White paper – Hashing

Autor: Ramix (Ramiro A. Gómez)
Sitio web: www.peiper.com.ar
Fecha: 6 FEB 2008

Indice

Introducción

En esta edición de Peiper decidimos armar un white paper con una de la técnicas más interesantes que tiene la informática, no decimos la más interesante, pero sí una de ellas. Esto es así porque el hashing tiene varios usos, que van desde búsqueda rápida de datos hasta encriptación, pasando por códigos de verificación.

¿Qué es el hashing?

El hashing es una técnica que consta de datos de entrada, una función hash y una salida. Esta función calcula un código específico para un dato de entrada (como puede ser un nombre, por ejemplo). El valor calculado puede parecer aleatorio, pero no lo es, ya que las operaciones para computar la salida son siempre las mismas. La función hash asocia siempre la misma salida para una entrada determinada.
Por ejemplo:

hash(“María”) = 1082358727484

luego, a los 3 días calculamos hash(María) nuevamente

hash(“María”) = 1082358727484

y el valor computado es el mismo.

La salida de la función hash es siempre un número, pero se puede convertir a caracteres u otro tipo de dato. Además, este número depende exclusivamente de la entrada porque las operaciones se realizan sobre estos.
Pero la pregunta que nos hacemos es ¿para qué quisieramos asignarle un código a un dato de entrada (como ser un nombre)? La respuesta es que este número lo podemos usar para almacenar ese dato en la posición indicada.
Ejemplo: hash(“Pepe”) = 130
Entonces vamos a almacenar en la posición 130 la cadena “Pepe”, además de sus datos personales, por ej. Cuando querramos buscar “Pepe” no hace falta recorrer todo el arreglo. Calculamos la función hash a este nombre y accedemos a la posición del arreglo que nos indica la salida de la función.

Si no existiría el hashing, por ejemplo, para buscar un dato en un arreglo tendríamos que buscar en cada posición, una por una. Y esto se vuelve un problema para conjuntos grandes de datos.

Imaginen tener una lista de clientes de una tarjeta de crédito a nivel mundial, con un tamaño de 500 millones. Solo para encontrar uno de ellos habría que recorrer una media de 250 millones de posiciones!! Agreguen a esto si tenemos 1.000.000 de ellos queriendo acceder al mismo tiempo. Esta sí es una situación crítica!

Funciones hash

Aunque la filosofía de nuestro sitio es explicar cosas complejas de la manera más sencilla posible (algo que denominámos "algoritmia a la criolla"... y no es un plato de comida,je), vamos a introducir un poco de formalidad, de manera de verlo resumido en símbolos.

Una función hash es una función

$$h(x): N \to N / h(x) = y$$

(generalmente la entrada son números naturales y la salida también).
Muchas veces es imposible encontrar una función inversa

$$hi(y) = x$$

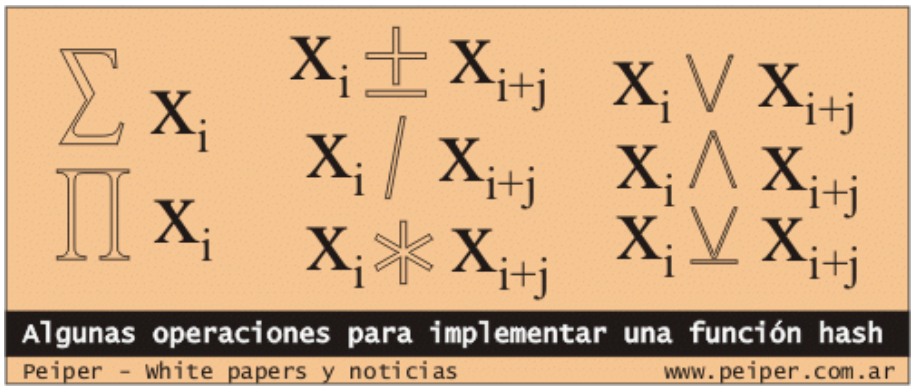
ya que dos entradas x totalemente distintas pueden llegar a producir una misma salida y. Si es posible encontrarla, es un proceso muy complejo que consume mucho tiempo si se realiza por fuerza bruta.

Una función hash debe ser:

- *rápida de calcular*: porque es posible que se necesite calcular muchas en poco tiempo
- *distribuir los elementos de manera uniforme en todo el rango de la salida*: para evitar los aglomeramientos que degradarían el rendimiento.
- *Siempre devolver el mismo código hash para una misma entrada*: por ej. tener cuidado de usar valores aleatorios calculados dentro de la función.
- *Provocar pocas colisiones* (en lo posible ninguna): cuando una función hash no tiene colisiones se dice que es "perfecta".

La técnica de hashing aplicada sobre tablas permite operaciones de búsqueda, inserción y borrado muy eficientes, de orden O(k), k:constante. En la práctica la eficiencia media de estas operaciones es O(1,5) u O(2). Para comparar, en búsqueda lineal es O(n/2), n:cantidad de elementos.
Ya podemos ver la gran ventaja de las tablas hash: la búsqueda es "casi" independiente de la cantidad de elementos.

Una función hash traduce los datos de entrada a números. Por ej., a cada carácter le asigna un número.



Una vez que tiene los números de entrada les aplica sumas, restas, multiplicaciones, divisiones, operaciones lógicas AND, OR, NOT y XOR (entre otras). También rota

los bits de un número a izquierda o derecha. Es decir, puede aplicar todas las operaciones que se realizan sobre números.

Una buena función de éstas, produce un número totalmente diferente aunque los cambios en los datos de entrada sean mínimos. Se relaciona de buena manera con un comportamiento caótico (donde influyen sobremanera las condiciones iniciales, la entrada).
Ej:

hash("Peiper") = 5875775475550

Ahora reemplazamos la P por D,

hash("Deiper") = 8416409012872

Colisiones

Si tenemos W cantidad de entradas posibles, y la cantidad de bits de la salida es Z; si $2^z < W$ quiere decir que tenemos más entradas que posibles salidas. Por ejemplo: tenemos 1000 nombres y nuestra salida es de 9 bits. Con 9 bits podemos formar 2^9 combinaciones distintas de los bits, o sea 512.

000000000, 000000001,
000000010, 000000011, etc...

Tenemos que mapear 1000 claves a 512 posiciones, y esto quiere decir que habrá muchas posiciones que tengan más de una clave. Como media tendrémos casi 2 claves por posición (1000/512).

Es imposible que si tenemos más elementos que cantidad de claves no se produzcan colisiones. Tal vez por eso siempre descubren colisiones en las funciones hash criptográficas MD4, MD5, etc. Y no me cabe duda que también lo harán con SHA y las futuras. Es que la cantidad de posibles combinaciones que se pueden hacer con la longitud de un archivo son infinitamente más que con 160 bits, por ejemplo. Por más que hagan un función hash de 16536 bits, si hasheamos archivos de más de 2000 bytes (16536/8) tendremos alguna colisión. Es una ley, y al comprenderla nos resultará trivial.

En general, hay 2 maneras de resolver colisiones hash cuando se aplican a tablas, hash abierto y hash cerrado.

El hash abierto consiste en asignarle una lista a una posición con colisión, de modo que al tener varios datos con el mismo código hash, se la debe recorrer elemento por elemento hasta encontrar el deseado. También hay opciones para usar árboles en vez de listas enlazadas y hasta nuevas tablas hash.
La opción de que cada elemento del arreglo principal referencia a una nueva tabla hash es ideal para grandes volúmenes de datos, donde almacenar todo en un mismo arreglo no sería conveniente debido al gran tamaño requerido. Esta opción se conoce como árboles hash. Y es fácil recordar que cada nodo el árbol es una tabla hash.

En el hash cerrado al haber una colisión se busca una nueva posición en la tabla. La elección se puede hacer con una nueva función hash, o eligiendo la próxima ubicación vacía de la tabla. Se hará entonces pos = h(x) + i, el valor i comienza en 1, y se incrementa en caso de que la posición esté ocupada.
Algunas de la opciones para incrementar i son:

- pos = h(x) + i, i = 1,2,3,4... : exploración lineal
- pos = h(x) + i^2, i = 1,2,3,4... : exploración cuadrática
- pos = h(x) + i^2+i, i = 1,2,3,4...
- pos = h(x) + polinomio(i), i = 1,2,3,4... : exploración polinomial

Los investigadores han observado un problema en la primera opción, llamado aglomeramiento. Al haber varias colisiones y resolverlas buscando la próxima ubicación libre, se forman grupos de datos densos, que no tienen espacios libres entre sus elementos. Este es un problema importante porque disminuye la performance de la búsqueda, inserción y borrado en la tabla, le quita beneficios a la técnica de hashing.

Usos del hashing

Tiene varios usos en la informática: en tablas hash, en criptografía, en verificación de integridad de datos.

En lo que respecta a las tablas hash, es una opción ideal para implementar bases de datos. Lo que hace muy atractivo su uso es la simplicidad que tiene y los resultados que logra. Otras alternativas al uso de estas tablas son los árboles, pero esta estructura de datos es de naturaleza más compleja, requiere algoritmos recursivos para recorrerlos, para balancear su contenido, etc.

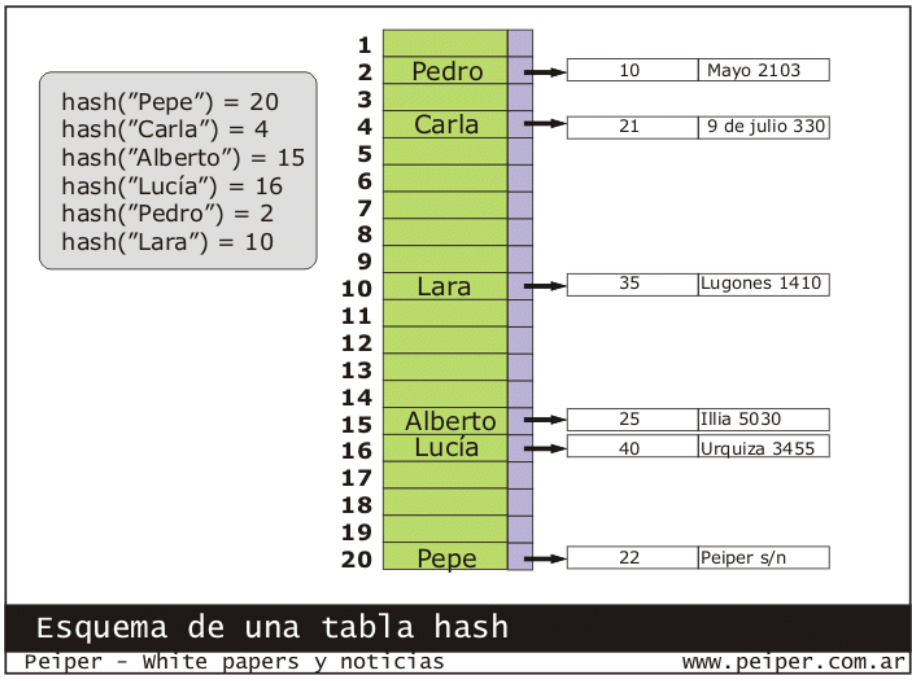
En criptografía el hashing se utiliza para firmar los mensajes y archivos. Firmar significa calcular un valor hash en base a los bytes de los datos de entrada. Una vez firmado el mensaje o archivo, si alguien modifica su contenido será delatado porque el valor hash almacenado en éste no coincidirá con el que se calcule al momento de desencriptarlo. Entonces, aunque el mensaje sí pudo ser alterado, el receptor lo sabe.
Hay muchos algoritmos para calcular una función hash,. Algunos de ellos son MD4, MD5, SHA, Tiger, etc.
Hace un tiempo se detectaron colisiones en MD5, que era uno de los más usados. Esto obligó a transladarse a un algoritmo más seguro, el SHA. No se han detectado colisiones aún, pero ya hay proyectos de computación distribuida que usan un enorme poder de cálculo para encontrarlas. Es sólo cuestión de tiempo para que suceda, debido a las razones que planteamos antes en este mismo texto.

La misma idea se usa para verificación de integridad de datos. Un archivo puede mantener un valor hash que es computado cuando se crea, y si por algún error en el hardware o software se modifica nos darémos cuenta porque los hashes no coincidirán.
Los protocolos de red usan mucho esta técnica para detectar problemas en la transmisión. Se asigna el valor hash a un dato antes de enviarlo por la red, y cuando el receptor lo recibe lo calcula nuevamente. Los 2 valores tienen que coincidir, de lo contrario hubo un error. En general se divide un dato en pequeños fragmentos, y se les aplica esta técnica a cada uno de ellos. Así, si ocurre un error no habrá que reenviar todo el archivo desde cero, sólo desde el fragmento con error.
Como se ve, una técnica por demás interesante.
Una implementación de esto no demasiado poderosa es el checksum o suma de verificación, que consiste en sumar todos los valores numéricos de entrada y proyectarlo en un rango específico. Para hacer esto se usa la función módulo (resto de una división).
Ej:

mensaje: 100 - 70 - 200 - 130
checksum(mensaje) = (100+70+200+130) mod 256

Tablas hash

Una tabla hash es un arreglo o vector en el que se almacenan los datos en la posición indicada por su valor hash.
Su ventaja es que permite accesos muy rápidos a los datos, con una eficiencia aproximada O(k), donde k es constante.
Su desventaja principal es que desperdicia memoria. Esto ocurre porque los datos suelen estar dispersados uniformemente por toda la tabla, y entre ellos quedan espacios vacíos. Se pueden usar, claro está, pero cargar una tabla a más del 70% de su tamaño no es recomendable porque la búsquedas comienzan a tender a cierta linealidad, como si de un lista encadenada se tratara.
Otra desventaja muy importante es que una tabla hash estática tiene dificultades para "crecer" o redimensionarse. La opción más simple cuando una tabla está saturada es crear una nueva más grande y copiar los datos uno por uno. Las técnicas para tablas hash redimensionables se conoce como hashing dinámico.



Esquema de una tabla hash
Peiper - White papers y noticias www.peiper.com.ar

Otro punto importante es el tamaño de las tablas hash. Se pudo determinar que con tamaños iguales a números primos las probabilidades de colisión se reducen.

En una tabla hash las celdas pueden contener disntintos datos, entre ellos punteros a un "cajón" de datos, punteros a una lista enlazada, punteros a otra tabla hash, punteros a árboles o grafos, y los datos en sí (sin usar punteros). Esta última opción no es muy conveniente porque el esquema pierde flexibilidad. Al usar sólo punteros, la tabla hash funciona como índice, y luego podemos agregar datos adicionales dinámicamente. El sistema operativo se encarga de ubicar los distintos bloques en memoria.

Hashing dinámico

El hashing dinámico surgió para solucionar el problema del crecimiento o encogimiento de las tablas hash estáticas. Estas tienen dificultades porque las funciones hash por lo general son de la forma:

$$h(x) = f(x) \bmod k$$

(k es el tamaño)

Al ser k una constante, si redimensionamos la tabla

tendremos que cambiar la función hash porque de lo contrario nos seguirá mapeando los valores en el primer intervalo de celdas.

Una solución es hacer a k una variable. Esta solución genera más colisiones en el intervalo que ya está más ocupado.
Para solucionar este problema se puede recurrir a una función de la forma

$$h(x) = f(x) \bmod x + y$$

(x es el tamaño variable,
y es el corrimiento dentro de la tabla)

Aunque esta solución parece sencilla, nos obliga a modificar la forma de buscar una clave en la tabla. Para ello debemos realizar varios hashes con la clave hasta agotar las variables x,y o hasta encontrar el dato. Tenemos que almacenar adicionalemente los valores x,y en una tabla especial. X nos indica el tamaño de la tabla en un momento dado e y el corrimiento (offset).

Hay muchas más soluciones, más complejas o más simples, que logran resultados muy buenos y casi no generan sobrecarga.

Conclusiones

En estas resumidas páginas nos adentramos en un concepto de informática poderoso, que es extensivamente usado hoy en día.
Vimos que el hashing tiene muchos campos de uso, como las tablas hash o la criptografía.
Con respecto a la tablas hash, sus ventajas principales son la velocidad para encontrar datos y la simplicidad del concepto. Sus mayores desventajas son que no permite almacenar los datos ordenados y desperdicia cierto espacio de almacenamiento.
Sin embargo, a nuestro criterio (el equipo de Peiper) las ventajas avasallan las desventajas y es muy conveniente usar este tipo de tablas en software que requiere búsquedas rápidas de información. Y ni hablemos de los árboles hash, que son capaces de albergar cantidades increíbles de información con tiempos de búsqueda casi constante.
En la criptografía la principal desventaja es que siempre existirán colisiones si el tamaño de la clave hash es menor que el tamaño de la entrada. Su ventaja es que permite firmar los mensajes de forma que se detecten cambios realizados sobre los datos.