Capítulo 4

Procesamiento de consultas

Visión general

- Una expresión de álgebra de tablas puede tener varias expresiones equivalentes.
- **Ejemplo**: $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ es equivalente a $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- Cada operación del álgebra de tablas puede ser evaluada usando uno de varios algoritmos diferentes.
- **Ejemplo**: para implementar la selección previa podemos buscar cada tupla en *instructor* para encontrar tuplas con *salario* menor a 75000. Si un índice árbol B+ está disponible para el atributo salario, podemos usar ese índice para localizar las tuplas.
- Para **especificar cómo evaluar una consulta** necesitamos anotarla con instrucciones especificando cómo evaluar cada operación.
  - Esas **anotaciones** pueden indicar el algoritmo a ser usado para la operación específica o el índice particular a usar.

Visión general

- Una operación del álgebra de tablas anotada con instrucciones de cómo evaluarla se llama una **primitiva de evaluación**.
- Una secuencia de primitivas de evaluación que pueden usarse para evaluar una consulta se llama un **plan de evaluación**.

- La **máquina de ejecución de consultas** toma el plan de evaluación de consulta, ejecuta ese plan y retorna las respuestas de la consulta.

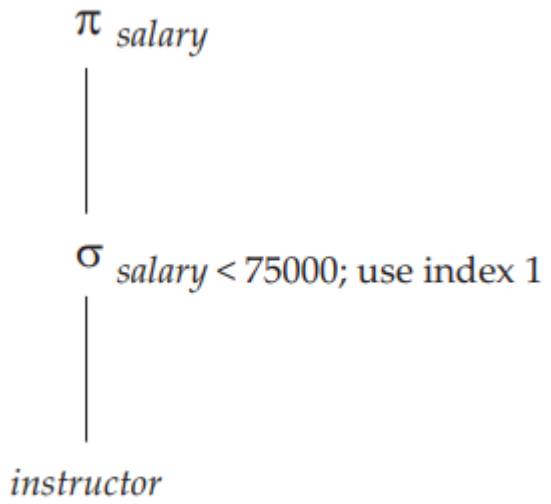


Figure 12.2 A query-evaluation plan.

Visión general

- Los diferentes planes de evaluación para una consulta dada pueden tener diferentes costos.
- Es responsabilidad del SGBD construir un plan de evaluación que minimiza el costo de evaluación de consultas.
- Una vez que un plan de evaluación es elegido, la consulta es evaluada con este plan.
- **Optimización de consultas**: entre todos aquellos planes de evaluación equivalentes encontrar aquel con menor costo.
  - El costo es estimado usando información estadística de la base de datos.
  - **P.ej:** número de registros en cada tabla, tamaño de los registros, etc.
- Con el fin de optimizar una consulta un optimizador de consultas debe conocer el costo de cada operación.

Visión general

- **En este capítulo vamos a estudiar:**
  - Cómo medir costos de consultas
  - Algoritmos para evaluar operadores del álgebra de tablas.
  - Técnicas de procesamiento de consultas
- En el próximo capítulo estudiamos cómo optimizar consultas, o sea, cómo encontrar un plan de evaluación con el menor costo estimado.

Medidas de costo de consulta

- En BD grandes, los **accesos a disco** que se miden como número de transferencias de bloques de disco son el costo más importante.
  - Ya que los accesos a disco son más lentos comparados con las operaciones en memoria.
- Se usará el número de transferencia de bloques de disco como una medida de costo real.
  - Por simplicidad de las cuentas vamos a asumir que todas las transferencias de bloques tienen el mismo costo.
- Es necesario distinguir entre las lecturas y las escrituras de bloques, ya que se tarda más tiempo en escribir un bloque que leerlo de disco.
  - Los datos son leídos de nuevo para asegurarse que la escritura fue exitosa.
- Una medida más precisa sería estimar:
  - El número de operaciones de búsqueda realizadas, el número de bloques leídos, el número de bloques escritos.

Medidas de costo de consulta

- Usamos el **número de transferencia de bloques** del disco y el **número de búsquedas en disco** (tiempo que le lleva a la cabeza lectora posicionare en un bloque deseado) para estimar el costo de un plan de evaluación de consultas.
  - t_T – tiempo promedio para transferir un bloque de datos
  - t_S – tiempo promedio de acceso a bloque
  - El costo para *b* tranferencias de bloques y *S* accesos a bloques es:

    $$b * t_T + S * t_S$$
- Valores típicos hoy en día serían t_S = 4 milisegundos y t_T = 0.1 milisegundos, asumiendo un tamaño de bloque de 4-KiB y una tasa de transferencia de 40 MB por Segundo.
- Podemos refinar las cuentas distinguiendo lecturas de disco de escrituras de disco.

Medidas de costo de consulta

- Los costos de todos los algoritmos considerados dependen del **tamaño del búfer** en memoria principal.
  - En el mejor caso todos los datos pueden ser leídos en búfer y el disco no necesita ser accedido de nuevo.
  - En el peor caso asumimos que el búfer puede sostener solo unos pocos bloques de datos – aproximadamente un bloque por tabla.
- Cuando presentamos estimaciones de costos, asumimos generalmente el peor caso.
- Vamos a ignorar los costos de CPU por simplicidad.

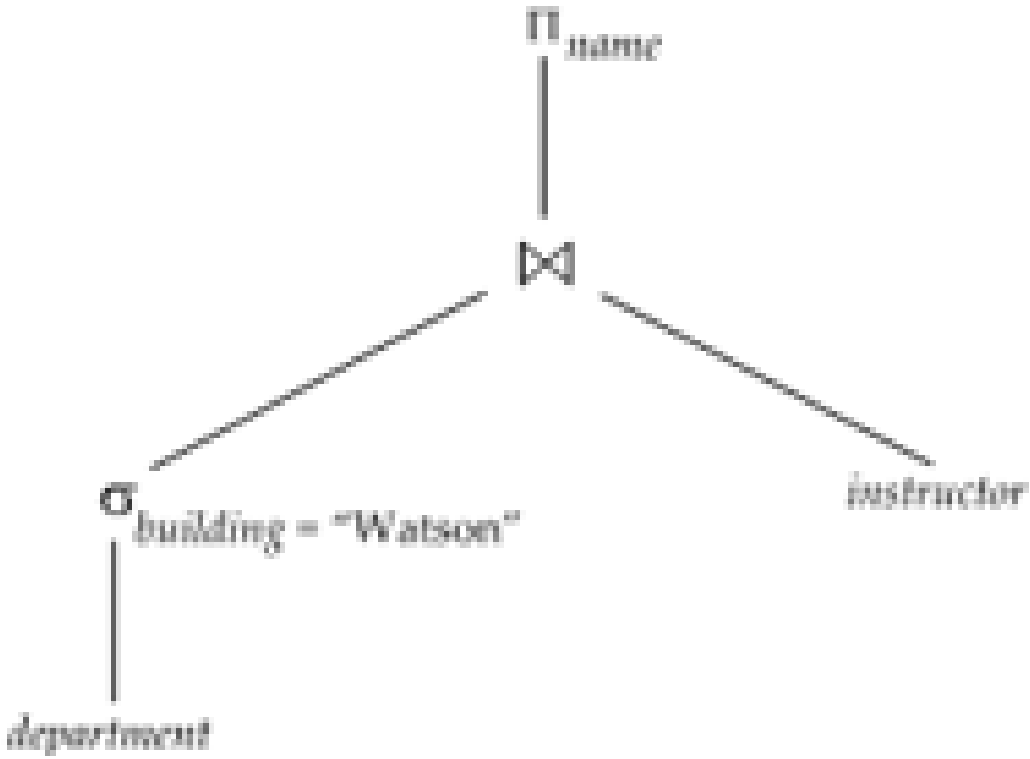
Árbol binario de ejecución

- El **árbol binario de ejecución** de una consulta es el árbol binario de una expresión de consulta.
  - Los nodos hoja son tablas de la base de datos
  - Los nodos internos son operadores del álgebra de tablas
- Un **operador lógico** del álgebra de tablas considera el algoritmo ineficiente (usado para calcularlo) definido en el capítulo del álgebra de tablas.
  - El árbol binario de ejecución está formado por operadores lógicos.
- El árbol binario de ejecución se puede calcular a partir de:
  - Expresión del álgebra de tablas
  - Expresión de consulta SQL.

Árbol binario de ejecución

- El árbol binario de ejecución asociado a la consulta

$$\Pi_{name}(((\sigma_{building='watson'}(department)) \bowtie instructor)$$



Evaluación del árbol binario de ejecución

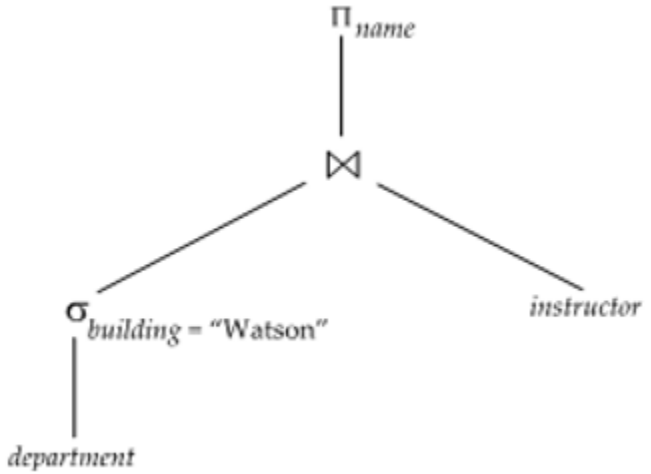
- **Operadores físicos** son algoritmos específicos para operadores del álgebra de tablas.
  - En general tienden a ser más eficientes que los algoritmos lógicos porque hacen uso de informaciones adicionales, como índices, tamaños de búfer en memoria, técnicas avanzadas de algorítmica, etc.
  - Un operador del álgebra de tablas va a tener unos cuantos operadores físicos.
- La evaluación de un árbol binario de ejecución va a estar en términos de operadores físicos.

Evaluación del árbol binario de ejecución

- El resultado de evaluar un operador de un nodo interno del árbol binario de ejecución que no es la raíz del árbol se llama **resultado intermedio**.
- Para la evaluación del árbol binario de ejecución hay dos enfoques:
  - **Materialización**: los resultados intermedios se guardan en disco en tablas temporales a las cuales tiene acceso el sistema gestor de BD (SGBD).
    - No hay índices sobre este tipo de tablas.
  - **Encausamiento**: a medida que se van generando los resultados intermedios se van pasando al siguiente operador.
    - Los resultados intermedios no se guardan en disco.
- **Comparando ambos enfoques**: Materialización usa mucho menos memoria y bastante más espacio en disco y encausamiento usa mucho menos espacio en disco, y bastante más memoria.

Materialización

- **Ejemplo**: Supongamos que tenemos el árbol binario de ejecución y usamos materialización:



1. Computamos y almacenamos en disco primero $\sigma_{\text{building = 'Watson'}}$ (department)
2. Computamos y almacenamos en disco la reunión natural del resultado intermedio anterior con *instructor*.
3. Computamos la proyección según *nombre*.

Materialización

- Con materialización:
  - Cambiar nodos lógicos por físicos en el árbol binario de ejecución.
  - Si hay más de un operador físico posible, elegir el menos costoso.
  - Evaluamos un operador físico por vez comenzando en el nivel más bajo.
  - Usamos resultados intermedios en tablas temporales para evaluar los operadores físicos del siguiente nivel.

Materialización

- **Problema**: Hay que estimar el tamaño de los resultados intermedios en cantidad de bloques a escribir a disco.
  - Esto es necesario cuando el operador es de selección o de reunión (selectiva o natural).
- **Solución**:
  1. Usar una función de probabilidad llamada **factor de selectividad** para calcular la cantidad de registros del resultado intermedio.
  2. A partir de cantidad de registros, calcular la cantidad de bloques del resultado intermedio.

Materialización

- Si el operador de selección o reunión usa predicado P, y el input del operador es i, denotamos al factor de selectividad mediante: fs(P, i).
- **Para selección:**

  cantidad de registros resultado intermedio = |r| * fs(P, r)
- **Para reunión:**
  - Cantidad de registros resultado intermedio = |r| * |s| * fs(P, r, s)
- Una forma de calcular el factor de selectividad es asumir uniformidad e independencia:
  - **Uniformidad**: todos los valores de un atributo son igualmente probables
  - **Independencia**: condiciones sobre diferentes atributos son independientes

Materialización

- **Ejemplo**: atributo sexo de persona: todos los valores de sexo son igualmente probables (la proporción de hombres es igual a la proporción de mujeres).
  - fs(sexo = ‘F’, persona) = 1/2
- **Ejemplo**: Atributos sexo y edad en persona, P = edad = 40 y sexo = ‘F’. La edad es independiente del sexo.
  - fs(edad = 40 and sexo = ‘F’, persona) = fs(edad = 40, persona) * fs(sexo = ‘F’, persona)
- Pero no es obligatorio usar uniformidad o independencia para calcular el factor de selectividad.

Materialización

- ¿Cómo calcular el número de bloques si tengo en el resultado N registros de tamaño R cada uno y B es el tamaño del bloque?

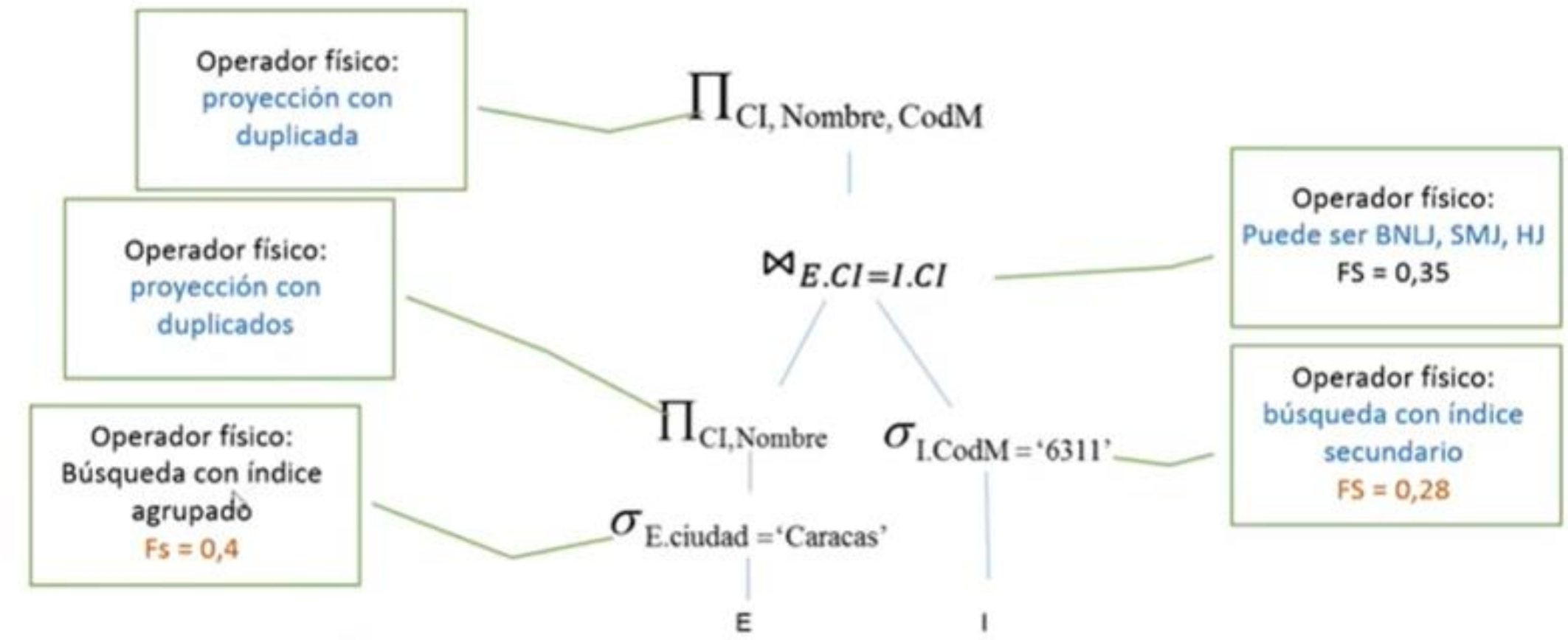
  NumBloques = ⌈(N x R) / B⌉ .

Materialización

- Ahora vemos cómo procesar y estimar el costo de una consulta con materialización.
- **Fase 1: decidir el plan de ejecución**
  - Armar el árbol binario de ejecución
  - Calcular el factor de selectividad para selecciones y reuniones (selectivas y naturales).
  - Decidir operadores físicos.
    - Solo se usan índices si la tabla de la BD lo amerita.

Materialización

- **Ejemplo de fase 1:**

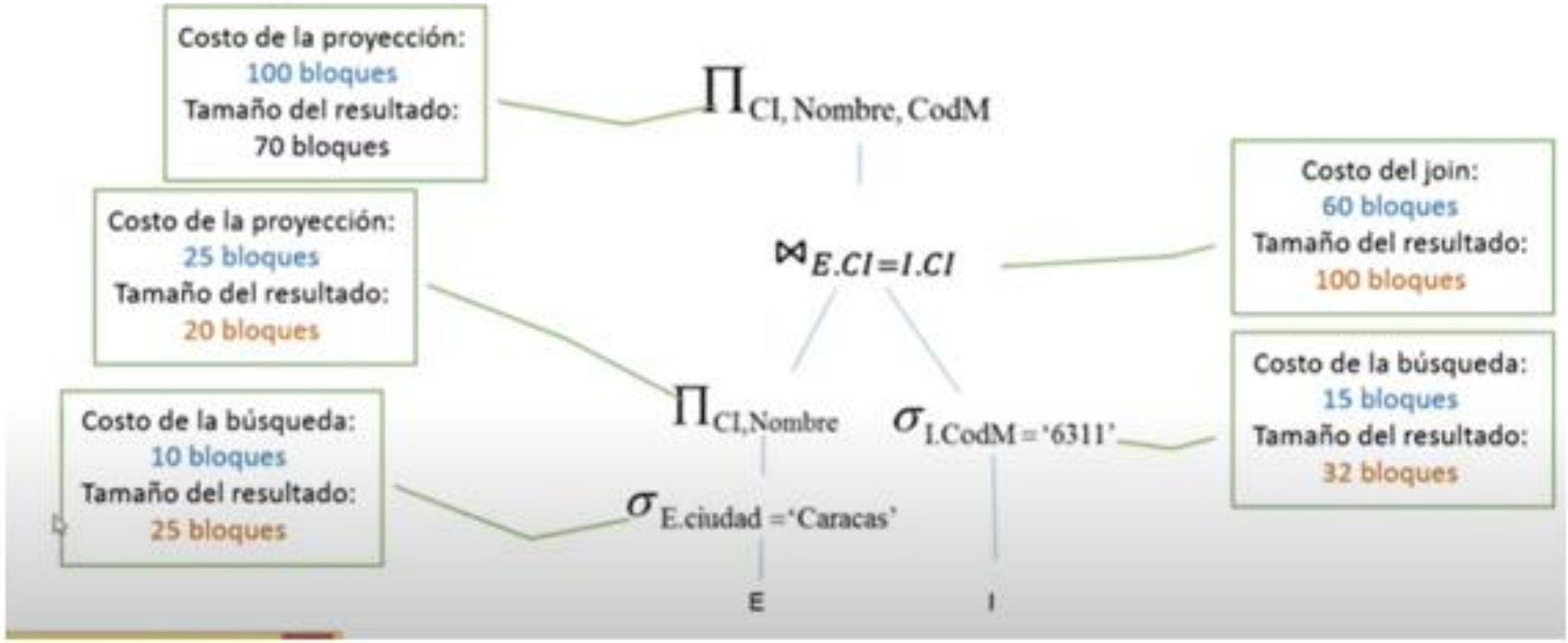


Materialización

- **Fase 2: estimar el costo de ejecutar el plan de evaluación**
  1. Calcular el tamaño en bloques de las tablas de la BD
  2. Calcular el tamaño de los resultados intermedios en bloques
  3. Calcular el costo de los operadores físicos
  4. Sumar los costos totales

Materialización

- **Ejemplo de Fase 2:**



Materialización

- Juntando todo:

  - Materialización
    - Es la más usada por disponer de mayor cantidad de espacio en disco
    - $Costo\ total = \sum costo(operaciones) + \sum costo(materialización)$
    - Costo total = (10+20+15+60+100) + (25+20+32+100)
    - Costo total = 382 (accesos a disco)
    - Esto se multiplica por la velocidad de transferencia y se tiene el tiempo

Materialización

- Para poder aplicar materialización en detalle nos falta:
  - Dar reglas para el cálculo del factor de selectividad.
  - Dar operadores físicos con sus costos.
- A continuación vamos a ver estas cosas.
  - Y lo haremos por operador lógico.

Operación de selección

- Definimos fs(P, r) para distintos tipos de propiedades P y tabla r.
- Desde ahora *r* tabla, *A*, *A'* atributos de *r*, *c* y *c'* constantes.
- **Regla 1:** Asumiendo uniformidad:
  - fs(A = c, r) = 1/V(A, r), donde V(A, r) número de distintos valores que aparecen en *r* para *A*.
- **Regla 2**: Asumiendo uniformidad, A con valor numérico:
  - fs(A $\geq$ c, r) = (max(A, r) - c) / (max(A, r) – min(A, r))
- **Regla 3**: Asumiendo uniformidad, A con valor numérico:
  - fs(A < c, r) = (c – min(A)) / (max(A) – min(A) + 1)

Operación de selección

- **Regla 4:** asumiendo uniformidad, A con valor numérico:
  - fs ($c \leq A < c'$, r) = $(c' - c)/ (\max(A, r) - \min(A, r))$
- **Regla 5**: asumiendo independencia:
  - fs($P1 \wedge P2 \wedge \ldots \wedge Pn$, r) = fs($P_1$, r) fs($P_2$, r) …. fs($P_n$, r)
- **Regla 6**: usando propiedad de probabilidades:
  - fs($\neg P$, r) = 1 – fs(P, r)
- **Regla 7**: asumiendo independencia
  - fs($P \vee Q$, r) = fs($\neg (\neg P \wedge \neg Q$, r)) = 1 – fs($\neg P \wedge \neg Q$, r) = 1 – (fs($\neg P$, r) * fs($\neg Q$, r))
    = 1 – ((1 – fs(P, r)) * (1 – fs(Q, r)))

Operación de selección

- **Algoritmo de búsqueda lineal**
  - Escanear cada bloque del archivo y testear todos los registros para ver si satisfacen la condición de selección.
  - Estimación de costo = b_r transferencias de bloques + 1 acceso a bloque
    - b_r denota el número de bloques conteniendo registros de la tabla *r*
    - Otra fórmula para el costo: $t_S + b_r * t_T$
  - Si se hace búsqueda lineal y se selecciona según igualdad para clave:
    - costo = (b_r /2) transferencias de bloques + 1 acceso a bloque
    - Este costo es para el caso promedio.

Operación de selección

- **Algoritmo para índice primario usando árbol B+ con igualdad en clave candidata**
  - Recorrer la altura del árbol más una E/S para recoger el registro.
  - Cada una de estas operaciones requiere un acceso a bloque y una transferencia de bloque.
  - $Costo = (h_i + 1) * (t_T + t_S)$,
  - donde h_i denota la altura del índice.

Operación de selección

- **Algoritmo para índice primario usando árbol B+ igualdad para no clave candidata**
  - Hay un acceso a bloque para cada nivel del árbol y un acceso a bloque para el primer bloque.
  - Los registros van a estar en bloques consecutivos.
  - Sea b el número de bloques conteniendo registros con la clave de búsqueda especificada, todos los cuales son leídos.
  - Estos bloques son bloques hoja asumiendo que están almacenados secuencialmente y no requieren accesos adicionales a bloque.
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

Operación de selección

- **Algoritmo para índice secundario usando árbol B+, igualdad en clave candidata**
  - Este caso es similar al índice primario.
  - *Costo* = $(h_i + 1) * (t_T + t_S)$,
  - donde h_i denota la altura del índice.
- **Algoritmo para índice secundario usando árbol B+, igualdad en no clave**
  - Aquí el costo de recorrido del índice es el mismo. Sin embargo, cada registro puede estar en un bloque diferente, requiriendo un acceso a bloque por registro.
  - Sea *n* el número de registros recogidos.
  - Costo = $(h_i + n) * (t_T + t_S)$
  - Esto puede ser muy costoso.

Selecciones involucrando comparaciones

- Podemos implementar selecciones de la forma $\sigma_{A \leq V}(r)$ o $\sigma_{A \geq V}(r)$ usando:
  - un escaneo de archive lineal,
  - o usando índices
- **Algoritmo para índice primario:**
  - La tabla está ordenada en *A*.
  - para $\sigma_{A \geq V}(r)$ usar el índice para encontrar el primer registro $\geq v$ y escanar la tabla secuencialmente desde allí.
    - Costo = hi $*$ (tT + tS) + b $*$ tT
    - b el número de bloques conteniendo registros con A $\geq v$.
  - para $\sigma_{A \leq V}(r)$ escanear la tabla secuencialmente hasta la primera tupla > *v;* no usar el índice.

Selecciones involucrando comparaciones

- **Algoritmo para índice secundario**
  - para $\sigma_{A \geq V}(r)$ usar el índice para encontrar la primera entrada del índice $\geq v$ y escanear el índice secuencialmente desde allí, para encontrar los punteros a los registros.
  - Costo = $(hi + n) * (t_T + t_S)$,
    - donde *n* número de registros con A >= v.
  - para $\sigma_{A \leq V}(r)$ escanear hojas del índice encontrando punteros a los registros, hasta la primera entrada > *v.*
  - En ambos casos retornar los registros que son apuntados
    - Requiere una E/S para cada registro
    - búsqueda lineal de la table puede ser más barata.

Selecciones involucrando comparaciones

- Leer implementación de selecciones complejas:
  - Sección 12.3.3 del Silberschatz.

Operación de proyección

- **Algoritmo para proyección**
  - Requiere recorrer todos los registros y realizar una proyección en cada uno.
  - Se recorren todos los bloques de la tabla.
  - Estimación de costo = b_r transferencias de bloques + 1 acceso a bloque
    - b_r denota el número de bloques conteniendo registros de la tabla *r*
  - Proyección generalizada puede ser implementada de la misma manera que proyección.

Operación de ordenamiento

- Estudiamos el caso de ordenamiento donde las tablas son mayores que la memoria principal.
  - Ordenar tablas que no caben en memoria se llama **ordenamiento externo**.
  - El algoritmo de ordenamiento externo más usado se llama **ordenamiento-combinación externo (external merge sort)**.

Operación de ordenamiento

- Sea *M* la cantidad de bloques en búfer de memoria principal disponibles para ordenación.
  1. **Crear corridas ordenadas**

     i = 0

     repeat

     Leer *M* bloques de la tabla en memoria

     ordenar esos bloques en memoria

     Escribir los datos ordenados al archivo de ejecución R_i,

     i = i+1

     until el fin de la tabla

     - Sea N el valor final de i
  2. **Combinar las corridas**

Operación de ordenamiento

- Luego las corridas son combinadas. Suponemos que *N* <*M*.

  Leer un bloque de cada uno de los *N* archivos Ri en un bloque del bufer de memoria.

  repeat

    Seleccionar el primer registro en el orden de ordenamiento entre todos los bloques del bufer.

    Escribir el registro al bufer de salida y borrarlo del bloque del búfer

    if bufer de salida esta lleno **then** escribirlo a disco.

    Borrar el registro de su página de bufer de input.

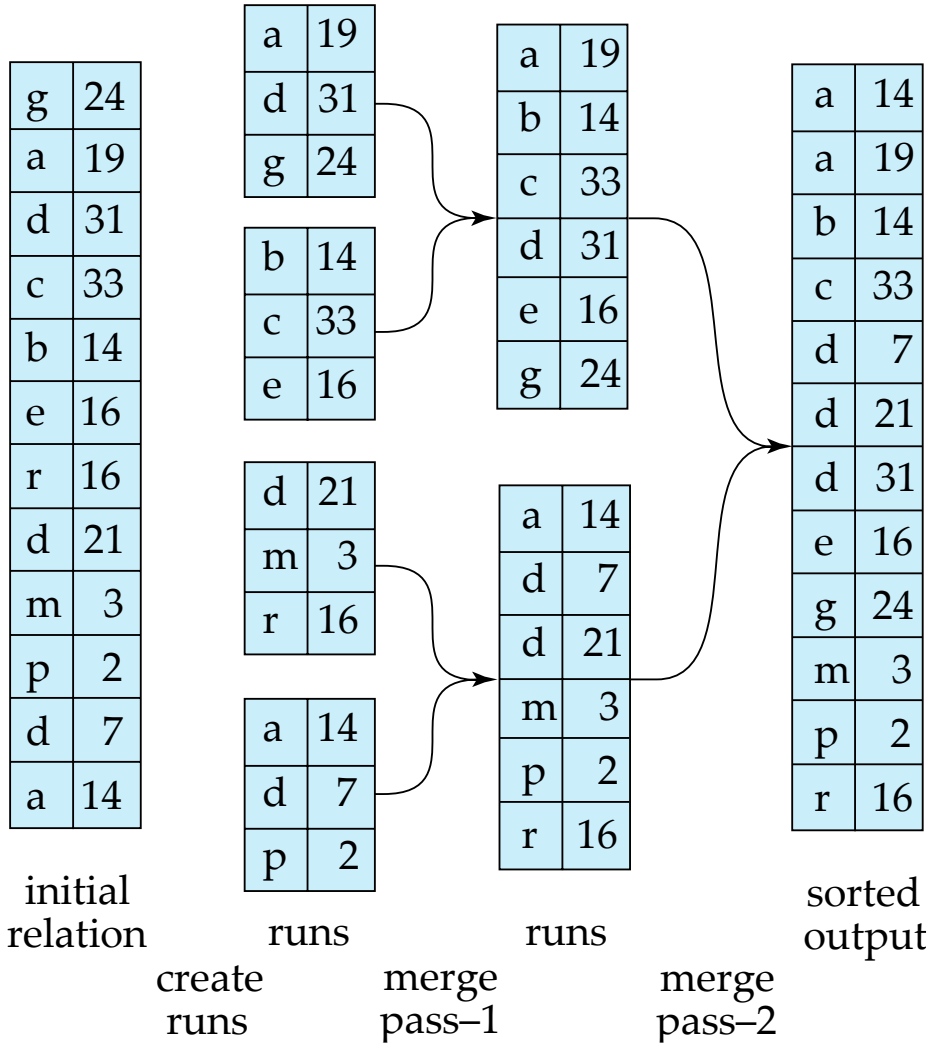
    if pagina del bufer de corrida Ri pasa a estar vacía **then** leer el próximo bloque de la corrida Ri en el bloque de bufer.

  until que todos los bloques del input búfer estén vacías.

Operación de ordenamiento

- Si $N \geq M$, varias pasadas de combinación son requeridas.
  - En cada pasada, grupos contiguous de M-1 corridas son combinados.
  - Una pasada reduce el número de corridas por un factor de M-1 y crrea corridas más largas por el mismo factor.
  - Pasadas repetidas son realizadas hasta que todas las corridas han sido combinadas en una.

Operación de ordenamiento



Operación de ordenamiento

- **Costo del ordenamiento:**
  - Leer justificación en el Silberschatz.
  - Número de transferencias de bloques para ordenamiento externo:
    $b_r (2 \lceil \log_{M-1} (b_r / M) \rceil + 1)$
  - Donde b_r es cantidad de bloques de la tabla
  - Cantidad de accesos a bloque:
    $2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor - 1} (b_r / M) \rceil - 1)$,
  - donde b_b bloques son alojados en cada corrida

Operación de eliminación de duplicados

- **Eliminación de duplicados**
  - Se ordena la tabla; al hacerlo, todos los duplicados van a ser adyacentes entre sí. Y todos los duplicados menos uno pueden ser eliminados.
  - **Optimización**: en ordenación externa duplicados pueden ser eliminados durante generación de corridas así como en pasos de combinación intermedios.
  - El peor caso de costo para eliminación de duplicados es el mismo que el peor caso de costo para ordenar una table.

Ejercicio

- Sea la consulta: $\Pi_{id}\, \sigma_{(edad > 40)}(persona)$
- Donde:
  - Persona ocupa 20 bloques de 10 registros cada uno.
  - La probabilidad de que un registro tenga edad mayor a 40 es 0.5.
  - Asumir que entran 50 id por bloque.
  - Hay 100 edades distintas
  - Entran 20 edades por nodo en el índice.
- Calcular cuántos bloques se leen y cuántos bloques se escriben para cada uno de los siguientes casos:
  1. No hay índice en edad
  2. Hay índice en edad usando árbol B+.

Operación de reunión selectiva

- Si r tabla y A atributo de r, entonces V(A, r) número de distintos valores que aparecen en *r* para *A*.
- **Regla**: asumiendo uniformidad:
  - fs(r.A = s.B, r, s) = 1 /max(V(A, r), V(B, s))
  - la uniformidad significa: para cada valor de atributo A/B en una tabla hay la misma cantidad de tuplas por el atributo B/A que cazan en la otra tabla.
- **Observaciones**:
  - Si A clave candidata de r: fs(r.A = s.B, r, s) = 1 /max(|r|, V(B, s))
  - Si B clave foránea en s referenciando r: fs(r.A = s.B, r, s) = 1 /|r|
  - Si toda tupla en r produce tuplas en la reunión selectiva:
    - fs(r.A = s.B, r, s) = 1 /V(B,s)
  - Si se puede asumir independencia:
    - fs(r.A = s.B Λ r.A' = s.B', r, s) = fs(r.A = s.B , r,s) * fs(r.A' = s.B', r,s)

Operación de reunión selectiva

- Vamos a ver algoritmos para computar la reunión selectiva (y por ende la reunión natural)
- **Algoritmo de reunión selectiva de loop anidado**
- **for each** tuple t_r **in** r **do begin**
  for each tuple t_s **in** s **do begin**
  test pair (t_r, t_s) to see if they satisfy the join condition
  if they do, add $t_r \bullet t_s$ to the result.
  end
  end

Operación de reunión selectiva

- Nomenclatura: r se llama **tabla externa** y s se llama **tabla interna** de la reunión selectiva.
- El algoritmo anterior no requiere de índices.
- El algoritmo anterior es costoso porque examina todo par de tuplas en las dos tablas.
- En el peor caso, si hay suficiente memoria solo para sostener un bloque para cada tabla, el costo estimado es:
  - $n_r * b_s + b_r$ transferencias de bloques
  - $n_r + b_r$ accesos a bloques.
  - donde n_r es cantidad de registros en *r* , b_r es la cantidad de bloques en *r,* y b_s es la cantidad de bloques en *s.*

Operación de reunión selectiva

- **Algoritmo de reunión selectiva de loop anidado de bloques**

for each block B_r **of** r **do begin**
for each block B_s **of** s **do begin**
for each tuple t_r **in** B_r **do begin**
for each tuple t_s **in** B_s **do begin**
if (t_r, t_s) satisface condición de reunión selectiva
then add $t_r \bullet t_s$ to the result.
end
end
end
end

Operación de reunión selectiva

- **Estimación de peor caso:**
  - $b_r * b_s + b_r$ transferencias de bloque + 2 * b_r accesos a bloque
  - Cada bloque en la tabla interna *s* es leído una sola vez por cada bloque en la tabla externa.
- **Mejora del algoritmo anterior:**
  - M tamaño de memoria en bloques.
  - Usar M-2 bloques de disco para tabla externa, usar los restantes 2 bloques para tabla interna y salida.
  - Costo = $\lceil b_r / (M-2) \rceil * b_s + b_r$ transferencias de bloques + $2 \lceil b_r / (M-2) \rceil$ accesos a bloque

Operación de reunión selectiva

- **Algoritmo de reunión selectiva de loop anidado indexado**
  - Asumir que hay un índice en el atributo de la relación interna de la reunión.
  - Para cada tupla t_r en la tabla externa *r* usar el índice para buscar tuplas en *s* que satisfacen la condición de reunión con tupla t_r.
  - **Peor caso**: el búfer tiene espacio para solo un bloque de *r* y un bloque del índice y para cada tupla en *r* hacemos búsqueda en índice de *s*.
    - Costo de la reunión selectiva: $b_r (t_T + t_S) + n_r * c$
    - Para leer *r* en el peor caso hay b_r accesos a bloque y b_r transferencias de bloque.
    - Aquí *c* es el costo de recorrer el índice y recolectar todas las tuplas de *s* que cazan para una tupla de *r*.
    - El valor *c* puede estimarse como el costo de una selección en *s* usando la condición de la reunión.

Operación de reunión selectiva

- **Algoritmo de reunión merge-sort:**
- Supongamos que hay un atributo de la reunión.
- Ordenar ambas tablas en el atributo de la reunión.
- Hacer merge de las relaciones ordenadas para hacer la reunión.
  - El paso de reunión es similar a la etapa de merge del algoritmo merge sort.
  - La principal diferencia es manejo de valores duplicados en el atributo de la reunión – cada par con el mismo valor de atributo del join debe ser juntado.
  - Leer detalles del algoritmo en el libro.
- Cada bloque necesita ser leído una sola vez asumiendo que todas las tuplas para un valor dado de los atributos del join entran en memoria.
- Entonces el costo de reunión merge-sort es:
  - $b_r + b_s$ transferencia de bloques $+\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ accesos a bloque
  - Donde b_b bloques de búfer son asignados a cada tabla.
  - + el costo de ordenar si las tablas no están ordenadas.

Operación de reunión selectiva

- **Algoritmo de reunión merge-sort híbrido:**
- **Suposición**: la primera tabla está ordenada y la segunda tabla está desordenada pero tiene un índice secundario árbol B+ en los atributos de la reunión.
- **Algoritmo**:
  1. El algoritmo combina la tabla ordenada con las entradas hoja del índice secundario B+.
  2. El archivo resultante contiene tuplas de la relación ordenada y direcciones para las tuplas de la relación no ordenada.
  3. Se ordena el resultado por las direcciones de las tuplas de la relación no ordenada, permitiendo el retorno eficiente de las tuplas correspondientes en el orden de almacenamiento físico para terminar la reunión.
- **Costo**:
  - Segundo paso: Costo de ordenar el archivo resultante.
  - Tercer paso: costo de acceso a tuplas en el orden de almacenamiento físico

Agregación

- **Algoritmo**:
  - **Agregación** puede ser implementada de una manera parecida a eliminación de duplicados.
  - Ordenación puede ser usada para juntar tuplas del mismo grupo y luego las funciones de agregación pueden ser aplicadas a cada grupo.
  - **Optimización**:
    - En lugar de recolectar todas las tuplas en un grupo y luego aplicar operadores de agregación, se puede implementar los operadores de agregación sum, min, max, count y avg al vuelo a medida que los grupos van siendo construidos.
    - Para el caso de sum, min, max, cuando dos tuplas en el mismo grupo son encontradas, el sistema las reemplaza por una sola tupla conteniendo el sum, min, o max respectivamente de las columnas siendo agregadas.
    - Para count se mantiene un count de cada grupo para el cual una tupla ha sido encontrada.
    - Para avg, se computan sum and count al vuelo y se divide sum por count al final.
- **Costo**: es el mismo que el costo de eliminación de duplicados.

Concatenación

- Concatenación requiere leer ambas tablas: primero la de la izquierda y luego la de la derecha.
  - A medida que se lee una tabla se genera el resultado.
- En el peor caso para computar r++s vamos a necesitar:
  - $b_r + b_s$ transferencias de bloques y accesos a bloques
- Si el resultado es un resultado intermedio: se escriben $b_r + b_s$ bloques en disco.

intersección

- **Algoritmo**:
  - Para calcular la intersección de dos tablas, podemos primero **ordenarlas** a ambas por la clave primaria.
  - Luego podemos escanear una vez cada tabla ordenada para producir el resultado.
- **Costo**: Suponiendo las tablas r, s ordenadas de ese modo, $r \cap s$ requiere:
  - Si un solo bloque en búfer se usa por tabla:
    - $b_r + b_s$ transferencias de bloques
    - $b_r + b_s$ accesos a bloque
  - El número de accesos a bloque puede ser reducido alojando bloques extra en búfer.

Resta

- **Algoritmo**:
  - Para calcular la resta de dos tablas, podemos primero **ordenarlas** a ambas por la clave primaria.
  - Luego podemos escanear una vez cada tabla ordenada para producir el resultado.
- **Costo**:
  - Suponiendo las tablas r, s ordenadas de ese modo.
  - Si un solo bloque en búfer se usa por tabla:
    - $b_r + b_s$ transferencias de bloques
    - $b_r + b_s$ accesos a bloque

Encauzamiento

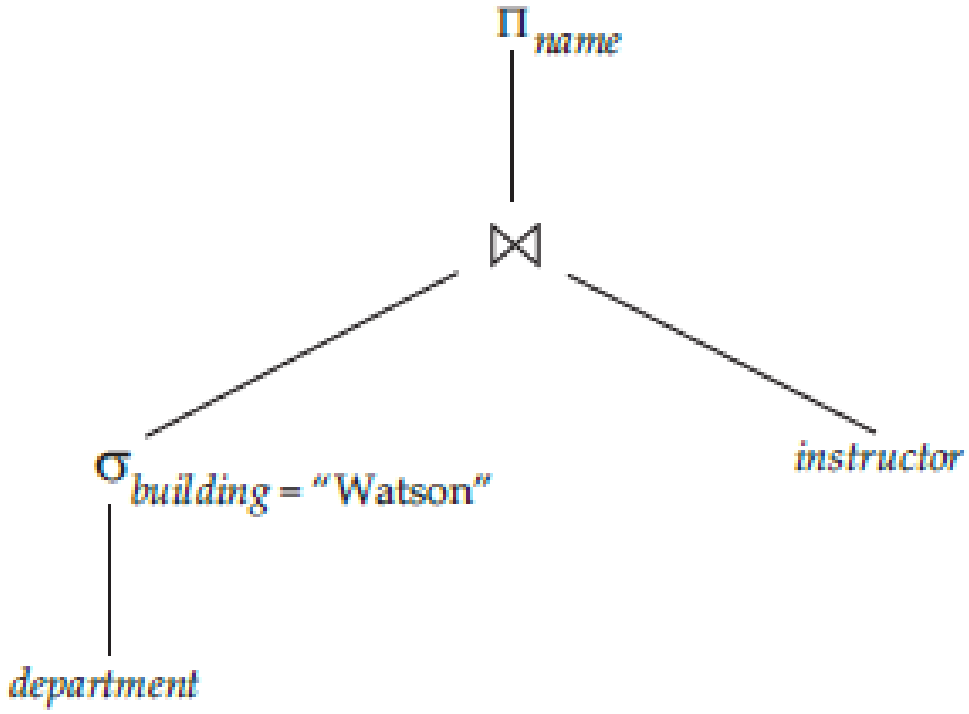
- **Meta**: mejorar la eficiencia de evaluación de consultas reduciendo el número de archivos temporales que son producidos.
- **Idea de encauzamiento**: se pueden combinar varias operaciones del AT en una tubería de operaciones en la cual los resultados de una operación son pasados para la siguiente operación de la tubería.
- **Ejemplo**: Sea $\Pi_{A, B}(r \bowtie s)$.
  - Con encausamiento cuando la operación de reunión genera una tupla de su resultado, pasa esta tupla a la operación proyección para procesamiento y se crea resultado final directamente.

Encauzamiento

- **Beneficios de encauzamiento**:
  1. Elimina el costo de leer y escribir tablas temporales, reduciendo así el costo de evaluación de consultas.
  2. Puede comenzar generando resultados rápidamente, si el operador raíz de un plan de evaluación de consulta es combinado en una tubería con sus inputs.
     - Esto es útil si los resultados son mostrados al usuario a medida que son generados.
- Es deseable en general reutilizar el código para operaciones individuales en la construcción de una tubería.

Encauzamiento

- Considerar el siguiente árbol de ejecución:
- Las 3 operaciones pueden colocarse en una tubería:
  - Esta pasa los resultados de la selección a la reunión natural a medida que son generados.
  - Los resultados de la reunión natural son pasados a la proyección a medida que son generados.
- Los requisitos de memoria son bajos porque los resultados de una operación no son almacenamos por mucho tiempo.
- Los inputs de operaciones no están disponibles todos a la vez para procesamiento.



Encauzamiento

- La manera más usada para ejecutar encausamiento se llama **tubería dirigida por demanda.**
  - El sistema hace **pedidos repetidos de tuplas** desde la operación en el tope de la tubería.
  - Cada vez que una operación recibe un pedido de tuplas, computa la próxima tupla o tuplas a ser retornadas, y luego retorna esa tupla.
  - Si los inputs de la operación no están en la tubería, las próximas tuplas a ser retornadas pueden ser computadas a partir de las tablas de entrada, mientras el sistema mantiene la pista de qué ha sido retornado.
  - Si la operación tiene algunos inputs en tubería, entonces la operación hace **pedidos de tuplas** de esos inputs en tubería.
  - Usando las tuplas recibidas de sus inputs en tubería, la operación computa las tuplas para su salida y las pasa arriba a su padre.

Encauzamiento

- **Implementación de tubería dirigida por demanda.**
  - Cada operación en la tubería puede implementarse como un **iterador** que provee las siguientes funciones en su interfaz: abrir(), siguiente(), y cerrar().
  - Un iterador produce la salida de una tupla por vez.
  - Varios iteradores pueden estar activos en un tiempo dado, pasando resultados hacia arriba en el árbol de ejecución.
  - El plan de consulta puede ejecutarse invocando los iteradores en un cierto orden.
  - **Interfaz de un iterador:**
  - **Abrir():**
    - inicializa el iterador alojando búferes para su entrada y salida e inicializando todas las estructuras de datos necesarias para el operador.
    - Además llama abrir() para todos los argumentos de la operación.

Encauzamiento

- **Implementación de tubería dirigida por demanda (cont.).**
  - **Siguiente():**
    - cada llamada a **siguiente()** retorna la próxima tupla de salida de la operación;
    - además ajusta las estructuras de datos para permitir que tuplas subsiguientes sean obtenidas.
    - Siguiente() ejecuta el código específico de la operación siendo realizada en los input.
    - Además llama siguiente() una o más veces en sus argumentos.
    - En **siguiente()** el **estado del iterador** es actualizado para mantener la pista de la cantidad de input procesado.
    - Cuando no se pueden retornar más tuplas se retorna un valor especial: **NotFound**.

Encauzamiento

- **Implementación de tubería dirigida por demanda (cont.).**
  - **Cerrar():**
    - termina la iteración luego que todas las tuplas que pueden ser generadas han sido generadas, o el número requerido de tuplas ha sido retornado.
    - Se llama cerrar() en todos los argumentos del operador.
  - La implementación de una operación llama abrir() y siguiente() en sus inputs para obtener las tuplas de los input cuando son necesitadas.
  - Cuando describimos iteradores y sus métodos asumimos que hay una clase para cada operador físico implementado como un iterador y la clase define abrir(), siguiente() y cerrar() en las instancias de la clase.

Encauzamiento

- **Ejemplo**: Iterador para escaneo de tabla R

```
Open() {
    b := the first block of R;
    t := the first tuple of block b;
}

GetNext() {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block)
            RETURN NotFound;
        ELSE /* b is a new block */
            t := first tuple on block b;
    }  /* now we are ready to return t and increment */
    oldt := t;
    increment t to the next tuple of b;
    RETURN oldt;
}

Close() {
}
```

Encauzamiento

- **Ejemplo**: **escaneo ordenado** de tabla R. Se leen tuplas de R y se las retorna en forma ordenada.
  - No se puede retornar la primera tupla hasta que se ha examinado cada tupla de R. Asumimos que R entra en memoria principal.
  - Abrir() lee R en memoria principal y ordena las tuplas de R.
  - Siguiente() retorna cada tupla en el orden de ordenación.

Encauzamiento

- **Ejemplo**: concatenación R ++ S. Sean *R* y *S* iteradores que producen tuplas de R y S. *R* y *S* son escaneos de tablas.

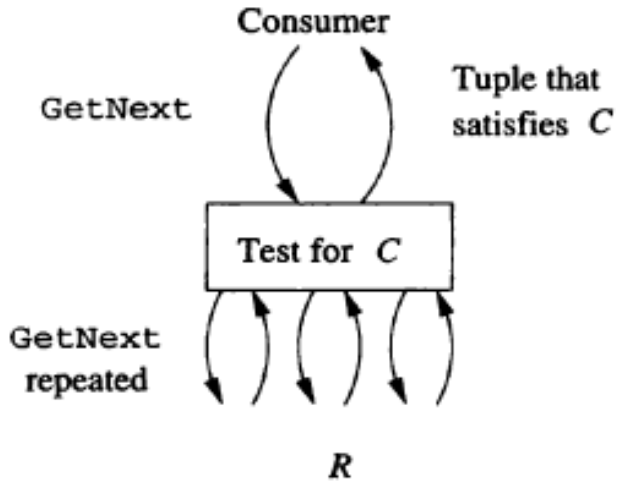
```
Open() {
    R.Open();
    CurRel := R;
}

GetNext() {
    IF (CurRel = R) {
        t := R.GetNext();
        IF (t <> NotFound) /* R is not exhausted */
            RETURN t;
        ELSE /* R is exhausted */ {
            S.Open();
            CurRel := S;
        }
    }
    /* here, we must read from S */
    RETURN S.GetNext();
    /* notice that if S is exhausted, S.GetNext()
       will return NotFound, which is the correct
       action for our GetNext as well */
}

Close() {
    R.Close();
    S.Close();
}
```

Encauzamiento

- **Ejemplos**:
  - **iterador implementando selección usando búsqueda lineal:**
    - **Abrir()** comienza el escaneo del archivo R y el estado del iterador guarda el punto en el cual el archivo ha sido escaneado.
    - Cuando se llama **siguiente()** el escaneo del archivo continua después del punto previo.
    - Puede ser necesario llamar siguiente() varias veces en R hasta que una tupla que satisface la condición de la selección es encontrada.
    - Cuando la próxima tupla satisfaciendo la selección es encontrada por escanear el archivo R, la tupla es retornada luego de almacenar el punto donde fue encontrada en el estado del iterador.



Esta figura muestra como funciona el iterador de selección (rectángulo *test for C*) usando iterador para R.

Encauzamiento

- **Ejemplo: Iterador para proyección:**
  - Supongamos hay tabla de origen R con iterador.
  - Siguiente() del iterador de proyección llama siguiente() en iterador de R y proyecta la tupla obtenida apropiadamente y retorna el resultado al consumidor.
- **Ejemplo: Iterador implementando reunión por combinación (merge-sort join):**
  - Asumo como inputs tablas R y S con iteradores de escaneo ordenado.
  - **Abrir()** va a llamar R.abrir() y S.abrir() o sea, que si estos no están ordenados, entonces van a ordenarse.
  - Cuando se llama **siguiente()** sobre la reunión, va a retornar el siguiente par de tuplas que cazan.
    - La **información de estado** va a consistir de hasta donde cada tabla de input ha sido escaneada.