Evocaciones Asociativas de Respuesta Múltiple

Año 2026

Introducción

Hasta el momento hemos trabajado suponiendo que se quiere resolver un servicio asociativo de respuesta única o servicio con dependencia funcional y para ello planteamos la existencia de una relación $R \subseteq X \times Y$ (donde X e Y pueden ser dominios compuestos), y sobre esa relación un único servicio de evocación asociativo:

$$\begin{array}{cccccc} R & \subseteq & X & \times & Y & \\ s & : & * & & ? & \text{donde } X \to Y \end{array}$$

Con la existencia de la dependencia funcional $X \to Y$ se busca almacenar nuplas en un espacio de memoria, respondiendo de manera eficiente a evocaciones asociativas aportando un atributo $x \in X$ y recibiendo un $y \in Y$. La forma más general para el encabezado de la rutina de evocación es:

Evocar(***in*** *x,* ***out*** *y,* ***out*** *exito*)

En el mismo vemos los atributos ***x, y*** de la nupla y además consideramos un parámetro de éxito, ya que si bien sabemos que para un ***x*** particular perteneciente a la relación habrá un único ***y*** asociado puede pasar que dicho ***x*** no pertenezca a la relación si la misma no es completa respecto de ***X***. El procesamiento de la evocación en el programa que la invoca generalmente tiene la forma:

```
.
.
.
Evocar(x, y, exito)
if(exito)
        procesar(y)
  else /* tomar medidas ante fracaso */
.
.
.
```

En el caso que la relación sea completa respecto de ***R*** el encabezado de la evocación se simplifica y tiene la forma:

Evocar(***in*** *x,* ***out*** *y,*)

y para procesar el resultado de su invocación:

```
.
.
.
Evocar(x, y)
procesar(y)
.
.
.
```

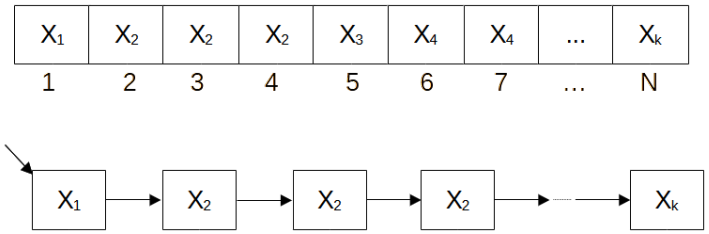
Rutinas para resolver un servicio de respuesta múltiple

Cuando consideramos la no existencia de dependencia funcional entre ***X*** e ***Y*** (se denota con $X \nrightarrow Y$) pero aún buscamos resolver evocaciones asociativas de manera eficiente nos encontramos con situaciones en las que para un ***x***:

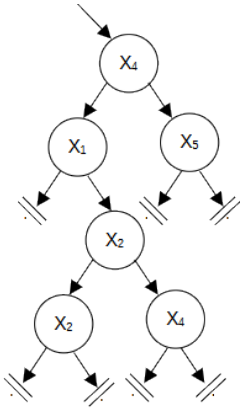
- el ***x*** no exista en la estructura,
- exista un único ***y*** para el ***x*** buscado,
- existan varios ***y***'s para el ***x*** buscado.

Entonces si pensamos en las estructuras de almacenamiento que conocemos la disposición de los elementos repetidos :

- Podrán estar distribuidos en cualquier parte de la lista en listas desordenadas (secuenciales o vinculadas).
- En las listas ordenadas (secuenciales o vinculadas) todos los elementos con igual valor de ***x*** deben estar juntos para mantener el orden.

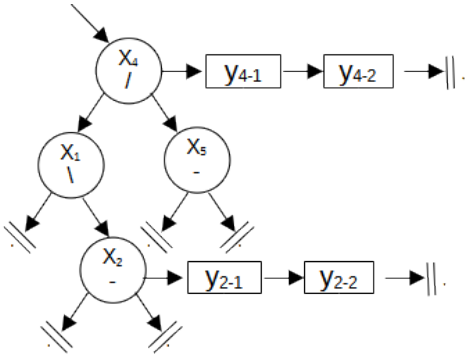


- En un Árbol Binario de Búsqueda los elementos repetidos estarán todos a la izquierda o derecha (recuerde la definición formal para el *ABB*) del primer elemento incorporado.



- En Lista de Dos Niveles las sublistas mantienen orden, por lo que si hubieran ***x*** ´s repetidos estarán todos juntos en al menos una sublista.

- En la Skip List los elementos se encuentran ordenados, por lo que aquellos que coincidan en el valor de ***x*** estarán ubicados consecutivamente en el nivel 0, aunque no será así en niveles superiores (recordar que el nivel se genera aleatoriamente para cada nupla).
- En AVL y Splay Trees la propia definición de estas estructuras no admitía ***x*** ´s repetidos, entonces en los casos en que un ***x*** tenga más de un ***y*** asociado, éstos se ubicarán en una lista asociada al ***x***.



- En los Rebalses Abiertos, independientemente del tratamiento utilizado, los elementos se encuentran en listas desordenadas que inician en la posición retornada por la función de pseudo-azar. Por lo que las nuplas con igual valor de ***x*** estarán distribuidas en cualquier posición de la lista correspondiente.
- En Rebalse Separado las nuplas con igual valor de X (nupla completa) se ubican en cualquier posición de la lista vinculada desordenada determinada por la función de pseudo-azar.
- En el Direccionamiento Directo como los valores de ***x*** no se almacenan, los valores de ***y*** asociados a un ***x*** determinado se agrupan en la lista vinculada asociada a la posición retornada por la función de enumeración.

A partir de la idea descripta, para almacenar las nuplas en las estructuras debemos plantear las rutinas necesarias para resolver el servicio de evocación cuando $X \twoheadrightarrow Y$; entonces, en un primer intento podríamos pensar en agregar más de un parámetro de salida para recuperar los valores de ***Y*** asociados

Evocar(***in*** x, ***out*** y_1, y_2,...,y_n, ***out*** *éxito*)

Aún conociendo el número máximo de parámetros a retornar esa cantidad podría ser muy grande para algunas relaciones. Por ejemplo, se puede pensar en una relación en que dado el código de carrera se guarden los alumnos de la misma, por lo que codificar una rutina con semejante cantidad de parámetros no sería viable y además si en algún momento alguna carrera tuviera más de *n* alumnos ya no sería útil esa configuración.
El problema se origina al tratar de pensar la solución como si aún tuviéramos dependencia funcional, pero como ya no tenemos esa condición y para independizarnos del número de ***y*** ´s asociados vamos a usar una rutina que se invoque repetitivamente, y que en cada invocación devuelva un ***y***.
Esta rutina, a la que llamaremos *Deme_Otro*, debe asegurar que no se repita ni se omita ninguna respuesta. Para poder retornar el próximo ***y***, la rutina *Deme_Otro* necesita saber el ***x*** sobre el que se está realizando la evocación; luego el encabezado sería de la siguiente forma:

Deme_Otro(***in*** *x,* ***out*** *y)*

Pero esto implica que en cada invocación debemos volver a informar el ***x***. Para evitarlo existirá otra rutina, a la que llamaremos *Inicio* que permita pasar el ***x*** una única vez. Además, será necesaria otra rutina que indique si hay o no más respuestas para el ***x*** sobre el que se está realizando la evocación. Resumiendo, son tres las rutinas necesarias para resolver un servicio sin dependencia funcional:

Inicio(***in*** x)
Hay_más (): **boolean**
Deme_Otro(***out*** *y)*

Estas rutinas compartirán, además de la estructura donde está almacenada la relación ***R***, un mismo espacio de datos con las variables necesarias para resolver el servicio. Como mínimo, tendremos dos variables:

- x_{int}: que mantiene el *x* corriente sobre el que se está realizando la evocación.
- i_{int}: es una posición (índice o apuntador) en la estructura donde está almacenada ***R***. Esta variable indica la posición a partir de la cual se debe continuar la búsqueda del próximo ***y***.

La rutina *Inicio* es la encargada de inicializar adecuadamente estas variables en función del ***x*** sobre el que se realiza la evocación. La rutina *Deme_Otro* debe, además de devolver el próximo ***y***, avanzar la variable i_{int} para asegurar que ese mismo ***y*** no sea tomado nuevamente como otra respuesta distinta.
El programa principal o invocante, en el momento de necesitar el servicio, debe invocar una única vez la rutina *Inicio* para informar el ***x*** sobre el que se realiza la evocación; luego, mientras existan más respuestas (esto lo informa la rutina *Hay_más*), se invoca la rutina *Deme_otro* para obtener el próximo ***y***. Entonces, la estructura del programa invocante tendrá la forma:

```
                                .
                                .
                                .
                                Inicio(x)
                                While (Hay_mas()){
                                      Deme_otro(y)
                                      Procesar(y)
}
                                .
                                .
                                .
```

Notar que estamos asumiendo que el programa principal realizará con todos los ***y*** exactamente el mismo procesamiento. Esta suposición es correcta dado que si así no fuera significaría que el servicio está mal planteado, y debería hacer sido más selectivo.
El hecho de que el programa principal procese a todos los ***y*** de la misma manera implica que tampoco importa el orden en que retornemos estos ***y***. Lo único que sí interesa es que se devuelvan todos, sin repetir ni omitir ninguno.

Conversión a un problema de respuesta única

Sea $R \subseteq A \times B$; si esta relación ***R*** no es función entonces existen elementos en el dominio a los que les corresponden más de una imagen; por ejemplo podríamos tener que $(a, b1) \in R$, $(a, b2) \in R$

y $(a, b3) \in R$ pero, toda relación ***R*** que no es función induce una función, que denotaremos con f_R, definida de la siguiente manera:

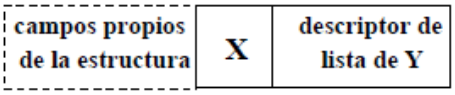
$$f_R : A \to \mathcal{P}(B) \mid f_R(a) = \{b / \exists (a, b) \in R\}$$

En el problema que estamos tratando tenemos $R \subseteq X \times Y con X \nrightarrow Y$. Dado que $X \nrightarrow Y$, en algún momento dado puede suceder que la relación tenga las tuplas (x, y_1) y (x, y_2). Es decir, tenemos una situación isomorfa a la anterior. Usando un razonamiento similar al ya utilizado, podemos definir una nueva relación ***R'*** que estará formada por pares de la forma $(x, f_R(x))$; es decir, la primer componente del par es ***x*** y la segunda es el conjunto de ***y***'s asociados a ese ***x***, y ese conjunto sí es único. Por lo tanto, nuestro problema se traduce en:

$$\begin{array}{cccccc} R & \subseteq & X & \times & \mathcal{P}(Y) & \\ s & : & * & & ? & \text{donde } X \to \mathcal{P}(Y) \end{array}$$

Veamos ahora cómo afecta esto a las estructuras de almacenamiento. Cuando teníamos $R \subseteq X \times Y$ las celdas de nuestras estructuras de almacenamiento debían tener como mínimo dos campos: uno para ***x*** y otro para ***y***; dependiendo de la estructura podían existir campos adicionales para mantener información propia del manejo de la estructura:

Ahora tenemos que $R' \subseteq X \times \mathcal{P}(Y)$; por lo tanto, las celdas de nuestras estructuras de almacenamiento deberán tener un campo para contener ***x*** y otro para contener conjuntos de ***y***'s. La forma más fácil de representar un conjunto es por medio de una lista; entonces las celdas ahora serán de la siguiente forma:



La lista de los ***y***'s podría ser secuencial o vinculada, en general dependiendo de si el problema es estático o dinámico. Además, como no necesitamos buscar un ***y*** en particular sino que necesitamos devolver todos los ***y***'s asociado a un ***x***, no tiene sentido ordenar esta lista de ***y***'s. Usar para los ***y***'s estructuras más complicadas que una lista desordenada puede tener sentido cuando sobre la relación se plantean varios servicios (se verá más adelante).
La interfaz con el programa principal no cambia, dado que seguimos necesitando las tres rutinas para la comunicación con el programa principal. Pero ahora estas rutinas se simplifican:

- El ***x***, en caso de existir, está una única vez en la estructura. Luego, la rutina *Inicio* se puede encargar de localizarlo, porque esta localización se hará una única vez.
- Entre *Hay_más* y *Deme_ otro* deben encargarse de recorrer la lista donde están almacenados los ***y***'s.

- No se necesita más la variable x_{int}. Recordemos que esta variable servía para que las tres rutinas pudieran consultar, en el momento en que lo necesiten, el ***x*** sobre el cual se está realizando la evocación. Si forzamos dependencia, la única rutina que necesita conocer el ***x*** es *Inicio*, porque es la encargada de localizarlo.

Operaciones de Alta y Baja

Toda inserción o eliminación de una nupla en una relación realiza dos pasos perfectamente definidos:

Búsqueda + Modificación estructural

En el caso de la inserción, la búsqueda se realiza para verificar que la nupla no exista, dado que en *R* no pueden existir nuplas repetidas; en el caso de la eliminación la búsqueda sirve para verificar que la nupla que se quiere dar de baja realmente exista.
Habíamos visto que una forma de identificar las nuplas en una relación es por medio de la clave. Esto significa que la evocación asociativa por la clave siempre existe; ya sea porque es un servicio que el usuario requiere o porque es un servicio que internamente se necesita para manejar correctamente los procesos de inserción y eliminación.
En nuestro caso tenemos:

$$\begin{array}{cccccc} R & \subseteq & X & \times & Y & \\ s & : & * & & ? & \text{donde } X \twoheadrightarrow Y \end{array}$$

Si $Y \rightarrow X$, entonces la clave de *R* es *Y*; caso contrario, la clave es *XY* . En cualquiera de los dos casos, necesitamos sobre *R* un servicio asociativo adicional, lo que implica resolver varios servicios sobre una relación, temática que aún no hemos abordado. En el caso particular de que la clave sea *XY*, podemos resolverlo fácilmente con las herramientas vistas hasta ahora. Expresemos en símbolos el problema que se nos plantea:

$$\begin{array}{cccccc} R & \subseteq & X & \times & Y & \\ s_1 & : & * & & ? & \text{donde } X \twoheadrightarrow Y \\ s_2 & : & * & & * & \end{array}$$

Para s_1 hemos organizado una estructura que permite evocar eficientemente por *X*. Dado que en s_2 uno de los dominios aportados es *X*, podemos resolver esta evocación usando la misma estructura armada para s_1 realizando los siguientes pasos:

1. Como primer paso localizamos ***x***.

2. Si esta localización no es exitosa, entonces podemos asegurar que la nupla no existe.

3. Caso contrario procedemos a localizar ***y*** en la lista asociada al ***x*** encontrado en el paso 1.

4. Si la localización del paso 3 no es exitosa, entonces la nupla no pertenece a la relación. Caso contrario la nupla existe y en consecuencia la localización por ***xy*** es exitosa.

Notar que para s_2 sólo necesitamos la rutina *Localizar* pero no la rutina *Evocar*. Resumiendo, resolvemos la localización por *XY* como una localización por *X* seguida de una localización por Y . Es por esta razón que puede tener sentido armar para la lista de *Y* una estructura con mejor comportamiento que una lista desordenada.
Habiendo realizado la búsqueda, si la inserción o eliminación es válida, se procede a realizar la modificación estructural tal como lo veíamos en la materia anterior. En el caso de haber forzado dependencia hay que tener en cuenta algunas consideraciones adicionales:

- Si se está insertando la nupla (x, y) se pueden presentar dos casos:
  1. que el ***x*** ya exista: en este caso sólo debemos insertar el ***y*** en la lista correspondiente.
  2. que el ***x*** no exista: en este caso debemos insertar el ***x*** en la estructura armada para los ***x*** y dar de alta el ***y*** como primer elemento de la lista asociada a ese ***x***
- Si se está eliminando la nupla (x, y) debemos comenzar dando de baja ***y***, y luego, sólo si la lista de ***y*** queda vacía, damos de baja ***x*** en la estructura correspondiente; dado que no tiene sentido mantener un descriptor vacío, salvo que la estructura armada para los ***x****'s* sea un direccionamiento directo.

Reconocimientos

El presente apunte se realizó tomando como base notas de las clases del **Ing. Hugo Ryckeboer** en la Universidad Nacional de San Luis y completando con material de la materia **Organización de Archivos y Bases de Datos I** de la Universidad Nacional de San Luis.