Archivos Hash: Implementación y Aplicaciones

Frittelli, V. – Steffolani, F. – Harach, J. – Serrano, D. – Fernández, J. – Scarafia, D. – Teicher, R. – Bett, G. – Tartabini, M. – Strub, A.
Universidad Tecnológica Nacional, Facultad Regional Córdoba

Abstract
Los conceptos, elementos y técnicas para implementar una tabla hash en memoria son perfectamente extrapolables a la organización de un archivo en disco para favorecer búsquedas externas rápidas, siempre que se cuente con la capacidad del acceso directo al contenido del archivo y se esté dispuesto a permitir espacio extra de almacenamiento. Este trabajo propone algunas ideas y líneas prácticas que conducen a la implementación de archivos organizados como tablas hash, así como la exposición de experiencias concretas de aplicación en el ámbito académico (como parte de trabajos y de aplicaciones de soporte de datos en proyectos de investigación) y en el contexto profesional (como parte del proceso de control de transacciones y saldos en redes de dispositivos para manipulación de tarjetas sin contacto).

Palabras Clave
Tablas Hash. Direccionamiento Abierto. Listas de Desborde. Implementación de Archivos con Estructura Hash. Aplicaciones del Hashing Externo.

Introducción

Una *tabla hash* es una estructura de datos que permite almacenar objetos (o registros) de forma tal que luego una búsqueda tenga tiempo de recuperación constante (o sea, *O(1)*) sin importar la cantidad de elementos que tenga la tabla. La forma más obvia de lograr tal meta, consiste en dimensionar una tabla suficientemente grande para que pueda alojar todos los objetos que se necesita almacenar, y luego grabar cada objeto en la casilla cuyo índice coincida con el valor usado como clave o identificador primario del objeto (Drozdek, 2007) (si se trabaja en plataforma Java, normalmente en un ambiente polimórfico se usa como clave el valor retornado por el método *hashCode()* de cada objeto). Sea un conjunto de *n* objetos $\{O_0, O_1, O_2, \ldots O_{n-1}\}$, tales que k_i es el valor *hashCode()* del objeto O_i, al que denotamos como $O_i(k_i)$. Si *t* es una tabla de tamaño $m >= n$, entonces (en principio) $t[k_i]$ debe usarse para almacenar el objeto $O_i(k_i)$ y de esta forma, la recuperación de un objeto desde la tabla es directa: sólo necesitamos saber su clave (su *hashCode()*) y ella nos dice en qué casilla está el objeto. Sin embargo, sabemos que esto no siempre es viable: si la clave puede ser un número cualquiera en cualquier rango numérico, entonces la tabla debería ser extremadamente grande para posibilitar que cualquier valor sea un índice válido. Un ejemplo conocido se da cuando la clave es el número del documento nacional de identidad (o DNI) de cualquier ciudadano argentino: como ese número tiene un máximo de 8 dígitos, la tabla debería ser tan grande como de 100 millones de casillas para que cualquier DNI pueda ser un índice válido, aun cuando luego solo se almacenen unas pocas docenas de objetos.

La técnica conocida como *dispersión de claves* o simplemente *hashing*, busca resolver este problema usando una función *h* designada como *función de dispersión*, *función hash* o *función de transformación*. Esta función es provista por el programador y la idea es que tome como parámetro una clave de un objeto y retorne un índice válido para una tabla *t* con capacidad para *m* objetos. Así, si el objeto O_k tiene un valor de *hashCode()* igual a *k*, entonces O_k debe almacenarse en *t[h(k)]*:

$k = O_k$.hashCode();
$t[h(k)] = O_k$

Intuitivamente, se esperaría que el método *hashCode()* retorne un número entero que en la medida de lo posible sea diferente para dos objetos que sean considerados distintos, aunque esto no es obligatorio: dos

objetos diferentes de cualquier clase del programador podrían tener el mismo *hashCode()*.

Una manera obvia y trivial de implementar una función *h* de dispersión para una tabla de tamaño *m*, es hacer que *h* retorne el resto de dividir la clave del objeto por el tamaño *m* de la tabla (Drozdek, 2007), aunque solo con este recurso no se garantiza que se obtenga una buena función de dispersión, ya que puede aumentar la probabilidad de producirse *colisiones* de claves. Una *colisión* es una situación en la que dos objetos diferentes obtienen el mismo índice para entrar en la tabla. Es decir, la función de dispersión calcula el mismo valor para las claves de esos dos objetos. Esto puede producirse porque ambos objetos tienen la misma clave (el mismo *hashCode()*), o porque teniendo claves diferentes la función *h* de dispersión calcula el mismo valor para ambos. Por caso, si se usa simplemente la técnica de retornar el resto de dividir la clave por el tamaño de la tabla, se tendrá una colisión cada vez que dos claves sean *congruentes módulo tamaño de la tabla*.

De lo anterior resulta claro que una buena función *h* de dispersión debería cumplir con ciertas propiedades deseables para evitar (o al menos reducir) situaciones de colisión. Las dos propiedades más obvias son las siguientes (aunque pueden enunciarse otras dependiendo del contexto de aplicación):

- *Determinismo*: la función debería retornar el mismo valor cada vez que se la invoque para la misma clave.
- *Uniformidad*: todos los posibles valores de salida, deberían tener la misma probabilidad de ser calculados y retornados.

Como sea, una vez que se ha diseñado una buena función de dispersión y dado que no es posible almacenar dos o más objetos en la misma posición de una tabla, se usan estrategias para evitar el problema de las eventuales colisiones, conocidas como *técnicas de resolución de colisiones*. Existen dos de estas técnicas que son consideradas las más tradicionales (Langsam, Augenstein, & Tenenbaum, 1997):

- ✓ *Direccionamiento Abierto*: cada entrada de la tabla almacena un objeto (y sólo uno). Al principio, cada casilla está vacía o *abierta* (de allí el nombre de esta técnica). Si un objeto O_1 se mapea a la casilla *i* y la misma está abierta, O_1 se almacena en ella. El índice *i* de esa casilla se dice la *dirección madre* de O_1. A partir de este momento, la casilla *i* está *cerrada*. Si otro objeto O_2 colisiona en la casilla *i*, se prueba en la casilla *i+1*. Si está abierta, O_2 se almacena en ella. Pero si está cerrada, se sigue probando con *i+2*, *i+3*, *i+4*... (haciendo lo que se conoce como una *exploración lineal*) hasta llegar a una casilla abierta y en ella se almacena el nuevo objeto. Notar que entonces, varios objetos serán almacenados fuera de su dirección madre (aunque se esperaría que no muy lejos de ella en sentido secuencial). Si en la exploración lineal se llega hasta el final de la tabla sin encontrar una casilla abierta, se sigue desde la casilla cero (circularizando la tabla). Para buscar un objeto, se obtiene su dirección madre con la función *h* y se entra en la tabla en esa dirección. Si el objeto en esa casilla no es el buscado, se explora hacia abajo hasta encontrarlo (búsqueda exitosa), o hasta encontrar una casilla abierta (búsqueda infructuosa). En general la técnica de resolución de colisiones por direccionamiento abierto permite encontrar un objeto en tiempo *prácticamente constante* si la técnica está bien implementada: puede verse que se requiere un acceso directo a la dirección madre, y una muy corta exploración lineal en el peor caso para hallar el objeto. Puede preverse que a medida que la tabla se llene, las exploraciones lineales serán cada vez más largas, lo cual sugiere que la tabla

debería empezar con un tamaño mayor al número de objetos esperados y controlar el porcentaje de ocupación. Si el mismo llega a cierto valor crítico, se debería redimensionar la tabla, creando otra de mayor capacidad y volviendo a insertar en ella los objetos que tenía la anterior. Por otra parte, aun cuando el porcentaje de ocupación de la tabla sea adecuado, la exploración lineal tiende a producir cierto comportamiento no deseado conocido como *agrupamiento primario*, que es la tendencia de los objetos de la tabla a formar grupos o "islas" dentro de esa tabla: la tabla presenta muchos espacios libres, pero los objetos tienden a caer cerca de las mismas direcciones madre. Esto provoca que para ciertas claves, la inserción resulte muy costosa pues deben acomodarse dentro de un grupo ya grande que además crece en tamaño con la inserción de las nuevas claves. Una solución para el agrupamiento primario consiste en realizar lo que se conoce como *exploración cuadrática* (en lugar de exploración lineal): Si la casilla *i* está ocupada, se sigue con *i + 1* y luego con *i + 4*, *i + 9*, ... $i + j^2$ (con *j=1, 2, 3,* ...) Está claro que el agrupamiento primario se rompe con este tipo de exploración, pero ahora se produce el *agrupamiento secundario*: dada una dirección madre *i*, siempre se exploran las mismas casillas de allí en adelante, lo cual lleva a otro posible problema: la exploración cuadrática podría no garantizar que una clave se inserte finalmente en la tabla, aun habiendo lugar libre. No obstante, se puede probar (Weiss, 2000) que si el tamaño de la tabla es un número primo y el porcentaje de ocupación no es mayor al 50% de la tabla, entonces la exploración cuadrática *garantiza* que la clave será insertada.

- ✓ *Listas de Desborde*: también designada como *Encadenamiento Separado*. Cada casilla de la tabla contiene *una lista*, en la que se guardan los objetos que colisionaron en esa entrada. Al insertar un objeto, la función de dispersión indica en qué lista se debe agregar el objeto. De nuevo, esto implica que para buscar un objeto se debe hacer un acceso directo y (si la técnica está bien implementada) un *corto* recorrido secuencial en una lista, manteniendo el tiempo de búsqueda en orden prácticamente constante (en rigor, el tiempo de búsqueda será proporcional a la longitud de la lista en la que se debe buscar). Está claro que ahora no es importante el nivel de ocupación de la tabla, sino la *longitud promedio* de las listas en ella, pues de allí se obtiene el rendimiento promedio de una búsqueda. Si se puede asegurar que cada lista tenga entre ocho y diez nodos la técnica funcionará bien en la práctica (Sedgewick, 1995), lo cual permite pensar en tablas con un tamaño no tan grande como para tener un amplio espacio usado en listas que podrían estar vacías. Si se puede asegurar que las listas no crecerán demasiado, entonces puede evitarse tener que ampliar el tamaño de la tabla y volver a dispersar todas las claves en ella.

Elementos del Trabajo y Metodología

Se busca ahora llevar a la gestión de archivos las ideas generales que se expusieron en la sección anterior, para permitir búsquedas rápidas en memoria externa. Se pretende mostrar en qué forma podría diseñarse un marco de trabajo consistente en clases que implementen el *hashing externo*: un archivo de registros con acceso directo pueda organizarse él mismo como un archivo cuya estructura interna sea la de una tabla hash. Obviamente, lo que sea que se haga dependerá fuertemente de la técnica de resolución de colisiones que se decida implementar. Si bien cualquiera de ellas puede implementarse sin mayores problemas, en este trabajo se muestran líneas generales de acción para implementar

la técnica de *listas de desborde*. Se han propuesto otras variantes más concretamente orientadas a la organización de archivos con estructura hash en los que se espera que la tabla tenga tamaño variable (Fagin, Nievergelt, Pippenger, & Strong, 1979) y (Folk, Zoellick, & Riccardi, 1998). También se ha analizado el impacto de la aplicación del hashing externo cuando se cuenta con algún sistema de memoria interna a modo de buffer (Wei, Yi, & Zhang, 2009).

En el caso que aquí se presenta, el modelo de implementación del hashing externo que se propone tiene como antecedente inmediato el trabajo designado como "*Archivos de Acceso Directo: un Enfoque desde Java y la POO*" (Frittelli, Steffolani, Scarafia, & Serrano, 2011) en el cual se propuso un marco de trabajo para archivos de registros con acceso directo denominado *framework RegisterFile*, que se usa aquí como marco de referencia y puede ampliarse o modificarse según la necesidad del programador (de hecho, ese framework ya brinda mucha de la funcionalidad de acceso directo que se necesita en un archivo hash). En el marco citado, se cuenta con la clase abstracta *RegisterFile* que propone métodos abstractos para insertar, remover, modificar y buscar registros. Esta clase utiliza internamente un objeto de la clase *java.io.RandomAccessFile* para manejar el archivo de datos que será usado para implementar el hashing externo. Si consideramos a los servicios de la clase *RandomAccessFile* como servicios de bajo nivel, entonces podemos decir que la clase abstracta *RegisterFile* brinda servicios de nivel intermedio: sus métodos concretos invocan a servicios de la clase *RandomAccessFile* y eventualmente los complementan. Los servicios de alto nivel (como serían los métodos que específicamente se orientan a manejar un registro y accederlo en forma directa), quedan para las derivadas concretas.

Para comenzar a implementar el concepto de archivo hash, se ha introducido una segunda clase abstracta llamada *HashFile*, derivada desde *RegisterFile*. La clase *HashFile* aporta ya programada la función de dispersión *h* en su forma más trivial y más básica (quedando para el programador la tarea de modificarla y adaptarla según su criterio, en esa misma clase o en alguna de sus derivadas), más un par de métodos estáticos para determinar si un número es primo y el siguiente número primo a un número *n* dado (Weiss, 2000), que serán útiles cuando se quiera expandir la tabla: independientemente de si se usa o no el mecanismo de direccionamiento abierto, es una buena idea hacer que el tamaño de la tabla sea un número primo (Weiss, 2000). La clase provee también un método abstracto *tableSize()* que debería retornar el tamaño de la tabla manejada por cada derivada. Como cada derivada podría manejar tablas conceptualmente diferentes, la forma de determinar el tamaño de cada una se dejó para las derivadas. Un esquema muy general de esa clase en Java puede verse aquí:

```
public abstract class HashFile extends RegisterFile
{

   public HashFile ( )
   {
      super( "newfile.dat", "rw" );
   }

   public HashFile(String nombre, String modo)
   {
      super( nombre, modo );
   }

   public HashFile(File file, String modo)
   {
      super( file, modo );
   }

   public static long siguientePrimo( long  n )
   {
      if( n <= 1 ) return 3;
      if( n % 2  == 0)   n++;
      for( ;  !esPrimo( n ); n += 2 ) ;
```

```
      return n;
   }

   public static boolean esPrimo( long n )
   {
      if( n == 2 ) return true;
      if( n % 2 == 0 ) return false;
      long raiz = ( long ) Math.sqrt( n );
      boolean ok = true;
      for( long div = 3; div <= raiz; div += 2 )
      {
         if( n % div == 0 ) return false;
      }
      return true;
   }

   protected long h( Grabable obj )
   {
      int k = Math.abs( obj.hashCode() );
      return k % this.tableSize();
   }

   protected abstract long tableSize();
}
```

Las clases que implementen el concepto de archivo hash, entonces, deberían derivar desde *HashFile* y aprovechar con ello las definiciones previas ya vistas. En este trabajo, de hecho, se discute la forma de implementar la técnica de resolución de colisiones por listas de desborde agregando una nueva clase, llamada *ListHashFile*, que derive desde *HashFile*.

En principio, los constructores de la clase *HashFile* y sus derivadas deberían abrir el archivo donde se almacenarán los datos, y podría pensarse que también deberían pedir espacio de disco para crear la tabla inicial. Sin embargo, en este modelo, eso no es válido por un sencillo hecho: los archivos representados por *RegisterFile* usan los primeros bytes del archivo de soporte para grabar el *identificador completamente calificado* de la clase base del archivo (la clase de los objetos que se pueden grabar en él) Si al invocar un constructor el archivo fuera creado, ese constructor no sabría de qué tipo serán los objetos que luego serán grabados, y por lo tanto no podrá grabar la clase base. Y sin la clase base, tampoco puede crearse la tabla pues no sabría desde qué byte comenzar a grabarla. En el framework de *RegisterFile*, el nombre de la clase base se graba al invocar a algún método de inserción (*add(), append()*, etc.) Si alguno de esos métodos descubre que el archivo está vacío, graba el nombre de la clase base y luego hace la grabación del objeto. En este nuevo modelo para archivos hash, la idea es la misma: si alguno de esos métodos descubre que el archivo está vacío, deberá grabar el nombre de la clase y a *continuación pedir espacio para la propia tabla vacía*. A modo de ejemplo, se muestra el método *append()* de la clase *ListHashFile*:

```
public boolean append ( Grabable obj )
   {
      if( getMode().equals( "r" ) ) return false;
      if( ! isOk( obj ) ) return false;

      if( clase == null )
      {
         writeClassName( obj );
         createTable(11);
      }
      long y = h(obj);
      HeaderList hl = this.getHeaderList(y);
      agregar( hl, obj );
      this.setHeaderList( y, hl );
      return true;
   }

   private void createTable( long n )
   {
      if( clase == null || getMode().equals( "r" ) )
          return;

      // ajustar el tamaño de la tabla...
      if( n <= 0 ) n = 11;
      capacity = n;

      // grabar ese tamaño de la tabla...
      try
      {
         maestro.writeLong( capacity );
      }
      catch( IOException e )
      {
         JOptionPane.showMessageDialog(  null,
              "Error:  " + e.getMessage() );
         System.exit(1);
```

```
    }

    begin_headers = this.bytePos();

    // crear la tabla de nodos cabecera…
    Register r = new Register(new
          HeaderList());
    for(int i = 0; i < capacity; i++)
    {
       this.write( r );
    }
}
```

La clase *ListHashFile* usa el método *createTable()* que se acaba de mostrar, para crear la tabla que contiene a las distintas listas. La idea es que en cada posición de la tabla, se grabará un objeto de la clase *HeaderList*, tal que cada uno de estos objetos contiene dos atributos: un valor *long* llamado *first*, que tendrá la dirección de byte del primer nodo de esa lista, y otro valor *long* llamado *size* que será un contador en el cual se llevará la cuenta de la cantidad de nodos que esa lista tiene.

El marco de *RegisterFile* original prevé una clase *Register* que contiene una referencia al objeto cuyos datos serán grabados y un atributo adicional usado como marca de borrado lógico. Los objetos a grabar son instancias de clases que implementan la interface *Grabable*, la cual impone métodos para grabar y leer un objeto desde un *RandomAccessFile*. Esto sigue vigente en este modelo de archivos hash: los objetos que se graban en el archivo en principio se graban dentro de un *Register*, pero hay que prever un detalle: cada *Register* debe ahora incluir un atributo que contenga la dirección de byte del registro siguiente a él en la lista de desborde. Como la clase *Register* original no tiene tal atributo, la solución propuesta aquí consiste en declarar una nueva clase *NodeRegister* (derivada de *Register*), tal que ese "puntero" esté contenido en ella. Así, un *NodeRegister* es un *Register*, que además tiene un atributo *next* de tipo *long*, para almacenar la dirección del siguiente. Como ahora esas direcciones son simplemente valores *long* que representan posiciones de disco, entonces para simbolizar el valor *null* la clase *NodeRegister* provee una constante llamada *NIL*, cuyo valor es *–1*. Así, para indicar que un nodo no tiene sucesor o que la lista está vacía, se asignará *NIL* al "puntero" correspondiente.

Considerando todos estos elementos, finalmente un archivo *ListHashFile* tendrá una estructura como se ve en el gráfico siguiente, suponiendo a modo de ejemplo que la tabla contiene tres listas de desborde:

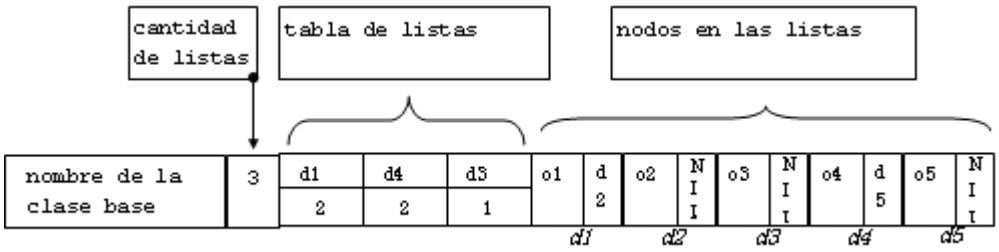

Figura 1: Estructura interna de un ListHashFile

En el gráfico de la *Figura 1* se supone que la primera lista contiene dos nodos: los objetos *o1* (en la dirección *d1*) y *o2* (en la dirección *d2*). El atributo *first* del primer *HeaderList* contiene el valor *d1* (dirección del primer nodo de la lista) y el atributo *size* de ese *HeaderList* vale *2* (pues la lista tiene dos nodos). El atributo *next* del nodo *o1* vale la dirección *d2*, que es el lugar donde está grabado el nodo *o2*. En forma similar, la segunda lista contiene a los nodos *o4* y *o5* (en las direcciones *d4* y *d5*), y la tercera lista contiene sólo al nodo *o3* en la dirección *d3*.

Para insertar un nuevo registro, primero se obtiene su dirección madre con la función *h*. Ese valor indicará en que *HeaderList* de la tabla entrar para tomar la dirección del nodo inicial de esa lista, desde su atributo *first*. Si esa dirección vale *–1*, la lista está vacía. Si no, la lista tiene algún nodo ya grabado. Si está vacía, simplemente se va al final del archivo, se toma el número de byte que corresponde a esa posición, se graba el nuevo registro en ese lugar y se almacena en el atributo *first* del *HeaderList* de esa lista el valor del byte donde se grabó. Si la lista tenía ya algún nodo grabado, *first* indica donde está el primero. Se usa *RegisterFile.seekByte()* para ubicarlo, y luego se lee. Si no es el registro que se quiere insertar, se toma la dirección indicada en su atributo *next*, y se accede a ese registro. Se repite la operación hasta dar con un nodo cuyo *next* sea *–1* o hasta encontrar un nodo repetido. Si no se encuentra uno repetido, se va al final del archivo y se graba el registro allí. Pero ahora se deberá actualizar el campo *next* del último registro que se leyó en la lista. El método *add()* de la clase *ListHashFile* hace estas operaciones, con el apoyo de algunos métodos auxiliares, y el resto de las operaciones (bajas, modificaciones, búsquedas, etc.) pueden deducirse de las ideas anteriores.

Finalmente, un control importante que debe realizarse es el del tamaño promedio de las listas de desborde. El proceso implementado debería monitorear ese largo promedio de las listas, y en caso de llegar al punto de intervención, tomar el control y aumentar el tamaño de la tabla (por caso, proponer un 50% más que el tamaño anterior, pero tomando como nueva capacidad al primer número primo que sea mayor a ese 50% de incremento). El proceso es simple pero debe hacerse con cuidado: se crea la nueva tabla (un nuevo archivo temporal), se toman todos los registros que estaban en la anterior (en el viejo archivo), y se redispersan en la nueva tabla: no se puede sólo copiar los registros de una a otra, pues los tamaños de ambas no coinciden y los valores calculados por *h* no serán consistentes. Al terminar, el viejo archivo se elimina, y el nuevo debe tomar el nombre del eliminado. También es posible pensar en un esquema de eliminación física de los nodos/registros marcados como eliminados, pero muy posiblemente se decida que eso sólo sea necesario realmente cuando se hace el proceso de redispersión citado más arriba.

Resultados y Aplicaciones

Los archivos hash brindan una plataforma para soporte de datos con tiempo de ejecución constante en operaciones de búsqueda, inserción, eliminación y modificación. En la práctica profesional, normalmente es tarea de los programadores el diseño de clases que apliquen estas técnicas para aprovechar la potencia del hashing en sistemas de almacenamiento externo (Folk, Zoellick, & Riccardi, 1998): las plataformas de desarrollo incluyen tablas hash para memoria interna, pero rara vez proveen sistemas de almacenamiento externo basados en dispersión de claves. Sin embargo, muchas estructuras de datos para realizar búsquedas veloces han sido directamente diseñadas para ser aplicadas en memoria externa y en esos casos suelen hacerse extrapolaciones hacia la memoria interna con fines didácticos, como es el caso de los ár*boles B* y todas sus variantes, ampliamente usados como soporte de índices en la gestión de bases de datos (Folk, Zoellick, & Riccardi, 1998), o la variante del hashing conocida como *hashing extensible* (Fagin, Nievergelt, Pippenger, & Strong, 1979), diseñada para contextos en los que se espera que la tabla tenga tamaño variable (Drozdek, 2007) (Folk, Zoellick, & Riccardi, 1998).

Un contexto concreto y real de aplicación en este sentido, surge del campo de la conciliación de saldos y transacciones entre los componentes de una red cuyo objetivo es la manipulación de saldos en

tarjetas sin contacto (como es el caso de programas de loyalty, sistemas de transporte público de pasajeros, o incluso las tarjetas de crédito NFC). Desde un punto de vista muy general la red cuenta con los siguientes componentes (ver *Figura 2*):

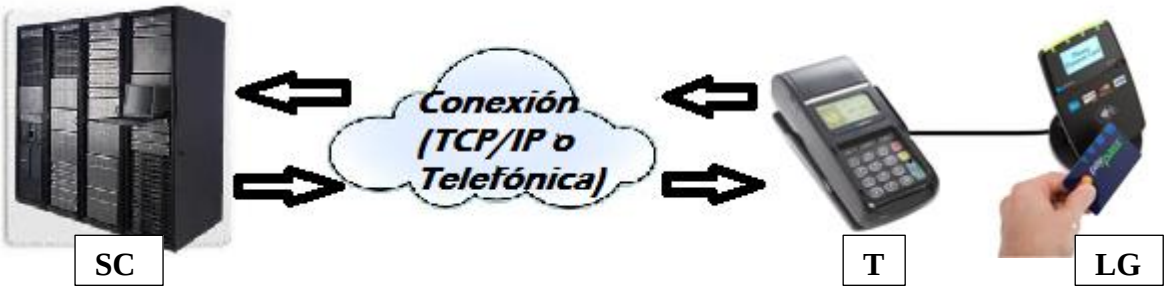

Figura 2: Esquema General de Componentes en una Red de Manipulación de Tarjetas

El *terminal T* y el *dispositivo de lectura y grabación LG* (que es el que contiene la capacidad de modificar la memoria de la tarjeta sin contacto), son equipos que están conectados entre ellos y se encuentran distribuidos a lo largo de los distintos puntos de venta. Existe un *sistema central SC* al que le compete tener el detalle de todas las transacciones realizadas durante los distintos ciclos de operaciones. El terminal *T* tiene la interfaz con el vendedor en el punto de venta (ya que posee teclado, pantalla, impresión de tickets, etc.) y tiene, además, la capacidad de comunicarse con el sistema central *SC*. El dispositivo *LG*, por su parte, tiene la capacidad de recibir órdenes del terminal *T* e impactar los saldos sobre las tarjetas, pero carece de cualquier interfaz que lo comunique con otro dispositivo que no sea el terminal *T*. Como es de suponer, los dispositivos *T* y *LG* mantienen su propio archivo de transacciones realizadas. En el caso ideal, ambos archivos contendrán las mismas transacciones, aunque los datos que se guarden en uno y en otro no sean necesariamente los mismos.

Por otro lado, el encargado de enviar *ambos* archivos es el terminal *T* ya que, como se dijo, es el único que posee conectividad con el sistema central *SC*. A modo de consideraciones adicionales, se puede mencionar que es importante que ambos archivos registren las mismas transacciones (aunque posean estructuras distintas), ya que aunque ambos dispositivos operen juntos, podrían llegar a ser de empresas distintas y por lo tanto el cotejo de estos registros es lo que permitiría realizar un proceso de clearing entre ambas. Si este es el caso, el terminal *T* y el dispositivo *LG* compartirán algunos campos comunes en cada transacción que almacenen, pero ocultarán y codificarán el resto y finalmente cada archivo será enviado a la empresa correspondiente.

El problema es que si bien en condiciones normales ambos archivos deben contener las mismas transacciones, asegurar la atomicidad de una transacción cuando hay tres dispositivos (el terminal *T*, el dispositivo *LG* y la propia tarjeta) no es una tarea trivial. Es aun más complicado si se considera que entre el dispositivo *LG* y la tarjeta no hay contacto físico y la tarjeta puede alejarse en cualquier momento durante la transacción. Para agregar incluso mayor complejidad, el terminal *T* y el dispositivo *LG* pueden ser desconectados, o alguno de los dos puede tener una falla. Considerando que sea cual sea la aplicación, lo que se mueve en el sistema son transacciones comerciales, el vendedor puede tener intereses en que esas

transacciones no se registren. Se puede esperar entonces que se intenten realizar operaciones fraudulentas desconectando los dispositivos para tratar de evitar el registro de las mismas. Incluso si no hay una intención fraudulenta, las condiciones de operación de los equipos pueden llevar a que en algún momento dado del ciclo de operación, alguna transacción no sea registrada en el terminal *T* (desconexiones accidentales, reinicios, fallas de hardware, etc.). Desde luego que tratándose de transacciones comerciales, al fin del ciclo de operación el terminal *T* debe garantizar que ha logrado registrar todas las operaciones que han sido realizadas y además, debe obtener las del dispositivo *LG* para informarlas al sistema central *SC*.

Para lograr el correcto control del ciclo de operaciones y la conciliación final entre los registros de *T* y los registros de *LG*, en una situación de aplicación real, se diseñó una implementación basada en la modificación del firmware del terminal *T*, dejando intacto el firmware del dispositivo *LG*. Un dato a considerar es que la cantidad máxima de transacciones que puede almacenar cualquiera de los dos equipos en un ciclo de operación es fija y conocida. Otra característica de este escenario es que el ordenamiento por algún campo del archivo de transacciones del terminal *T* no puede ser garantizado en todas las condiciones de trabajo. Y por último, se impuso una solución que tuviera un costo en tiempo de ejecución que no fuera elevado, considerando que los equipos son de una potencia de procesamiento moderada. Considerando la situación y los requisitos y restricciones que se acaban de describir, se decidió implementar un *archivo hash con listas de desborde*. El mismo es inicializado cada vez que comienza un ciclo de operaciones. Por cada operación que el terminal *T* registra, se agrega al archivo hash una entrada asociada a una clave elegida de acuerdo a los valores que identifican en forma unívoca a cada transacción y que pueden ser obtenidos tanto en el registro del terminal *T* como en el del dispositivo *LG*. Al momento de finalizar el ciclo de operación, a medida que el terminal *T* recibe el archivo de operaciones del dispositivo *LG*, se obtiene la clave, se realiza el cálculo de la dirección de entrada mediante la función de dispersión y se utiliza la tabla previamente creada y mantenida para conocer en tiempo constante si la transacción se encuentra o no en los registros del terminal *T*. Si la transacción no se encuentra, se reconstruye con la mayor cantidad de campos posibles (recordando que los registros del terminal *T* y del dispositivo *LG* no tienen la misma estructura). Como ventaja adicional, el uso de un archivo hash en este proceso permite obtener en cualquier momento del ciclo de operación (y muy velozmente) información sobre una transacción previa (por ejemplo, en el caso que se quiera reimprimir un comprobante).

Discusión

El uso de archivos hash conduce a búsquedas externas muy rápidas, de tiempo de ejecución constante, aunque al costo de utilizar espacio extra de memoria para mantener la tabla y/o las listas de desborde. Además, en general las técnicas de hashing están pensadas y diseñadas para búsquedas muy rápidas pero no necesariamente para otras operaciones tales como los recorridos secuenciales ordenados (Drozdek, 2007): normalmente una tabla hash (en memoria interna o externa) no está pensada para permitir recorridos ordenados, y de hecho, ni siquiera para garantizar que distintos recorridos en distintos momentos produzcan la misma secuencia de iteración (y por este motivo, las tablas hash suelen ser muy útiles para implementar la idea de *conjunto* o *set* como colección desordenada de datos).

Obviamente, si se necesita organizar datos para búsquedas muy veloces y no se tiene problema con las desventajas citadas, las tablas hash son la mejor opción.

Concretamente, además de la situación de conciliación de transacciones citada en el punto anterior, los autores de este trabajo tienen previsto usar archivos hash como soporte de datos que requieren ser ubicados y recuperados muy rápidamente en el contexto del diseño de *motores de juegos* para un proyecto de investigación orientado al diseño de juegos para la enseñanza de la Inteligencia Artificial (Frittelli, et al., 2013), en los que el tiempo de respuesta es crítico no sólo por las demoras producidas por constantes accesos a memoria externa, sino también por las operaciones de despliegue de gráficos que se modifican en tiempo real en la interfaz de usuario final.

Conclusión

El diseño y desarrollo de sistemas de archivos con características especiales no suele ser tema de estudio en carreras informáticas, ni tampoco un área en la que a nivel profesional se trabaje con profundidad. En general, se tiende a confiar en que un motor de bases de datos brindará todo el soporte y toda la eficiencia que haga falta para gestionar los datos que se necesite almacenar en forma externa, o bien se confía en que las diversas herramientas para manejo de archivos que proveen los lenguajes y plataformas de programación serán suficientes "tal como vienen" para satisfacer todo requerimiento de manejo de memoria externa. Sin embargo, se pierde de vista que las distintas estructuras de datos que se estudian, proponen y diseñan para realizar tareas eficientemente en memoria interna, pueden aplicarse en forma natural también para trabajos en memoria externa, fundamente si lo que se requiere es un sistema de almacenamiento orientado a reducir tiempos de ejecución. Con este trabajo, se ha buscado hacer un aporte que muestre que es posible y es práctico el diseño de sistemas de archivos basados en *búsqueda por dispersión* o *hashing*, y que estos *archivos hash* tienen campos de aplicación concretos en situaciones de control de datos de transacciones comerciales o en soporte de datos para motores de juegos en los que se necesita velocidad de respuesta. El lenguaje Java provee paquetes de clases muy complejos y completos para el manejo de flujos de datos desde y hacia la memoria externa, y un estudio detallado de la forma de implementar archivos con estructura interna orientada a trabajos especiales se facilita aplicando esas herramientas en el contexto de la Programación Orientada a Objetos, ya que el encapsulamiento, la herencia y el polimorfismo permiten ir construyendo el marco de trabajo en forma progresiva, agregando capa por capa el manejo de la complejidad.

Referencias.

[1]. Drozdek, A. (2007). Estructura de Datos y Algoritmos en Java. México: Thomson.

[2]. Fagin, R., Nievergelt, J., Pippenger, N., & Strong, R. (1979). Extendible hashing—a fast access method for dynamic files. Journal ACM Transactions on Database Systems (TODS), 315-344.

[3]. Folk, M., Zoellick, B., & Riccardi, G. (1998). File Structures – An Object Oriented Approach with C++. Reading: Addison Wesley.

[4]. Frittelli, V., Steffolani, F., Scarafia, D., & Serrano, D. (2011). Archivos de Acceso Directo: Un Enfoque desde Java y la POO. In Z. Cataldi, & F. Lage (Ed.), Libro de Artículos Presentados en I Jornada de Enseñanza de la Ingeniería - JEIN 2011. I, pp. 165-174. Buenos Aires: Universidad Tecnológica Nacional.

[5]. Frittelli, V., Strub, A. M., Destéfanis, E., Steffolani, F., Teicher, R., Tartabini, M., et al. (2013). Motores de Juegos e Inteligencia Artificial para la Enseñanza. Workshop de Investigadores en Ciencias de la Computacion (WICC 2013). Paraná: Red UNCI.

[6]. Langsam, Y., Augenstein, M., & Tenenbaum, A. (1997). Estructura de Datos con C y C++. México: Prentice Hall.

[7]. Sedgewick, R. (1995). Algoritmos en C++. Reading: Addison Wesley - Díaz de Santos.

[8]. Wei, Z., Yi, K., & Zhang, Q. (2009). Dynamic External Hashing: The Limit of Buffering.

Proc. ACM Symposium on Parallelism in Algorithms and Architectures.

[9]. Weiss, M. A. (2000). Estructuras de Datos en Java - Compatible con Java 2. Madrid: Addison Wesley.

Datos de Contacto:

Ing. Valerio Frittelli.
UTN Córdoba - *vfrittelli@gmail.com*

Ing. Felipe Steffolani.
UTN Córdoba - fsteffolani@gmail.com

Ing. Jorge Harach.
UTN Córdoba - jorge.harach@gmail.com

Ing. Diego Serrano.
UTN Córdoba - diegojserrano@gmail.com

Ing. Julieta Fernández.
UTN Córdoba - jujulifer@gmail.com

Ing. Diego Scarafia.
UTN Córdoba - scarafia@gmail.com

Ing. Romina Teicher.
UTN Córdoba - rteicher@gmail.com

Ing. Gustavo Federico Bett.
UTN Córdoba - gfbett@gmail.com

Ing. Marcela Tartabini.
UTN Córdoba - marcelatartabini@hotmail.com

Ing. Ana María Strub.
UTN Córdoba - anastrub@gmail.com