

NOMBRE: Benjamín Farías Valdés

N.ALUMNO: 17642531



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IIC3413 — Implementación de Sistemas de Base de Datos — 1' 2020

Control 2

Doy mi palabra que la siguiente solución de la pregunta 1 fue desarrollada y escrita individualmente por mi persona según el código de honor de la Universidad.

Figura 1: Firma

1. Bloom Filter

Cada uno de los **Bloom Filters** almacena a su conjunto respectivo, lo que se puede interpretar como:

$$\forall s \in S. \forall h_i : i \in (1 \dots k). B[h_i(s)] = 1$$

$$\forall s' \in S'. \forall h_i : i \in (1 \dots k). B'[h_i(s')] = 1$$

Esto significa que todas las posiciones dentro de los **bitmap** que representan el resultado de aplicar cualquiera de las funciones de **hash** sobre algún elemento de su conjunto almacenado, deberán necesariamente tener valor 1. Si queremos construir un nuevo **Bloom Filter** que contenga a ambos conjuntos, este debe cumplir con la siguiente condición:

$$\forall s^* \in (S \cup S'). \forall h_i : i \in (1 \dots k). B^*[h_i(s^*)] = 1$$

Como **no se conocen los conjuntos** S y S' , no es posible realizar supuestos sobre el tamaño de B^* , manteniéndolo entonces como n , y por lo tanto tampoco es posible modificar las funciones de hash utilizadas, ya que estas mapean al dominio $(0 \dots (n - 1))$. Sabiendo esto, es necesario construir B^* a partir de B y B' , considerando que su tamaño también será n y las funciones de hash serán las mismas que en ambos Bloom Filters.

Para esto basta con construir B^* mediante la unión de B y B' , de forma que todos los bits que estén en 1 en **cualquiera de los 2 bitmaps** también estén en 1 en el bitmap B^* . Esto se puede lograr directamente con el operador lógico bitwise **OR**:

$$B^* := B \text{ OR } B'$$

Por ejemplo, si $B = 01011$ y $B' = 11001$, entonces $B^* = B \text{ OR } B' = 11011$.

Por construcción de B^* , todos los elementos de los conjuntos S y S' estarán almacenados en él, por lo que **no existen Falsos Negativos**.

En cuanto a los **Falsos Positivos**, estos representan el **error** del nuevo Bloom Filter B^* , lo que será analizado a continuación.

Revisando el caso de construir B^* desde 0, se deberán almacenar en él a todos los elementos dentro de $S \cup S'$. Primero, gracias a que las funciones de hash son pseudo-aleatorias, la probabilidad de que la i -ésima celda del bitmap sea 0 tras una cierta cantidad de inserciones distribuye **binomial**, por lo que tras insertar $|S \cup S'|$ elementos, dicha probabilidad corresponde a:

$$P(B^*[i] = 0) = \left(1 - \frac{1}{n}\right)^{k*|S \cup S'|}$$

Entonces, por complemento y las propiedades del número de **Euler**:

$$p = P(B^*[i] = 1) = 1 - \left(1 - \frac{1}{n}\right)^{k*|S \cup S'|} \approx 1 - e^{-\frac{k*|S \cup S'|}{n}}$$

La probabilidad de un Falso Positivo, asumiendo que una fracción $1 - p$ de celdas son 0, está acotada por:

$$P\left(\bigwedge_{j=1}^k B^*[i_j] = 1\right) \leq \prod p = p^k = \left(1 - e^{-\frac{k*|S \cup S'|}{n}}\right)^k$$

Ahora, revisando el caso de construir B^* a partir de B y B' , por construcción sabemos que todos sus bits que valen 1, poseen ese valor debido a que en alguno de los 2 bitmaps originales (o en ambos) esos bits también valían 1. Esto significa que si en ambos bitmaps originales había un 0 en el bit i , entonces en el nuevo bitmap también habrá un 0 en dicha posición. Por lo tanto, todos los bits de B^* que valen 1 están en la posición resultante de aplicar una de las funciones de hash a algún elemento de S o S' , por lo que es equivalente a construir el filtro mediante $|S \cup S'|$ inserciones desde 0, ya que cada una de estas inserciones es independiente de la otra. Siguiendo esto, finalmente se llega a la misma expresión para la probabilidad del error:

$$P\left(\bigwedge_{j=1}^k B^*[i_j] = 1\right) \leq \prod p = p^k = \left(1 - e^{-\frac{k*|S \cup S'|}{n}}\right)^k$$

NOMBRE: Benjamín Farías Valdés

N.ALUMNO: 17642531



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IIC3413 — Implementación de Sistemas de Base de Datos — 1' 2020

Control 2

Doy mi palabra que la siguiente solución de la pregunta 2 fue desarrollada y escrita individualmente por mi persona según el código de honor de la Universidad.

A handwritten signature in blue ink, appearing to be 'B. Farías', written over a horizontal line.

Figura 2: Firma

2. Joins

2.1. Block Nested Loop Left Outer Join

El **Left Outer Join** equivale a la unión de un **Natural Join** entre R y S con el conjunto de todas las tuplas de R que **NO hicieron match** al momento de realizar la operación **next()**, rellenando sus campos de S que no están en R con **null**. Si definimos al predicado p como la **igualdad entre todos los atributos en común de R y S** , entonces todas las tuplas resultantes del **Natural Join** son obtenidas mediante el algoritmo original de **Block Nested Loop Join**. Esto significa que falta agregar al procedimiento la funcionalidad de detectar a las tuplas de R que **NO hicieron match** con S , retornándolas como $(r, null)$.

Para implementar esto sin modificar la estructura característica del algoritmo, se puede agregar un **espacio de memoria adicional** al final del **Buffer**, denominado **Matched**, en el que se guarda una secuencia de valores **booleanos** que indica, **para cada tupla $r \in \text{Buffer}$** , si la tupla ya hizo match con S . Por simplicidad, se define la llamada **Matched(r)**, que entrega el valor booleano en Matched asociado a la tupla r del Buffer (**True** si la tupla r ya hizo match, y **False** en caso contrario).

Gracias a esta modificación en el Buffer, se puede implementar el **BNL Left Outer Join** mediante los siguientes cambios al procedimiento original:

- Dentro de la operación **fillBuffer()**, cuando el Buffer se **llene con tuplas**, se deberá añadir al final de este la secuencia de valores **Matched** que contiene los bools asociados a cada tupla dentro del Buffer, inicialmente con sólo valores **False**.

- Dentro de la operación **next()**, se deben realizar los siguientes cambios:
 - Cuando ocurra un **match** entre r y s , además de retornar la tupla (r, s) , se deberá actualizar el valor de **Matched** asociado a r : **Matched**(r) = **True**.
 - Una vez se sale del **while que itera sobre S** , y antes de la ejecución del siguiente **fillBuffer()**, se deberá iterar por sobre todas las tuplas r del Buffer accediendo a **Matched**(r), y en caso de que este sea **False**, se retorna la tupla $(r, null)$.

Mediante estos cambios, todas las tuplas r del Buffer que **NO hicieron match con S** , serán retornadas con valores nulos **antes de pasar al siguiente bloque de R** , logrando retornar todas las tuplas de R incluso si no pertenecen al **Natural Join**, lo que corresponde al **Left Outer Join**.

Al momento de calcular el costo, es posible notar que las únicas operaciones adicionales que se ejecutan en este algoritmo corresponden a accesos a las tuplas del Buffer y a la secuencia **Matched**. Como el Buffer está en memoria, no se requieren accesos I/O al disco, por lo que todas estas operaciones no afectan al costo del algoritmo. Finalmente, el costo de **BNL Left Outer Join** corresponde a:

$$cost(R \bowtie S) = cost(R) + \frac{pages(R)}{B} * cost(S)$$

Donde B corresponde a las páginas del Buffer sin considerar el espacio reservado al final para la secuencia **Matched**.

2.2. Sort-Merge Full Outer Join

El **Full Outer Join** equivale a la unión de un **Natural Join** entre R y S con el conjunto de todas las tuplas de R que **NO hicieron match con S** (rellenando sus campos de S que no están en R con **null**) y también con el conjunto de todas las tuplas de S que **NO hicieron match con R** (rellenando sus campos de R que no están en S con **null**). Si definimos al **predicado** como la **igualdad entre todos los atributos en común de R y S** , entonces todas las tuplas resultantes del **Natural Join** son obtenidas mediante el algoritmo original de **Sort-Merge Join**. Esto significa que falta agregar al procedimiento la funcionalidad de detectar a las tuplas de R y S que **NO hicieron match**, retornándolas como $(r, null)$ y $(null, s)$, respectivamente.

En la operación **next()** del algoritmo (**sin modificar**), se recorre un Buffer de tuplas t de S' , el que gracias al **while presente al final de fillBuffer()**, asegura que todas esas tuplas harán match con la tupla r actual, retornando las nuevas tuplas **(r, t)** (en caso de que el Buffer esté vacío, significa que no ocurrió match). Una vez se termina de recorrer el Buffer, se vuelve a recorrer cuantas veces sea necesario hasta que la siguiente tupla r cambie su valor en la parte izquierda del predicado (A). Cuando esto último ocurra, simplemente se vuelve a ejecutar **fillBuffer()**, donde r posee un nuevo valor de A y s también posee un nuevo valor de B , ya que **NO quedó dentro del último Buffer generado**.

Teniendo todo esto presente, las tuplas que hacen match entre R' y S' **siempre serán retornadas** a través de la operación **next()**. Esto significa que las tuplas que **NO hacen match** son las que entran al **primer while de fillBuffer()**. Inferido esto, y **sin necesidad de modificar la estructura del Buffer**, se puede implementar el **Sort-Merge Full Outer Join** mediante los siguientes cambios al procedimiento original:

- Dentro de la operación **fillBuffer()**, cuando se cumpla $r(A) < s(B)$, se debe retornar $(r, null)$ **antes** de llamar a la siguiente tupla de R' .
- Dentro de la operación **fillBuffer()**, cuando se cumpla $r(A) > s(B)$, se debe retornar $(null, s)$ **antes** de llamar a la siguiente tupla de S' .
- Cuando se termine de recorrer la relación R' o S' **completa** (la que tenga el menor valor máximo de A o B), entonces todas las tuplas restantes de la otra relación serán retornadas directamente como $(r, null)$ (si S' es la que se termina de recorrer primero) o $(null, s)$ (si R' es la que se termina de recorrer primero).

Mediante estos cambios, todas las tuplas r que **NO hicieron match con S** , serán retornadas con valores nulos **al avanzar hacia la siguiente tupla de R' en la operación `fillBuffer()` (o bien al terminar de recorrer a S')**, mientras que todas las tuplas s que **NO hicieron match con R** , serán retornadas con valores nulos **al avanzar hacia la siguiente tupla de S' (o bien al terminar de recorrer a R')**, logrando retornar todas las tuplas de R y S incluso si no pertenecen al **Natural Join**, lo que corresponde al **Full Outer Join**.

Al momento de calcular el costo, es posible notar que las únicas operaciones adicionales que se ejecutan en este algoritmo corresponden a accesos a las tuplas actuales en R' y S' . Como dichas tuplas están actualmente en memoria, no se requieren accesos I/O adicionales al disco, por lo que todas estas operaciones no afectan al costo del algoritmo. Finalmente, el costo de **Sort-Merge Full Outer Join** corresponde a:

$$cost(R \bowtie S) = cost(R) + cost(S) + 2 * (pages(R) + pages(S))$$

Donde el último término corresponde al costo de ordenar cada una de las relaciones dentro de la operación **open()**.