Pauta / Examen / Bases Datos

9 de Diciembre 2008 / Prof. C. Gutiérrez / Aux. M. Monsalve / Ayu. D. Díaz

Problema 1

El siguiente conjunto de esquemas modela los pedidos de una tienda que reparte comida a domicilio:

```
Pedido(id,dir_nro,dir_calle,dir_comuna)
Pedido_Productos(id,tipo,valor)
Repartidor(rut,nombre,apellido)
Reparto(rut,id)
```

Usando álgebra relacional, conteste las siguientes consultas:

1. Los domicilios (número, calle y comuna) que han hecho más de tres pedidos.

   Solución. Se puede resolver con o sin agregación. Aquí nos enfocamos al caso agregado:

   $$\mathcal{C} \leftarrow {}_{<dir\_nro,dir\_calle,dir\_comuna>}\mathcal{G}_{<dir\_nro,dir\_calle,dir\_comuna,COUNT(\star)\,as\,N>}\texttt{Pedidos}$$

   $$\mathcal{R} \leftarrow \pi_{<dir\_nro,dir\_calle,dir\_comuna>}\sigma_{N\geq 3}\mathcal{C}$$

   La solución sin agregación consiste en hacer tres productos cartesianos de `pedido` que tengan la misma dirección pero diferente `id`.

2. El costo de cada pedido (la suma del valor de los productos que lo conforman).

   Solución. Tenemos que usar agregación para esta consulta:

   $$\mathcal{R} \leftarrow {}_{<id>}\mathcal{G}_{<id,SUM(valor)>}\texttt{Pedidos} \bowtie \texttt{Producto\_Pedidos}$$

3. El repartidor que ha repartido a menos comunas, o a ninguna.

   Solución. Hay muchas maneras de contestar esta pregunta. Pero consiste básicamente en obtener el repartidor que ha repartido a menos comunas, y ver el caso en que no reparte.

   Para propósitos de correccion, **obtener el mínimo sin lidiar con el caso extremo sólo otorga la mitad del puntaje**. De esta manera, el alumno puede contestar parcialmente la pregunta.

Una solución sencilla es inventarse una comuna ficticia, a la que todos los repartidores han "visitado". Luego, al obtener el repartidor que ha visitado menos comunas también cubrimos el caso del repartidor que no ha visitado ninguna comuna (pues ahora visitó la comuna ficticia).

$$\mathcal{A} \leftarrow (\pi_{<rut>}\texttt{Repartidor}) \times \{"\#"\}$$
$$\mathcal{B} \leftarrow \pi_{<rut,dir\_comuna>}(\texttt{Reparto} \bowtie \texttt{Pedido})$$
$$\mathcal{C} \leftarrow {}_{<rut>}\mathcal{G}_{<rut,COUNT(*)\,as\,N>}(\mathcal{A} \cup \mathcal{B})$$
$$\mathcal{D} \leftarrow \mathcal{G}_{<MIN(N)\,as\,M>}\mathcal{C}$$
$$\mathcal{R} \leftarrow \pi_{<rut>}(\mathcal{C} \bowtie_{N=M} \mathcal{D})$$

Problema 2

Considere el esquema de relación

$$R(A, B, C, D, E, F, G)$$

y el conjunto de dependencias funcionales

$$F = \{AB \to F, BC \to D, E \to DG\}$$

Normalice R en Forma Normal de Boyce-Codd (FNBC).

Solución. Primero, determinemos la llave. Tenemos los siguientes atributos no determinados funcionalmente: A, B, C, E. Aplicando las dependencias funcionales de F en orden, tenemos:

$$ABCE \to ABCEF \to ABCDEF \to ABCDEFG$$

Luego, $ABCE$ es la única llave candidato.

Segundo, notamos que (R, F) no está en FNBC. Ninguna dependencia funcional comienza con $ABCE$, luego no se cumple $X \to Y$ con X superllave. Es necesario normalizar.

Usando las dependencias funcionales en orden:

$AB \to F$:

$$\mathbf{ABCDEFG} \Rightarrow \mathbf{ABCDEG} + \mathbf{ABF}$$

$BC \to D$:

$$\mathbf{ABCDEG} + ABF \Rightarrow \mathbf{ABCEG} + \mathbf{BCD} + ABF$$

$E \to DG$:

$$\mathbf{ABCEG} + BCD + ABF \Rightarrow \mathbf{ABCE} + \mathbf{EG} + BCD + ABF$$

Así hemos obtenido la siguiente descomposición en FNBC:

$$R_1(A, B, C, E), R_2(E, G), R_3(B, C, D), R_4(A, B, F)$$

(y curiosamente no preserva la dependencia $E \to D$).

Problema 3

Considere el problema de una implementación que permita evaluar la división en álgebra relacional.

1. ¿Qué índices podrían ser usados y cómo en la evaluación de la división? Limítese a: archivos ordenados, árboles B+ y hashing.

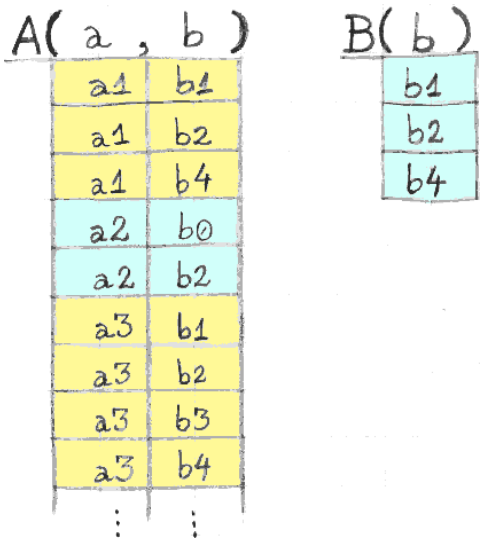
   Solución. Había que presentar ideas y/u observaciones para cada índice. Se podía notar que la búsqueda por rangos es inútil, que la búsqueda por igualdad es esencial, que la indización tiene que ser sobre los atributos adecuados y que los ordenamientos y el control de repetidos (que se puede hacer a través del uso de índices) aumenta notoriamente la calidad de los algoritmos, en términos de eficiencia.

2. Proponga una estrategia para el caso sin índices. Use pseudocódigo.

   Solución. Si no contamos con índices, siempre podemos crearlos en el vuelo de la manera que más nos acomode. (Aunque parece que varios alumnos entendieron que no podían usar ninguna estructura de datos o presentación de los datos. Igual se les revisó con *bondad* sus algoritmos iterativos.)

Algunas ideas de algoritmos. Este punto está separado porque estas ideas son aplicables a ambas partes del problema.

Primero, consideremos el caso sencillo de archivos ordenados. Sea $A(a, b)$ y $B(b)$. Para calcular $A \div B$, podemos ordenar A por a y luego por b, y podemos tener B ordenado (por b). Estudiemos una situación como la siguiente:



¿Cómo recorremos A y B para obtener $A \div B$? Hay dos maneras. Una es, por cada $ab \in A$, buscar $b \in B$ usando búsqueda binaria. La otra es con pasadas secuencuales en B, ya que, para un a fijo, A y B están ordenados por b. Simulemos este último caso:

- Situamos un puntero al principio de A y uno al principio de B.
- En A, vemos $(a1, b1)$. **Estamos trabajando con a1.**

  - Recorremos B en busca de $b1$. Está al principio de B, así que avanzamos su puntero, quedándonos en $b2$.
- Avanzamos A y quedamos en $(a1, b2)$.

  - Buscamos $b2$ en B y lo hallamos inmediatamente. Avanzamos B, y quedamos en $b4$.
- Avanzamos A y quedamos en $(a1, b4)$.

  - Buscamos $b4$ en B y lo hallamos inmediatamente. Avanzamos B, vemos que lo terminamos: **entonces aceptamos a1 como parte de A ÷ B.** Reiniciamos el puntero de B.
- Avanzamos A en busca de un a diferente ($a1$ ya fue aceptado). El siguiente elemento es $(a2, b0)$. **Estamos trabajando con a2.**

  - Buscamos $b0$ en B. El puntero de B está en $b1$, y $b1 > b0$ así que no hacemos nada. (No podemos descartar $a2$ por esto.)
- Avanzamos A y quedamos en $(a2, b2)$.

  - Buscamos $b2$ en B. El puntero de B está en $b1$, y $b1 < b2$, o sea, **no existe (a2,b1) en A,** luego **descartamos a2.** Reiniciamos el puntero de B.
- Avanzamos A hasta buscar el siguiente a (distinto de $a2$). Nos encontramos con $(a3, b1)$. **Estamos trabajando con a3.**

  - Buscamos $b1$ en B, cuyo puntero está situado en $b1$. Avanzamos B hasta $b2$.
- Avanzamos A y quedamos en $(a3, b2)$.

  - Buscamos $b2$ en B, cuyo puntero está situado en $b2$. Avanzamos B hasta $b4$.
- Avanzamos A y quedamos en $(a3, b3)$.

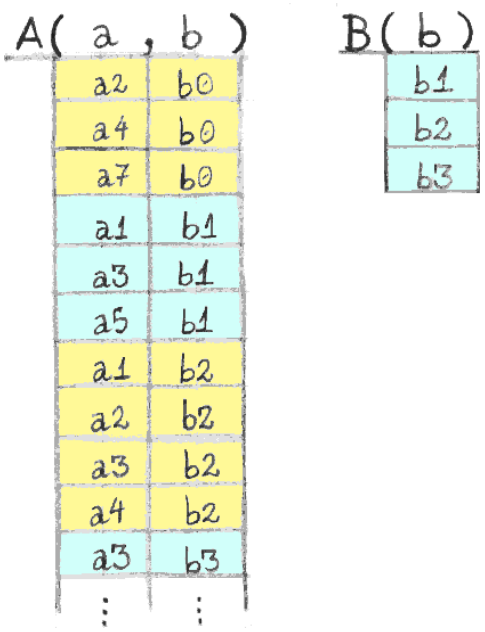
  - Buscamos $b3$ en B, cuyo puntero está situado en $b4$. Como $b4 > b3$, no avanzamos B.
- Avanzamos A y quedamos en $(a3, b4)$.

  - Buscamos $b4$ en B, cuyo puntero está situado en $b4$. Como llegamos al final de B, **aceptamos a3 como parte de A ÷ B.** Reiniciamos el puntero de B.
- Avanzamos A y...

El algoritmo para el caso anterior es sencillo: *usamos dos punteros, uno para A y uno para B. Tomamos el primer a que hallamos en A (en tuplas de la forma (a, b)). Vemos en B si el b está presente; si el b que buscamos es menor o igual al que apuntamos en B, no hacemos nada, pero si es mayor, rechazamos a. Si a no ha sido rechazado, avanzamos A en uno y repetimos lo anterior, hasta completar B (si es posible). Si a es rechazado, reiniciamos el puntero de B y avanzamos A hasta el siguiente a.*

La diferencia, en tiempo, de aplicar en B búsqueda binaria o secuencial depende de la naturaleza de los datos. (Es una complejidad adaptativa.) Búsqueda binaria ofrece mucha más estabilidad mientras que búsqueda secuencial ofrece tiempos muy cortos en los mejores casos.

Pero podemos mejorar los casos anteriores. Ordenemos $A(a, b)$ primero por b y luego por a, y que B siga ordenado. Estudiemos una situación como la siguiente:



Con este ordenamiento podemos descartar rápidamente los a que no serán parte de $A \div B$, pero debemos mantenerlos en memoria (primaria o secundaria). Por ejemplo, podríamos crear un índice de hashing sólo para esto. Pero vayamos a nuestro caso:

- Ubicamos un puntero al inicio de A y uno al inicio de B.
- A está en $(a2, b0)$. **Trabajamos con b0.**

  - Buscamos $b0$ en B, el que está situado en $b1$. Como $b0 \neq b1$, **descartamos trabajar con b0.** Reiniciamos B (que ya estaba al principio).
- Avanzamos A hasta un nuevo b. Llegamos a $(a1, b1)$. **Trabajamos con b1.**

  - B apunta a $b1$, lo que está OK. Avanzamos B.

  - Este es el primero valor serio de b. Llenamos nuestro conjunto X con los valores de A que encontremos. Por ahora, $X = \{a1\}$.

- Avanzamos A y llegamos a $(a3, b1)$. b no ha cambiado, así que agregamos $a3$ a X: $X = \{a1, a3\}$.
- Avanzamos A y llegamos a $(a5, b1)$. b no ha cambiado, así que agregamos $a5$ a X: $X = \{a1, a3, a5\}$.
- Avanzamos A y llegamos a $(a1, b2)$. b cambió. **Trabajamos con b2.**
  - B apunta a $b2$, lo que está OK. Avanzamos B.
  - Desde ahora, sólo marcaremos y removeremos elementos de X.
  - Borramos los elementos marcados de X (ninguno).
  - Marcamos X: $X = \{a1*, a3*, a5*\}$.
  - Trabajamos con $a1$, así que lo desmarcamos: $X = \{a1, a3*, a5*\}$.
- Avanzamos A y llegamos a $(a2, b2)$. $a2$ no está en X, así que no hacemos nada.
- Avanzamos A y llegamos a $(a3, b2)$. $a3$ está en X, luego lo desmarcamos: $X = \{a1, a3, a5*\}$.
- Avanzamos A y llegamos a $(a4, b2)$. $a4$ no está en X, así que no hacemos nada.
- Avanzamos A y llegamos a $(a3, b3)$. b cambió. **Trabajamos con b3.**
  - B apunta a $b3$, lo que está OK. Avanzamos B.
  - Borramos los elementos marcados de X ($a5$). Queda: $X = \{a1, a3\}$.
  - Marcamos X. Queda: $X = \{a1*, a3*\}$
  - Trabajamos con $a3$, así que lo desmarcamos: $X = \{a1*, a3\}$.
- Avanzamos A y ...

El algoritmo anterior se puede resumir en: *usamos dos punteros, uno para A y uno para B. Luego hacemos un algoritmo de dos fases.*

FASE 1: buscamos en A el primer b que sea igual al primero de B. Si no hay, devolver $\emptyset$ y terminar. Si lo hay, avanzar B y crear $X = \emptyset$. Avanzar en A y, mientras b no cambie, agregar los elementos a X: $X = X \cup \{a*\}$.

FASE 2: para cada nuevo b, vemos si está en B. Si es anterior al de B, avanzar A hasta un nuevo $B4$. Si es mayor, devolver $\emptyset$ y terminar. Si es igual, borrar los elementos marcados de X y marcar los que quedaron.

Si b no cambia, y si $a \in X$, desmarcar a en X. Avanzar A.

FINAL: devolver X.

El algoritmo anterior es bastante eficiente: es lineal en el tamaño de A y B, pero le queda el costo de manejar X. Afortunadamente, X se puede manejar con índices, como tablas de hashing y árboles B+.

Los algoritmos anteriores son igualmente cómodos de trabajar con árboles B+. Recordemos que, a partir de las hojas de un árbol B+, podemos obtener algo parecido a un archivo ordenado.

Usar hashing no entrega especial ayuda en cuanto a revisar ordenadamente los archivos, pero agiliza el acceso.

También se pueden combinar índices. Por ejemplo, en vez de llevar un puntero en B en los algoritmos presentados en detalle, se pudo tener un índice de hashing en éste, asegurando tiempos de acceso bastante cortos (aunque mayores).

El uso de los algoritmos de fuerza bruta, que tendían a comparar todos con todos no es muy deseable. Sus optimizaciones radican en usar buenas estructuras de datos para ver los a resultantes o usar operaciones de álgebra relacional de manera eficiente, adecuada al caso. En este ejemplo:

$$A \div B = \pi_{A \div B} A - \pi_{A \div B} \left((\pi_{A \div B} A) \times B - A \right)$$

se podría haber optimizado el gran paréntesis del lado derecho, incluso completando índices en el vuelo para acelerar la última sustracción a realizar (la de más afuera).

En síntesis, hay bastantes maneras de evaluar la división en álgebra relacional de manera eficiente. Las eficiencias varían de algoritmo a algoritmo, pero hay bastantes chances de vencer a la fuerza bruta. No se especificarán más algoritmos por una cuestión de extensión; debió quedar más que clara la gran variedad de algoritmos posibles.

Problema 4

Considere una base de datos con escritura forzada y las siguientes transacciones:

T1: $R(A), R(B), W(A), R(C), W(D)$
T2: $R(C), R(D), W(C)$
T3: $R(C), W(A), W(B), W(C), W(D)$

Usando estas transacciones:

1. Cree un plan que no sea serializable por conflictos.

   Solución. Hay que intercalar transacciones de manera de lograr conflictos, haciendo un ciclo. Lo más fácil es hacer esto entre dos transacciones. Por ejemplo, T1 y T2:

   $$R_2(C), R_1(A), R_1(B), W_1(A), \mathbf{R_2(D)}, \mathbf{R_1(C)}, \mathbf{W_1(D)}, \mathbf{W_2(C)}, T3...$$

   Otro ejemplo, con T2 y T3:

   $$R_3(C), W_3(A), W_3(B), \mathbf{W_3(C)}, \mathbf{R_2(C)}, \mathbf{R_2(D)}, \mathbf{W_2(D)}, \mathbf{W_3(D)}, T1...$$

   Como vemos, los planes anteriores tienen ciclos, por lo que no son serializables por conflicto.

2. Cree un plan que sea serializable por vistas.

   Solución. La solución más astuta consiste en hacer planes seriales, ¡los que obviamente son serializables! Ejemplo:

   $$T1, T2, T3$$

   $$T3, T1, T2$$

   $$...$$

   Una solución menos astuta, pero igualmente válida, es organizar un plan serializable por conflictos. Como sabemos, si el plan es serializable por conflictos, lo es por vistas.

3. Cree un plan no recuperable.

   Solución. Un plan no es recuperable cuando una transacción lee un dato modificado por otra transacción que no ha hecho *commit*.

   Por ejemplo, comencemos el plan no recuperable con T3 y T1:

   $$R_3(C), \mathbf{W_3(A)}, \mathbf{R_1(A)}, R_1(B), ...$$

   T3 escribe sobre A y, antes que termine su ejecución (y llegue al *commit*), T1 lee de A. Con esto violamos la recuperabilidad.

Críticas generales. Los alumnos no saben expresar sus ideas en papel. A diferencia de un texto en computador, un alumno puede construir esquemas de manera rápida y sencilla en el papel, combinando texto y figuras para comunicar ideas de manera efectiva. Pero esto no fue lo que ocurrió.