Recuperación de datos

1. Introducción

Uno de los principales objetivos en el uso de las computadoras es la posibilidad de poder procesar grandes volúmenes de datos en forma veloz, de recuperar los datos de un determinado registro conociendo únicamente una clave o de insertar un registro validando que no exista previamente. Los distintos métodos de búsqueda, toman como argumento primario a la clave para recuperar la dirección física donde se encuentran los datos del registro. Estas técnicas tienen como principal objetivo obtener la dirección del registro identificado por la clave en el menor tiempo posible. Un sistema bien presentado y con múltiples opciones de consultas, será ineficiente si el sistema tarda más de lo que el usuario considera lógico esperar detrás de una pantalla y en muchas ocasiones la demora se produce por algoritmos ineficientes de recuperación de datos.

Claves.

La clave, puede estar formada por un único campo o varios. Podríamos, en principio definirla como un conjunto de datos asociado a cada registro de forma tal de poder identificarlo unívocamente. Aquí no vamos a tratar de dar una definición muy estricta, como si lo será, por ejemplo, la definición de una clave primaria cuando se estudie teoría de bases de datos relacionales.

En las organizaciones de datos donde la clave está contenida dentro del registro recibe el nombre de clave **interna**. Otras organizaciones manejan archivos externos al archivo de datos, donde guardan la información referente a las claves que se denominan **externas**.

Algoritmos de recuperación de datos.

Las distintas técnicas utilizadas para recuperar datos, o los algoritmos de búsqueda, pueden devolver un registro completo o un puntero a ese registro. Estos algoritmos deben contemplar la posibilidad de insertar una clave única, con lo cual deberá verificar que la clave no exista previamente **(DUPKEY)** o de recuperar una clave validando que la misma exista **(INVALID KEY)**. Hay otros métodos que trabajan con claves duplicadas, otros que recuperan registros por su posición relativa en el archivo y otros que directamente no utilizan claves.

Búsqueda secuencial.

Es la forma más sencilla de buscar un dato y consiste en efectuar una lectura secuencial del archivo hasta detectar que el valor que tiene(n) cierto(s) campo(s) coincide(n) con un argumento de búsqueda. En caso de no hallarse el argumento de búsqueda, se detectará el invalid key recién después de haber leído todo el archivo. El orden de complejidad de este algoritmo es lineal pues en un archivo de **n** registros la cantidad de comparaciones esperadas para hallar una clave (en promedio) es **n/2**, cuando la clave buscada existe en el archivo.

Búsqueda secuencial Indexada.

Esta técnica mejora notablemente el tiempo de recuperación pero al hacer uso de una tabla externa ordenada es necesario disponer de más espacio para la misma. Cada elemento de la tabla contiene un valor de clave y la dirección donde se encuentran los datos de esa clave. Los elementos de la tabla deben seguir el mismo orden que el definido para las tablas en el archivo y la cantidad de elementos que contiene la tabla estará definido por la cantidad máxima de registros del archivo.

El procedimiento a considerar entonces es examinar cada elemento de la tabla hasta hallar la clave buscada o el primer valor de clave superior, con lo cuál la clave es la inmediata anterior. Si la clave buscada se encuentra en la tabla de índices, el puntero asociado tiene la dirección de los datos que se quieren recuperar. Si la clave buscada es menor que la clave(i+1) de la tabla y mayor a la clave(i), con la dirección del puntero de la clave(i) se accede al archivo de claves y se recorre secuencialmente hasta hallar la clave buscada o la clave(i+1) en cuyo caso se detectaría la ausencia de la clave (invalid key). La principal ventaja de este método reside en que si el tamaño de la tabla se ajusta a los requerimientos de tiempos de recuperación y espacio ocupado, la búsqueda se optimiza notablemente. Suponiendo que el factor de la tabla es 1/16, equivale a decir que por cada elemento de la tabla se consideran 16 claves. Si el volúmen de datos es muy grande, probablemente sea conveniente crear una nueva tabla, llamada **secundaria** con el mismo criterio de funcionamiento que la tabla primaria pero que apunte a la tabla primaria en lugar de apuntar a los datos. **Ejemplo:**

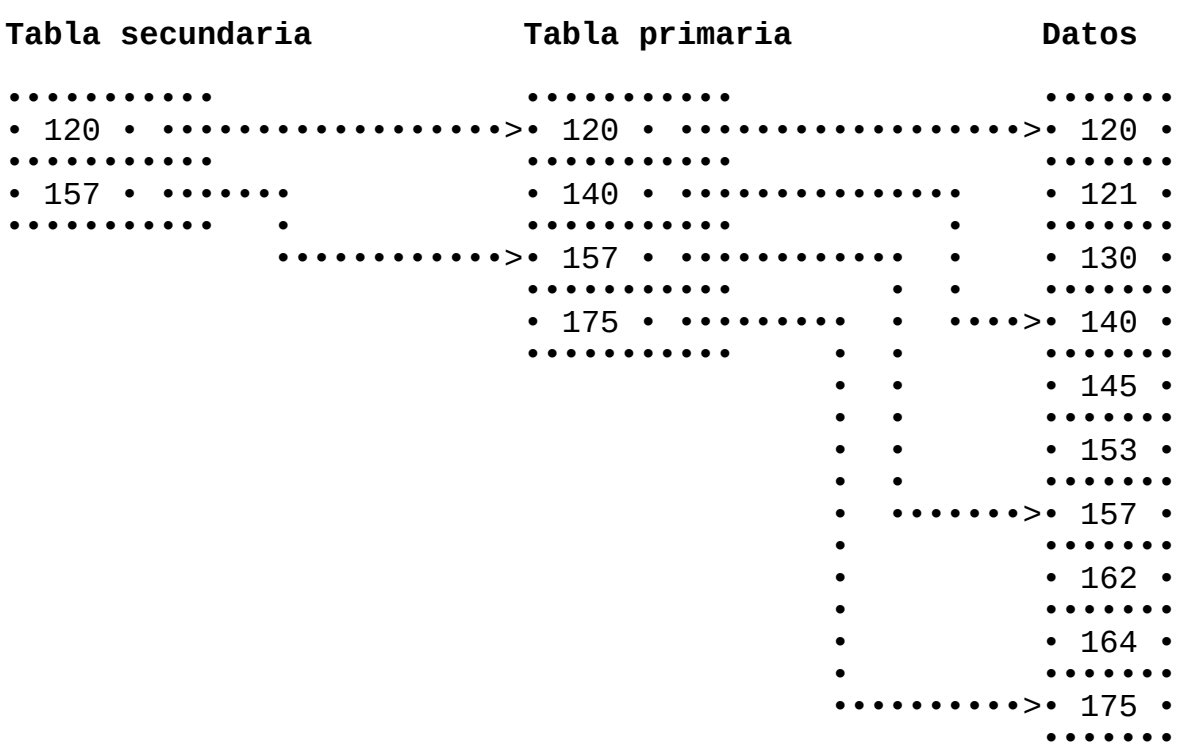

Figura 1

2. Hashing o randomización de claves.

En las técnicas de búsqueda basadas en estructuras arbóreas, siempre es necesario comparar una cierta cantidad de claves hasta encontrar la buscada. Lo óptimo, sería minimizar la cantidad de comparaciones, evitando aquellas que se consideran innecesarias o redundantes con lo cuál se disminuirían los tiempos de recuperación. De esta manera es posible pensar en establecer una relación directa entre la dirección de los datos y el valor de la clave en lugar de obtener la dirección en función de la posición relativa de la clave respecto de las restantes.

Supongamos que el valor de la clave **key** es un número entero de tres dígitos. Puede definirse entonces una tabla, **table()**, de 999 elementos, donde cada uno guardará la dirección que le corresponda al valor absoluto de la clave tomado como subíndice.

Es decir, dada la clave **key = n**, en **table(n)** estará la dirección que apunta al registro con clave = n.

Si bien este algoritmo es óptimo en términos de tiempos de recuperación, por no realizar ninguna comparación, es impracticable pues es imposible subordinar el valor de una clave a la cantidad máxima de elementos de una tabla. Por ejemplo es muy común que el plan de cuentas contables de una compañía cuente con nueve o más dígitos y sólo utilice 200 ó 300 cuentas, con lo cuál debería definirse una tabla de 999.999.999 elementos de los cuales 200 ó 300 tendrían datos y el resto estaría vacío. Esta solución es inadmisible en términos de eficiencia y de espacio inutilizado.

Lo ideal sería hallar algún método que permita hallar la dirección de un registro en función de su clave, con la menor cantidad de comparaciones y ocupando el menor espacio posible. En el problema anterior, una tabla de 999 elementos soportaría holgadamente el plan de cuentas sin desperdiciar mucho espacio.

El problema se reduce entonces a encontrar un método que devuelva un subíndice entre 1 y 999 dado un número entre 1 y 999.999.999. Este método se denomina función de hashing, la cual identificaremos con **hash()**, y puede definirse entonces como **la función que aplicada a una clave devuelve el subíndice de la tabla** (donde se encuentra la dirección con los datos del registro buscado). En el ejemplo anterior, la función **módulo 1000** puede encontrar, dada una clave **key = x**, donde **x** es un número entre 1 y 999.999.999, un valor de subíndice entre 1 y 999.

Aún considerando la mejora producida, se siguen desperdiciando 700 elementos de la tabla. Para minimizar este desperdicio, lo óptimo sería conocer previamente la cantidad máxima de registros que puede tener el archivo, **maxrec**, y dimensionar la tabla con un número máximo de elementos, **maxtab**, igual al primer número primo que sea mayor en valor absoluto a **maxrec**. **Ejemplo 4 :** si **maxrec = 300** => **maxtab = 307**

El principal problema inherente a este método se produce cuando el valor que devuelve la función de hashing **hash()** es el mismo para dos o más claves iguales, lo que se define como colisión. Más precisamente:

Colisión

Sean k1 y k2 claves pertenecientes a los registros r1 y r2 respectivamente, con k1 ≠ k2. Y sea h(k) la función de hash.

Si h(k1)=h(k2) entonces decimos que se produce una colisión.

Ejemplo:

Supongamos que $\mathbf{k_1}$ y $\mathbf{k_2}$ son claves de dos cuentas del plan de cuentas.

$\mathbf{k_1 = 112245015}$
$\mathbf{k_2 = 118225989}$

Si la función de hash elegida es $\mathbf{hash(k_i) = k_i \bmod_{307}}$ =>

hash(112245015) = **289**
hash(118225989) = **289**

En este ejemplo puede apreciarse que la función de hash devuelve, para dos claves distintas, el mismo valor de subíndice.

La existencia de lo que llamamos colisiones es **crucial** debido a que, dado que la cantidad de claves posibles siempre es mayor que la cantidad de posiciones en la tabla, entonces siempre se producirán colisiones. ¿Qué queremos decir por "siempre"?. Si bien pueden plantearse métodos que intenten reducir la probabilidad de colisión, la misma nunca llegará a ser cero. Lo que es un concepto en sí matemático, probabilístico ($\forall$ hash(key): P(colisión)>0), tiene una implicancia fundamental desde el punto de vista computacional, algorítmico: que inevitablemente se producirá una colisión al cabo de un tiempo finito (de allí que digamos que siempre habrá colisiones). En consecuencia para hallar la forma mas adecuada de aplicar hashing a un problema de recuperación de datos nos debemos plantear:

a) ¿Cuál es la función de hash que dada la forma de mi clave me minimiza la probabilidad de que se produzcan colisiones?. Esta claro que, dicha función será la que me garantice una distribución uniforme sobre todo el conjunto de posiciones de la tabla, es decir, la que implique que inicialmente, cada posición de la tabla tiene la misma probabilidad de ser ocupada.

b) ¿Cuál es la mejor forma de proceder ante una colisión?. A lo largo del desarrollo de este tema estudiaremos las distintas soluciones mas conocidas.

Generalizando podríamos decir que: *una función de hash es eficiente cuando minimiza el número de colisiones y ocupa los elementos de la tabla de manera uniforme*. Si bien cuanto mayor sea el número de elementos de la tabla, menor será la probabilidad de colisión, el dimensionamiento desmesurado de la tabla ocasionaría un desperdicio de espacio no deseado y otra vez se deberá analizar cada situación en particular, en términos de tiempo de procesamiento y espacio utilizado.

2.1. Funciones de Hash.

Del análisis anterior se desprende que el principal objetivo de una función de hash es minimizar la probabilidad de colisión sin ocupar memoria innecesariamente. Existen distintos métodos, algunos de los cuáles se presentan a continuación a modo de ejemplo, pero en la práctica nunca se utiliza uno en particular sino una combinación de varios de acuerdo a la aplicación que se esté analizando.

Método de la división o del módulo

Consiste en obtener un valor entre 1 y maxtab resultante del resto de la división del valor numérico de la clave y maxtab, es decir que se utiliza la función **módulo** como función de hash. **Ejemplo 6 :**

hash(key) = key mod$_{947}$ => **hash(2866) = 25**

Método del cuadrado medio.

Consiste en multiplicar el valor numérico de la clave por sí mismo y obtener un valor entre 1 y maxtab de los dígitos medios del valor obtenido. **Ejemplo 7 :** si maxtab = 947 y se quiere hallar el valor de la clave 2866

hash(key) = números medios de (key^2) =>
hash(2866) = **8213956** => **hash(2866) = 139**
Si maxtab = 1013 podría tomarse indistintamente
hash(2866) = 2139 o **hash(2866) = 1395**

Método de dobles.

Consiste en dividir la cantidad de dígitos del valor numérico de la clave en dos o más partes iguales y realizar la operación **or exclusivo** con el valor binario de cada una de esas partes. **Ejemplo 8 :** si maxtab = 947 y se quiere hallar el valor de la clave 2866

Se divide la clave en dos partes: **a_1 = 28** y **a_2 = 66**.

El valor binario de **a_1** es = 11100
El valor binario de **a_2** es = 1000010

La operación or exclusivo aplicada sobre los valores binarios de **a_1** y **a_2** dará un valor de subíndice en binario = 1011110 que pasado al sistema decimal es igual a :

hash(2866) = 94

2.2. Tratamiento de las colisiones.

Habíamos visto que si bien una función de hash puede ser mejor que otra, no se pueden evitar las colisiones y por lo tanto, deben buscarse soluciones que minimicen tanto el número de accesos que se deben realizar para obtener la dirección de los datos como el espacio ocupado por las estructuras de datos utilizadas. En general podemos agrupar a todos los métodos para el tratamiento de colisiones en dos: rehashing (direccionamiento abierto) o chaining(encadenamiento).

Rehashing o direccionamiento abierto.

Se define rehashing como la utilización de una segunda función de hash hasta que se encuentre una posición libre.

La solución más trivial o mas simple es la llamada **solución lineal**; si la posición de la tabla de direcciones, **table(dir)**, en el valor de subíndice

obtenido al aplicar la función de hash a una clave, **dir**, está ocupado, se ocupa la siguiente posición libre de la tabla. Es decir:

```
dir = hash(key)
do while table(dir) .not. = vacío
   dir := dir + 1
   if (dir = maxtab )
      dir = 1
   endif
enddo
```

Dicho de otra forma, la función de rehashing dependería exclusivamente del subíndice i hallado con la función de hash original y sería:

```
rhash(i) = (i+1) % maxtab
```

En el código anterior, luego de aplicar la función de hashing, la repetitiva busca el primer elemento libre en la tabla **table()** a partir del subíndice **dir** en adelante, incrementando en uno el subíndice cada vez que detecte un elemento de la tabla ocupado. Se asume que esta rutina controla el overflow de la tabla. En el ejemplo que se dio al definir colisión, para distintos valores de clave, el subíndice devuelto por hash() era **289**. Supongamos que la tabla tiene ocupados por otras claves los elementos 290 y 291. La clave 118225989, ingresada con posterioridad a la clave 112245015, ocupará el elemento de tabla **292**.

Subíndice	Clave	Dirección
288		
289	112245015	xxxxxx
290	109936990	xxxxxx
291	118256691	xxxxxx
292	118225989	xxxxxx
293		
294		

Figura 2

La próxima clave cuyo valor de función de hashing sea igual a 289, 290, 291 o 292 ocupará el elemento de tabla **293**.

Resumiendo, el proceso consiste entonces en dada una clave **key**, se le aplica la función de hashing **hash()** y se obtiene un valor de subíndice **dir**, si el elemento de la tabla **table(dir)** está ocupado por otra clave, se aplicará una función de reasignación **rhash()** al valor **dir** para obtener un nuevo valor de subíndice. Este proceso es recursivo hasta hallar un elemento libre en la tabla.

Debe prestarse especial atención en el diseño del proceso de reasignación pues puede darse el caso de que un algoritmo busque eternamente un elemento disponible en la tabla o que informe que no hay más espacio disponible cuando en realidad sí hay elementos vacíos. Un ejemplo claro de una función de reasignación que pueda llegar a conclusiones equivocadas es la función **rhash(dir) = dir + 2**. En este caso, llevado al extremo, puede darse que todos los elementos de la tabla con subíndice

par estén ocupados, que los impares estén vacíos y la función indique que no existe más espacio disponible.

Una función de reasignación será óptima cuando la recursividad de la misma cubra la totalidad de los elementos de la tabla. Un ejemplo de una función de este tipo es la generalización misma de la solución lineal: **rhash(dir) = (dir + c) mod maxtab** donde **c** y **maxtab** son números primos.

Otro problema a considerar cuando se diseña una función de reasignación es el que se presenta cuando un elemento tiene una probabilidad de ser ocupado mucho mayor que otro. En el ejemplo de la figura 2, el elemento **293** tiene 4 veces más probabilidad de ser ocupada que el elemento **294** pues este elemento sólo podrá ser ocupado por la primer clave cuya función de hash devuelva el valor **294** mientras que el elemento **293** podrá ser ocupado por claves cuya función de hash devuelva el valor **289** ó **290** ó **291** ó **292** respectivamente. Esta particularidad del direccionamiento abierto se conoce con el nombre de **clustering** y una forma de evitarlo es a traves del **doble hashing**. Este método utiliza en la función de reasignación otra función que toma como argumento el valor de la clave. Es decir : **hash(key) = dir**. Si **table(dir)** es distinto de vacío, es decir está ocupada por otra clave, **rhash(dir) = (dir + f(key) mod$_{maxtab}$** donde **f(key)** es una función definida por el programador que toma como argumento el valor numérico de la clave.

Siguiendo con el ejemplo anterior, al ingresar la clave **key=118225989** la función de hash devuelve un valor de subíndice hash(key) = 289. Este elemento de la tabla fue ocupado previamente por la clave 112245015. Entonces se define **f(key) = 10 * key** es decir :

rhash(289) = (289 + (10 * 118225989)) mod$_{307}$ = **109**

La función de reasignación dependerá entonces del subíndice resultante de la función de hashing y del valor de la clave que se quiera ingresar.

Existe otra forma de resolver el problema de colisiones y es hacer depender la función de reasignación del número de veces que es aplicada. De esta manera, **rhash(dir, arg$_1$)** tendrá dos argumentos. Ejemplo:

rhash(dir, i) = (i+1) * dir mod $_{maxtab}$

donde i es una variable inicializada en 1

El principal problema del método de direccionamiento abierto, es que se basa en el uso de una estructura estática, es decir una tabla, la cuál debe ser dimensionada previamente y que en ocasiones puede resultar chica, en cuyo caso se deberá crear una tabla más grande y reorganizar todos los punteros en función del nuevo valor que tome **maxtab**. Otro problema es la dificultad que se presenta al dar de baja un elemento de la tabla. Supongamos que en el ejemplo del plan de cuentas se da de baja la clave **112245015**, con lo cuál el elemento de tabla quedaría vacío. Al querer recuperar la clave **118225989**, el valor de subíndice que devuelve la función de hash es la que se acaba de dar de baja, con lo cuál no puede determinarse si la clave 118225989 no existe o está incluída en alguna de las direcciones obtenidas por las sucesivas funciones de reasignación. Este tipo de inconvenientes podría solucionarse con el uso de marcas de estado pero el proceso de recuperación sería cada vez más complejo y se perdería de vista los objetivos principales que se

plantearon en un comienzo respecto de disminuir la cantidad de accesos y disminuir el espacio utilizado por las estructuras de datos auxiliares.

Chaining o encadenamiento.

Genéricamente, el concepto de chaining consiste en enlazar entre sí los distintos pares de clave/puntero que colisionen entre sí, de forma tal de agruparlos. El encadenamiento puede darse en forma interna dentro de la misma tabla de hash, o también utilizando algún tipo de estructura auxiliar, lo que se conoce como “separate chaining”, y en particular estudiamos como ejemplo la siguiente solución.

Hashing into buckets

Este proceso, basado en el concepto de las estructuras dinámicas, permite resolver el problema de las colisiones a través de listas linkeadas que agrupan los registros cuyas claves randomizadas generan la misma dirección. Básicamente el proceso tiene como estructuras de datos una tabla principal, conocida como **área base**, y una lista linkeada, **área de overflow**, cuyo registro tiene tres campos. En uno se guarda la clave, en el otro se guarda la dirección en donde residen los datos y en el tercer campo se guarda la dirección del próximo registro con igual valor inicial de subíndice **dir**. Cuando una función de hashing aplicada a una clave devuelve un elemento de tabla que fue ocupado previamente por otra clave, el proceso pide entonces una dirección de memoria para guardar los datos de la segunda clave y establece la relación mediante un link entre la primer clave, cuyos datos se encuentran en la tabla en el área base, con la segunda clave, situada en la lista linkeada en el área de overflow. Si se quisiera dar de baja el nodo, simplemente habría que dar una baja en la lista. La principal desventaja de este procedimiento es que se utiliza espacio para los punteros, pero comparado con el método de direccionamiento abierto, es posible construir una tabla principal más pequeña con el consecuente ahorro de espacio y sin el problema que se presenta al tener todas las entradas de la tabla ocupadas. En este caso, deberá medirse la eficiencia del procedimiento a través de la cantidad de accesos que deberán realizarse, es decir la cantidad de nodos a recorrer en la lista, para hallar una clave dada. Si el número de nodos es muy grande, se perderá mucho tiempo en recuperar un dato, con lo cuál habría que rediseñar la función de hashing inicial o el tamaño de la tabla principal.

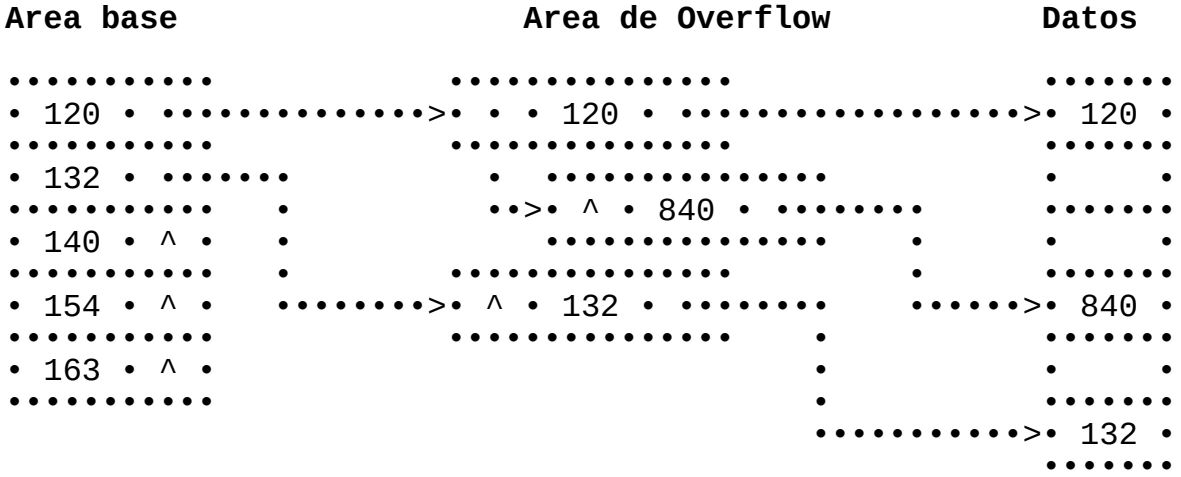

Figura 3