados será:

$\mathbf{B_Q = \lceil T_Q/FB_Q \rceil}$

5.3.1.4 Costo 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:

$\mathbf{CO = B_Q}$

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:

Algoritmo 1: BNLJ

para *para cada segmento de* $B-2$ *bloques de* R **hacer**
  para *cada bloque de* S **hacer**
    para *toda tupla* r *del segmento de* R *y tupla* s *del bloque de* S **hacer**
      si $r_i == s_j$ **entonces**
        Agregar $< r_i, s_i >$al resultado

Costo de input (CI)

$\mathbf{CI = B_R + B_S * \lceil B_R/(B-2) \rceil}$

Explicación: Coloco en memoria $B-2$ bloques de B_R y recorro B_S entero. Esto lo hago$\lceil B_R/(B-2) \rceil$ veces. A su vez, estoy recorriendo todo R una sola vez, entonces debo sumar B_R.

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.

Descripción:

Algoritmo 2: INLJ

para *para cada tupla* r_i *de* R **hacer**
  para *cada tupla* s_i *de* S */* $r_i = s_j$ *(obtenida según el índice de* S*)* **hacer**
    Agregar $< r_i, s_i >$al resultado

Costos de input (CI):
CI $= \mathbf{B_R} + \mathbf{T_R} *$ (buscar para la tupla $\mathbf{t_r}$ index entry/ies en índice de **S** + buscar valor/es apuntados por 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: El algorigmo 3 nos muestra como funciona el Sort Merge Join
Obs: Llamamos partición al conjunto de tuplas que poseen el mismo valor de cierto atributo

Algoritmo 3: SMJ

```
si R no está ordenada en el atributo i entonces
    ordenar R
si R no está ordenada en el atributo j entonces
    ordenar S
r ← primera tupla de R                                   // itera sobre R
s ←primera tupla de S                                    // itera sobre S
mientras r ≠ null ∧ s ≠ null hacer          // al final se obtiene null
    si r ≤ s entonces
        r ← siguiente tupla en R                // No hay partición en S para r
    si no, si r = s entonces                // Se encontro partición en S para r
        mientras r = s hacer                 // Se procesa partición actual de R
            inicio_part_s ← s    // Marca inicio de la partición en S para r
            mientras r = s hacer          // Se procesa partición actual de S
                Agregar < r, s >al resultado
                s ← siguiente tupla en S
            s ← inicio_part_s
            r ← siguiente tupla en R
    en otro caso                                                    // r > s
        s ← siguiente tupla en S // s no apuntaba al inicio de la partición
            en S de ri entonces sigo buscando
```

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: $(\lceil \log_{B-1} \lceil B_R/B \rceil \rceil + 1) * 2B_R$

Donde $(\lceil log_{B-1} \lceil B_R/B \rceil \rceil + 1)$ 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

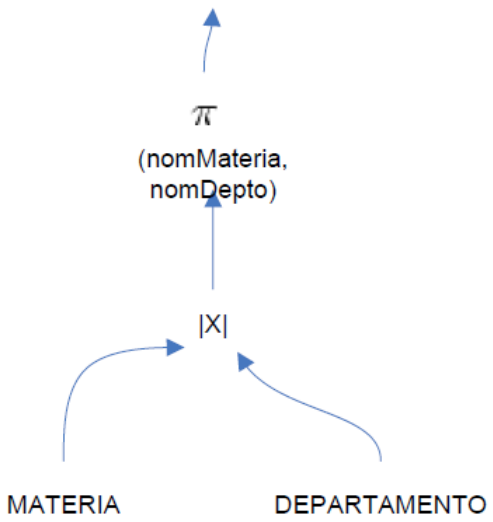
Hasta ahora vimos cómo evaluar el procesamiento de operaciones algebraicas individualmente, estimando sus costos de input y de output.

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_depto
```

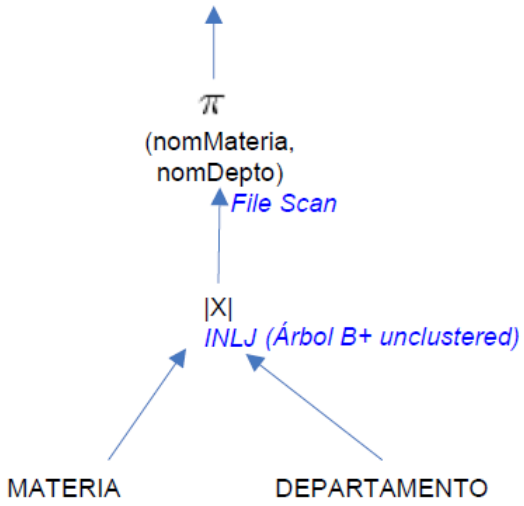
Podría representarse por el siguiente árbol (no es la única representación posible):



- 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 $\bowtie$ y π
- Los arcos de un nodo hacia sus sucesores representan las relaciones de input de esa operación, en este caso $\bowtie$ tiene dos entradas, MATERIA y DEPARTAMENTO, en tanto que π tiene un una entrada, MATERIA $\bowtie$ DEPARTAMENTO.
- Los arcos de un nodo hacia sus antecesores representan la relación *output* de esa operación, en este caso, $\bowtie$ tiene un *output* que es MATERIA $\bowtie$ DEPARTAMENTO.
- El arco saliente del nodo raíz representa el resultado final. En este caso, el arco saliente de π representa la relación resultante de realizar:
  π(nomMateria, nomDepto) (MATERIA $\bowtie$ 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 índice árbol B^+ uncluster-ed sobre DEPARTAMENTO.cod_depto, un posible plan de ejecución sería el siguiente:



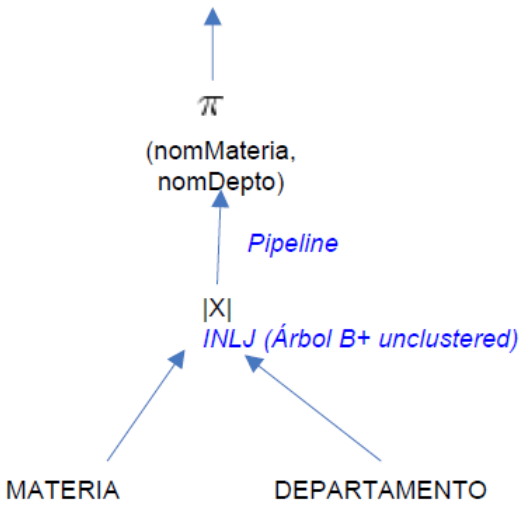
Para obtener los costos totales de ejecución de un plan, tenemos que sumar todos los costos de input y output de los diferentes nodos del mismo (con excepción, como ya se dijo, del output del nodo raíz, ya que es siempre el mismo para cualquier plan).

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
$\sigma_{C_1 \wedge C_2 ..... \wedge C_n}(R) \equiv \sigma_{C_1}(\sigma_{C_2}(...\sigma_{C_n}(R)))$

Conmutatividad de σ
$\sigma_{C_1}(\sigma_{C_2}(R)) \equiv \sigma_{C_2}(\sigma_{C_1}(R))$

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

Conmutatividad de σ con respecto a π
$\pi_{A1,A2...An}(\sigma_C(R)) \equiv \sigma_C(\pi_{A1,A2...An}(R)$
Si C referencia solamente a atributos dentro de A1...An

Conmutatividad del Producto Cartesiano (o Junta)
$R \times S \equiv S \times R$

Conmutatividad de σ con respecto al Producto Cartesiano (o Junta)
$\sigma_C(R \times S) \equiv (\sigma_{C_R}(R)) \times \sigma_{C_S}(S))$ donde $C = C_r \cup C_S$

Conmutatividad de π con respecto al Producto Cartesiano (o Junta)
$\pi_L(R \times S) \equiv (\pi_{L_1}(R) \times \pi_{L_2}(S)$ donde $L = L_1 \cup L_2$

Conmutatividad de operaciones de conjuntos $\cup$ e $\cap$
$R \Theta S \equiv S \Theta R$
si $\Theta \in \{\cup, \cap\}$

Asociatividad del Producto Cartesiano, Junta, $\cup$ e $\cap$
$(R\ \Theta\ S)\ \Theta\ T \equiv R\ \Theta\ (S\ \Theta\ T)$
si $\Theta \in \{\cup, \cap\ \bowtie, \times\}$

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.

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.***

Referencias

[1] *Fundamentals of Database Systems, 7th Ed* Elmasri Ramez; Navathe Shamkant - Pearson, 2016

[2] *Database Management Systems.* R. Ramakrishnan; J. Gehrke - McGraw-Hill, 2003

[3] *Fundamentos de Bases de Datos. 6ta Edición* A. Silberschatz; H. Korth; S. Sudarshan - McGraw-Hill, 2014

[4] *Principles of Database and Knowledge Base Systems,* J. Ullman - Science Press, 1988

[5] *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