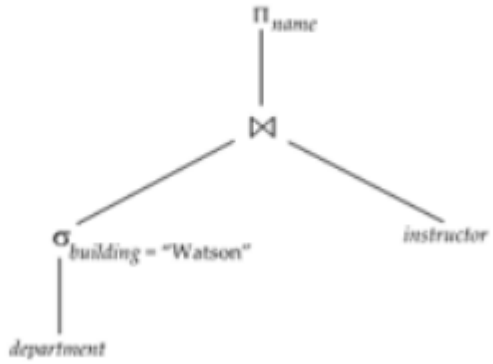
Capítulo 4 - Procesamiento de consultas

Árbol binario de ejecución

- Es el árbol binario de una expresión de consulta (en álgebra de tablas)

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



  - Nodos hoja $\rightarrow$ tablas de la BD
  - Nodos no hoja / interos $\rightarrow$ operadores del álgebra de tablas
- Su evaluación (en términos del costo de la consulta) va a estar en términos de los operadores físicos de la BD (algoritmos específicos para operadores del álgebra de tablas, hacen uso de índices, tamaños de búfer en memoria, etc.)
- En esta parte vamos a considerar un plan de evaluación particular para una consulta **sin ningún tipo de optimización**

Convenciones para medidas de costos

- Como costo, se contarán:
  - **Número de transferencia de bloques de disco**
    - Asumimos que todas las transferencias de bloques tienen el mismo costo
    - No diferenciamos entre transferencias de bloques de lectura vs. de escritura (aunque cueste más escribir que leer)
  - **Cantidad de accesos a bloques**
    - Tiempo que le lleva a la cabeza lectora posicionarse en el bloque deseado
  - Si asumimos tamaño de bloque de $4KiB$ y tasa de transferencia de $40MBps$
    - Transferencia de bloques $\rightarrow 0.1ms$
    - Acceso a bloque $\rightarrow 4ms$
- **Tamaño del búfer** en memoria principal
  - Asumimos el peor caso $\rightarrow$ el búfer solo puede sostener un bloque por tabla

Evaluación del árbol binario de ejecución

- El resultado de evaluar un operador de un nodo interno del árbol binario de ejecucuón que **no** es la raíz se llama *resultado intermedio*
- Hay **dos enfoques**

- **Materialización** $\rightarrow$ los resultados intermedios se guardan en disco en tablas temporales (sin índices)
    - Memoria, + Disco
- **Encauzamiento** $\rightarrow$ a medida que se van generando resultados intermedios, se van pasando al siguiente operador (**pipeline**)
    - Memoria, - Disco

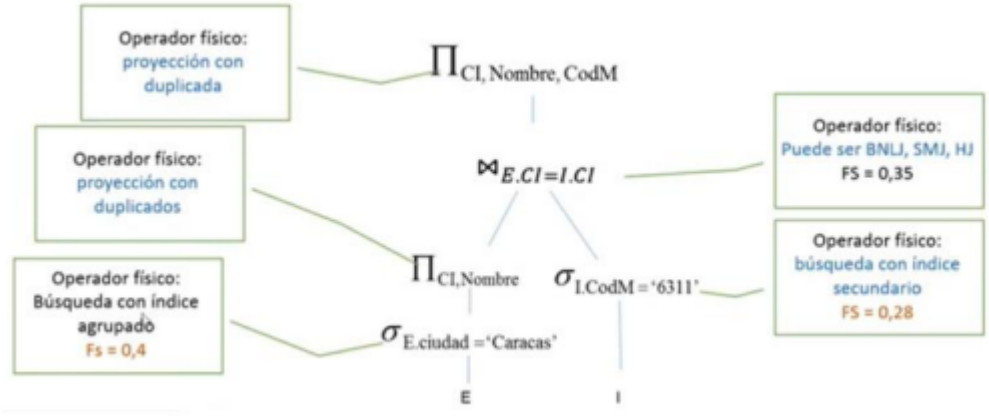
Materialización

Conceptos importantes a tener en cuenta

- Para estimar el tamaño de los resultados intermedios en cantidad de bloques a escribir a disco para los *operadores de selección y reunión (selectiva y natural)*, vamos a tener en cuenta la función de probabilidad **factor de selectividad**
    - Si se usa el predicado P y el input del operador es i, entonces el factor de selectividad es $fs(P, i)$
        - Asumimos *uniformidad* e *independencia*
    - Cantidad de registros del resultado intermedio
        - **Selección** $\rightarrow |r| \times fs(P, r)$
        - **Reunión** $\rightarrow |r| \times |s| \times fs(P, r, s)$
- Si tenemos r registros de tamaño k, y el tamaño del bloque es b, entonces el **número de bloques** que consideramos es $\lceil \frac{r \times k}{b} \rceil$

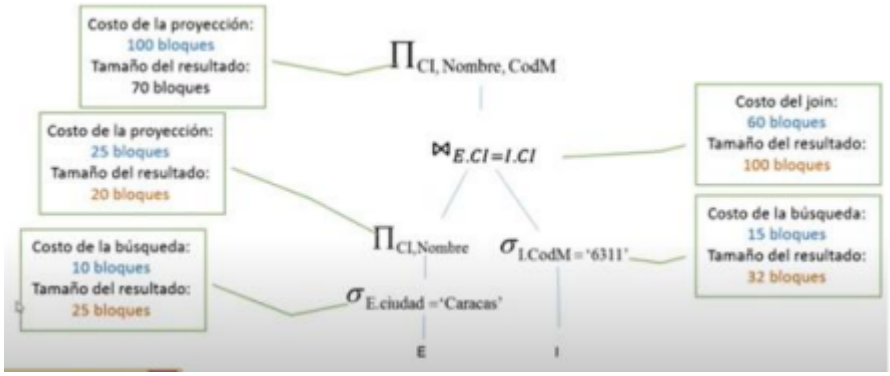
Esquema de fases

- **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)



- **Fase 2**: estimar el costo de ejecutar el plan de evaluación
    - Calcular el tamaño en bloques de las tablas de la BD (hojas)
    - Calcular el tamaño de los resultados intermedios en bloques (de abajo hacia arriba)
    - Calcular el costo de los operadores físicos

- Sumar los costos totales (transferencia de bloques + acceso a bloques)



- Es decir, $costoTotal = \sum costo(operaciones) + \sum costo(materializacion)$
  - Si multiplicamos por velocidad de transferencia, tenemos el tiempo
    - $Costo\ total = \sum costo(operaciones) + \sum costo(materialización)$
    - Costo total = (10+20+15+60+100) + (25+20+32+100)
    - Costo total = 382 (accesos a disco)

Encauzamiento

- Estilo **pipeline** (se pasa cada tupla de un nodo *hijo* a su *padre* para que lo procese)
- Pros
  - Elimina el costo de leer y escribir tablas temporales, reduciendo el costo de evaluación de consultas
  - Puede comenzar generando resultados rápidamente si el operador root de un plan de evaluación de consulta es combinado en una pipeline con sus inputs
    - Útil también si los resultados son mostrados al usuario a medida que son generados
  - Los requisitos de memoria son bajos porque los resultados de una operación no son almacenados por mucho tiempo
- Contras
  - Los inputs de operaciones no están disponibles todos a la vez para procesamiento
- *Implementación*
  - Cada operación en el pipeline puede implementarse como un **iterador** que provee las siguientes funciones en su interfaz:
    - $abrir()$
      - Inicializa el iterador alojando búferes para su E/S e inicializando todas las estructuras de datos necesarias para el operador
      - Además, llama $abrir()$ para todos los argumentos de la operación
    - $siguiente()$
      - Cada llamada a $siguiente()$ retorna la próxima tupla de salida de la operación (ejecuta el código específico de la operación siendo realizada en los inputs)
        - Ajusta las estructuras de datos para permitir que tuplas subsiguientes sean obtenidas
      - Llama $siguiente()$ una o más veces en sus argumentos
      - 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**
    - $cerrar()$

        - Termina la iteración luego de 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
    - Un iterador produce la salida de una tupla por vez

Iterador de escaneo de tabla R

```
abrir() {
        b := primer bloque de R;
        t := primera tupla de b;
}

siguiente() {
        if (t es pasada la última tupla de b) {
                incrementar b al próximo bloque;
                if (no hay próximo bloque)
                        return NotFound;
                else t := primera tupla de b
        }

        t_ret := t;
        incrementar t a la próxima tupla de b;
        return t_ret;
}

cerrar() {}
```

Iterador de concatenación ($R++S$)

```
abrir() {
        R.abrir();
        tabla_actual := R;
}

siguiente() {
        if (tabla_actual = R) {
                t := R.siguiente();
                if (t != NotFound)
                        return t;

                S.abrir();
                tabla_actual := S;
        }

        return S.siguiente();
```

```
}

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

Iterador de selección (búsqueda lineal)

Consideramos condición C y tabla R

```
abrir() {
        R.abrir();
}

siguiente() {
        t := R.siguiente();
        while (C(t) no se cumpla && t != NotFound)
                t := R.siguiente();

        return t;
}

cerrar() {
        R.cerrar();
}
```

Iterador de reunión por combinación (merge-sort join)

Consideramos que la lectura de R y S se hace de forma ordenada

```
abrir() {
        R.abrir();
        S.abrir();

        val_act := None
        r_act := None
        s_act := None
}

siguiente() {
        if (r_act = None)
                r_act := R.siguiente();

        if (s_act = None)
                s_act := S.siguiente();
```

```
        if (r_act = NotFound && s_act = NotFound) {
                val_act := NotFound;
        } else if (r_act = NotFound) {
                val_act := s_act;
                s_act := None;
        } else if (s_act = NotFound) {
                val_act := r_act;
                r_act := None;
        } else if (r_act <= s_act) {
                val_act := r_act;
                r_act := None;
        } else {
                val_act := s_act;
                s_act := None
        }

        return val_act;
}

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