Universidad de Buenos Aires
Facultad de Ingeniería

Organización de Datos (75.06)

Trabajo Práctico

Fecha: Primer Cuatrimestre de 2013

Profesor titular: Arturo Servetto

Docente a cargo: Maximiliano Stibel

Grupo: 2

Padrón	Nombre y apellido	E-mail
92168	Nadia Galli	nani16.galli@gmail.com
92235	María Inés Parnisari	maineparnisari@gmail.com
92454	Martín Zaragoza	zaragozamartin91@gmail.com
92691	Nicolás Sibikowski	niko.sibi@gmail.com
93395	Juan Federico Fuld	juanfedericofuld@hotmail.com

Índice

1 Objetivo

El objetivo del trabajo es crear una aplicación que permita a un usuario crear un índice sobre una biblioteca de canciones, y luego poder resolver consultas sobre dicha biblioteca, utilizando el índice.

2 Hipótesis de trabajo

1. Los archivos a indexar tendrán extensión `txt`.
2. Las canciones a indexar tendrán el siguiente formato:

```
<autor1>(;<autor2><autor3>...)(-<año>)-<titulo>-<idioma>
<letras>
```

   a) Los parámetros dentro de paréntesis son opcionales.
   b) El campo `<idioma>` debe ser alguno de los siguientes:
      1) en, english, inglés
      2) sp, spanish, español, espaniol, espanol

3. El usuario correrá la aplicación en el sistema operativo Linux.

3 Extensiones posibles

1. Utilizar técnicas de *stemming* y una lista de *stopwords* para reducir el tamaño final del índice.
2. Implementar el mecanismo de exclusión en la compresión PPMC para mejorar los resultados.

4 Fase de diseño

En la fase de diseño se obtuvo el siguiente diagrama final, que muestra las relaciones entre las diferentes estructuras de datos.

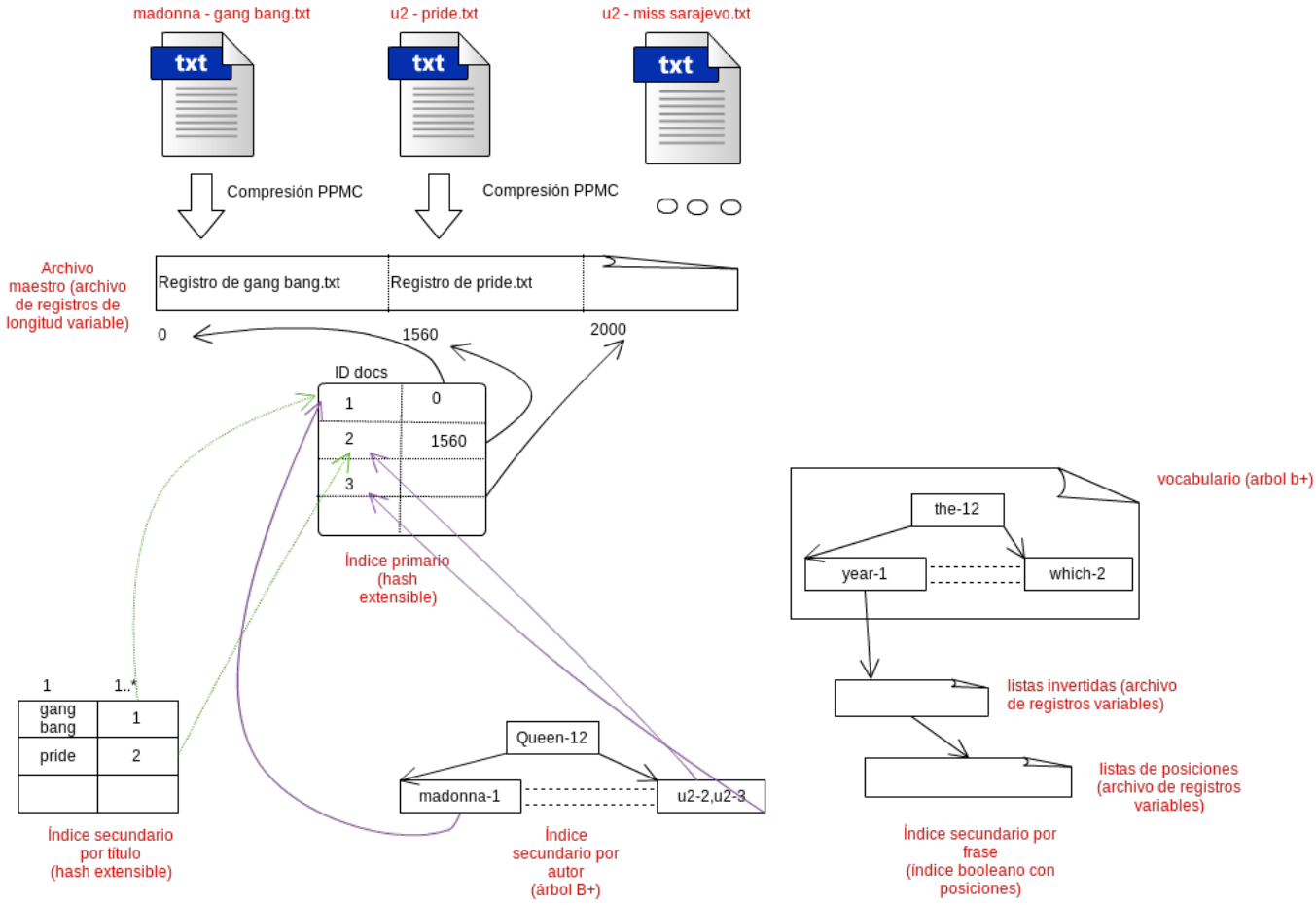

Figura 1: Diagrama general del trabajo.

5 Fase de implementación

Herramientas y conceptos utilizados

En el desarrollo del trabajo se aplicaron los siguientes conceptos:

- ▷ *Test Driven Development*: consiste en desarrollar pruebas unitarias y de integración del código producido.
- ▷ *Pair Programming*: consiste en programar de a pares, para así reducir la aparición de errores y de discutir posibles soluciones a problemas.

Se utilizaron las siguientes herramientas:

- ▷ Entorno de desarrollo: Eclipse
- ▷ Sistema operativo de desarrollo: GNU/Linux
- ▷ Herramientas adicionales: `valgrind`, `cppcheck`, `google test`

Descripción

El trabajo práctico fue dividido en tres capas:

▷ La **Capa Física** contiene las clases y métodos para trabajar con los datos a bajo nivel.

▷ La **Capa Lógica** utiliza las primitivas de la Capa Física para crear estructuras de datos más complejas y eficientes.

▷ La **Capa Interfaz** provee la comunicación entre el usuario final y la Capa Lógica.

Capa Física

El siguiente diagrama de clases ilustra los componentes esenciales de la capa física:

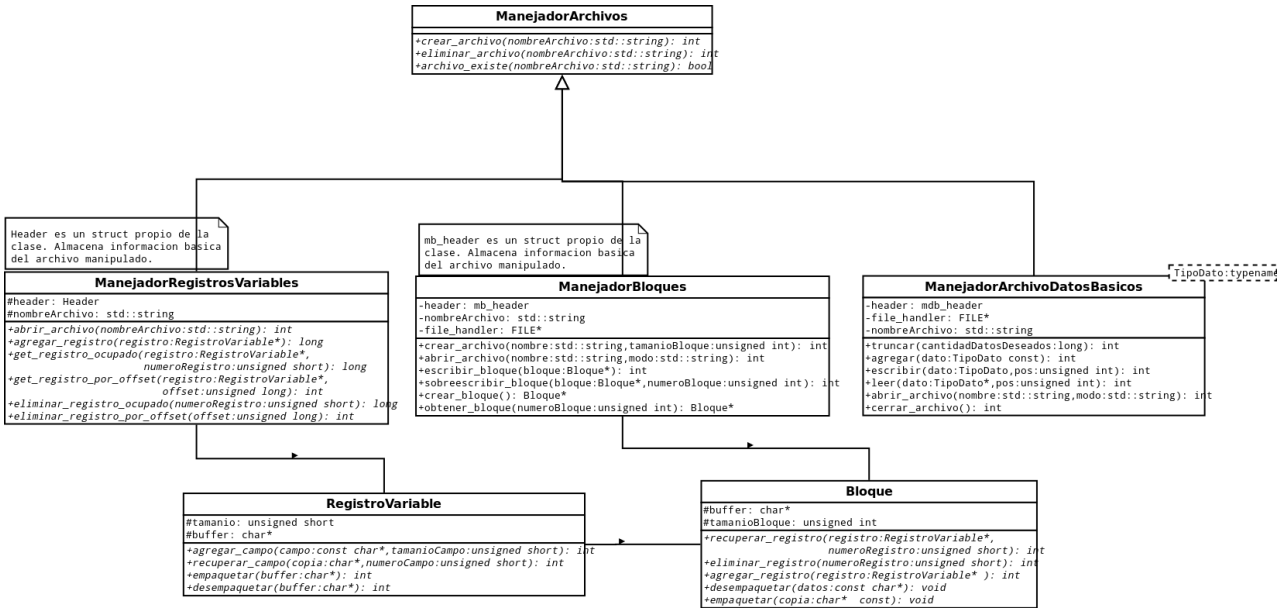

Figura 2: Diagrama de clase de capa física.

El diseño de las clases `RegistroVariable` y `Bloque` no presentaron problemas significativos. Las mismas se diseñaron para guardar datos de tipo genérico como arreglos de caracteres, los datos luego podrían recuperarse y reinterpretarse como fuese necesario con facilidad.

`RegistroVariable` guarda datos como campos junto con un prefijo de longitud de dicho campo.

▷ Un objeto `RegistroVariable` puede ser empaquetado y desempaquetado como un arreglo de bytes con el siguiente formato:

```
(int tamanioTotal; (int tamanioCampo , chars campo)+)
```

La clase `ManejadorRegistrosVariables` administra archivos que contienen `RegistrosVariables`.

▷ En el encabezado del archivo se guardan metadatos de cantidad de registros total, cantidad de registros libres y offset del primer registro libre.

▷ Los `RegistrosVariables` son guardados contiguos unos a otros junto con un prefijo de tamaño en bytes de cada uno.

▷ Los `RegistrosVariables` se los puede recuperar de dos formas:

- Especificando el número de registro que se desea, empezando desde el 0. La búsqueda se realiza en forma secuencial.
- Especificando el offset inicial del registro que se desea. El acceso es directo.

- ▷ Se utiliza una política de first-fit para la administración del espacio libre. Se utiliza una pila de registros libres o borrados cuyo primer elemento es apuntado desde el `header` (a partir del byte offset) del archivo. Al momento de insertar un registro, se busca en la pila el primer `RegistroVariable` eliminado cuyo tamaño sea suficiente como para guardar el registro nuevo; una vez hallado, el registro nuevo es guardado en la posición correcta del archivo y el registro libre anterior es removido de la pila de libres. Al momento de borrar un registro, el mismo se coloca al principio de la pila de libres y en el lugar que ocupaba el registro se conserva su prefijo de tamaño y se agrega el byte offset del próximo registro libre.

- ▷ La política first-fit puede no ser la más eficiente en cuanto a reutilización del espacio libre pero es la más veloz al momento de realizar una nueva inserción.

- ▷ Al momento de implementar la política de recuperación del espacio libre, para asegurar la integridad del archivo, al momento de buscar un registro libre para reutilizar, se verifica que su tamaño sea ligeramente superior al registro a insertar. Al recuperar un registro libre, el mismo se divide en dos partes, una es reutilizada para insertar el nuevo registro y otra permanece como registro libre pero desvinculado de la pila de libres. Esto se implementó de esta manera para preservar las búsquedas secuenciales en el archivo.

- ▷ Si se dan numerosas bajas, el archivo inevitablemente comenzará a fragmentarse , lo que implica que más adelante se deberá realizar una reestructuración o refactorización manual del mismo.

- ▷ El método `comprimir()` crea un nuevo `RegistroVariable` que contiene, primero, la cantidad de bytes originales en el registro (para poder descomprimir), y a continuación la compresión que resulta de comprimir el registro.

La clase `Bloque` guarda objetos del tipo `RegistroVariable` empaquetados como se especificó anteriormente. Los `Bloques` son de tamaño fijo; este tamaño se guarda dentro del archivo `Constantes.h`, y fue prefijado en 4 KB.

La clase `ManejadorBloques` administra archivos organizados en `Bloques` de tamaño fijo.

- ▷ La política de recuperación de espacio libre es de pila de bloques libres. Se decidió a favor de esta técnica y en contra del uso de un mapa de bits de bloques libres ya que nos pareció que era más sencillo implementarlo de esta manera.

- ▷ Para escribir un nuevo `Bloque` en el archivo se utiliza la primitiva `escribir`.

- ▷ Para actualizar un `Bloque`, el mismo debe ser leído del archivo, modificado y luego sobrescribirlo en el offset original usando la primitiva `sobreescribir`.

- ▷ Para eliminar un `Bloque`, se debe utilizar la primitiva `sobreescribir`, pasándole como parámetro un `Bloque` vacío.

- ▷ No hay fragmentación apreciable del archivo. El mismo se fragmentará de forma apreciable si se eliminaran numerosos bloques y cesara la inserción de datos.

- ▷ El mayor inconveniente para el diseño de ésta clase fue el diseño de una interfaz. Resultó complicado ocultar el funcionamiento interno de la clase y crear una interfaz sencilla que ocultara al usuario la manipulación a bajo nivel de los datos.

La clase `ManejadorArchivosBasicos` administra archivos que contienen datos de tipo básico (`int`, `char` , `short`, etc.).

- ▷ Se la implementó para facilitar el manejo de la Tabla de Hash.

- ▷ No presentó mayores dificultades de implementación.

Capa Lógica

Árbol B+

El siguiente diagrama de clases ilustra el diseño del árbol B+:

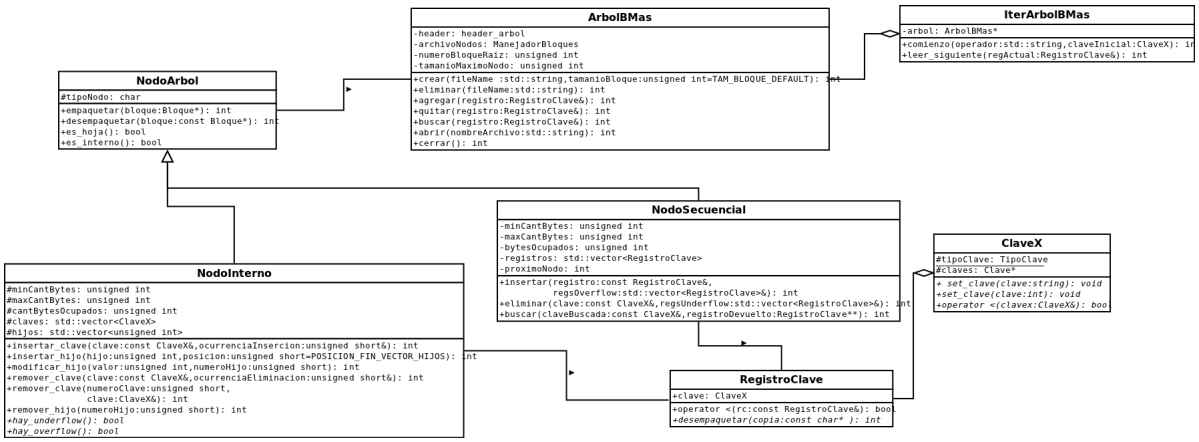

Figura 3: Diagrama de clases de árbol B+

Un problema en el diseño del árbol fue definir y controlar las condiciones de overflow y underflow. Otro inconveniente surgió al intentar abstraerse del tipo de clave que se utilizaría, esto se solucionó con la clase *ClaveX* descripta a continuación.

La clase `ArbolBMas` administra un archivo como una estructura de árbol B+. Se almacena en disco bajo la forma de un archivo en bloques (utilizando las primitivas de `ManejadorBloques`) con un bloque dedicado a cada `NodoArbol`.

La clase `RegistroClave` contiene un objeto de tipo `ClaveX` que nos habilita a abstraernos del tipo de clave utilizada para almacenar datos en el árbol. Tiene definidos todos los operadores de comparación. Los RegistroClave se empaquetan como un RegistroVariable, y se desempaquetan de una forma particular.

La clase `ClaveX` permite el manejo de claves de tipo `string` o `int` y puede ser ampliada para manejar otros tipos de datos. Contiene como elementos de un vector las claves de cada tipo que maneja. Sobrecarga sus métodos para recibir claves de distintos tipos y guarda internamente el tipo de clave con el que está trabajando.

La clase `NodoArbol` permite la existencia de estructuras distintas para los nodos internos y las hojas solucionando el inconveniente de guardar referencias nulas de manera innecesaria en el último nivel del árbol. Guarda un atributo que permite esta diferenciación.

La clase `NodoSecuencial` guarda objetos del tipo RegistroClave y representa las hojas del árbol. Puede informar si al insertar o eliminar un registro se produce un overflow o underflow respectivamente, además en cada caso devuelve en un vector los registros a ser redistribuidos. Contiene también una referencia al próximo `NodoSecuencial` del árbol, implementado como el número de bloque donde está guardado dicho nodo.

La clase `NodoInterno` guarda objetos del tipo `RegistroClave` y representa los nodos internos del árbol. Puede informar si al insertar o eliminar un registro se produce un overflow o underflow respectivamente, pero no devuelve en un vector los registros a ser redistribuidos. Almacena las claves y las referencias a sus hijos en memoria en dos vectores separados.

Hashing Extensible

El siguiente diagrama de clases ilustra el diseño del hashing extensible:

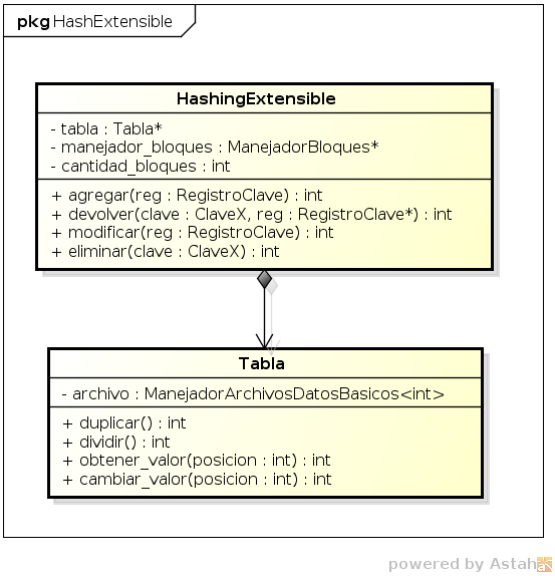

Figura 4: Diagrama de clases de Hashing Extensible.

La clase `HashExtensible` administra registros como una estructura de Hash Extensible prefijo.

▷ El `HashExtensible` está compuesto por dos archivos, uno es la `Tabla` y el otro es un `ManejadorBloques`[1] donde se guardarán los registros de tipo `RegistroClave`.

- Los datos de la `Tabla` se guardan como datos secuenciales, utilizando el `ManejadorArchivoTipoBasico` con `Tipo=int`. El tamaño de la `Tabla` se guardará también en dicho archivo. La `Tabla` solo guarda los números de bloque a los que se hace referencia en el archivo de bloques del hash. Por último, la `Tabla` tiene dos métodos especiales: uno para agrandarse (`duplicar()`) y otro para achicarse (`dividir()`).
- Cada bloque del `ManejadorBloques` contiene los registros a guardar y un atributo "tamaño de dispersión".

▷ Para insertar un `RegistroClave` en el archivo, se siguen los siguientes pasos:

1. Se le aplica una función de dispersión a la clave del `RegistroClave`. Esta función devuelve una posición de la `Tabla` donde se guarda el número del bloque en donde debemos guardar el `RegistroClave`.
2. Si al intentar insertar el `RegistroClave` en el bloque se produce overflow (es decir, el mismo no cabe en bloque), se obtiene el tamaño de dispersión del bloque.
3. Si el tamaño de dispersión del bloque es igual al tamaño de la tabla entonces:
   a) Duplicamos la `Tabla`.
   b) En el lugar de la `Tabla` donde se encontraba la dirección del bloque vamos a cambiarla por la posición de un nuevo bloque.
4. Si el tamaño de dispersión del bloque es menor al tamaño de la `Tabla` entonces vamos a recorrer la `Tabla` en forma circular desde la posición de tabla, donde el bloque se desbordó, haciendo pasos del tamaño de dispersión por dos y en esos lugares guardaremos la posición de un nuevo bloque.
5. Tomamos todos los registros del bloque más el registro que produjo overflow y los volvemos a insertar.

▷ Para eliminar un `RegistroClave`, se siguen los siguientes pasos:

[1] La `Tabla` es un archivo de control. Los dos archivos fueron separados para facilitar la implementación.

1. Se le aplica una función de dispersión a la clave del `RegistroClave`. Esta función devuelve una posición de la `Tabla` donde se guarda el número del bloque en donde debemos eliminar el `RegistroClave`.
2. Si al borrar el `RegistroClave` del bloque se produce underflow:
   - Tomar el tamaño de dispersión del bloque y moverse con esta distancia de izquierda a derecha de la posición de la `Tabla`, y vemos si guarda el mismo número de bloque.
   - Si son distintos no pasa nada, pero si son iguales se recorre la `Tabla` en forma circular desde la posición de tabla donde el bloque se desbordó, haciendo pasos del tamaño de dispersión por dos y en esos lugares guardaremos el número del bloque que reemplazará al anterior.
   - Si la primera mitad de la `Tabla` es igual a la segunda mitad, se divide la `Tabla`.

Para las búsquedas por título se utiliza un `HashingExtensible`, donde la clave a utilizar es el título normalizado de cada canción, y los posibles valores de la clave son los ID de las canciones que poseen dicho título.

Índices

Para la búsquedas por frase se implementó una estructura de tipo `IndiceInvertido`, que permite resolver consultas booleanas por frase.

Un `IndiceInvertido` es una estructura de datos que permite recuperar los "documentos" (en este caso, letras archivos de canciones) que contienen ciertos "términos" (en este caso, palabras de una letra de una canción).

El siguiente diagrama ilustra la clase correspondiente al `IndiceInvertido` para la búsqueda por frases:

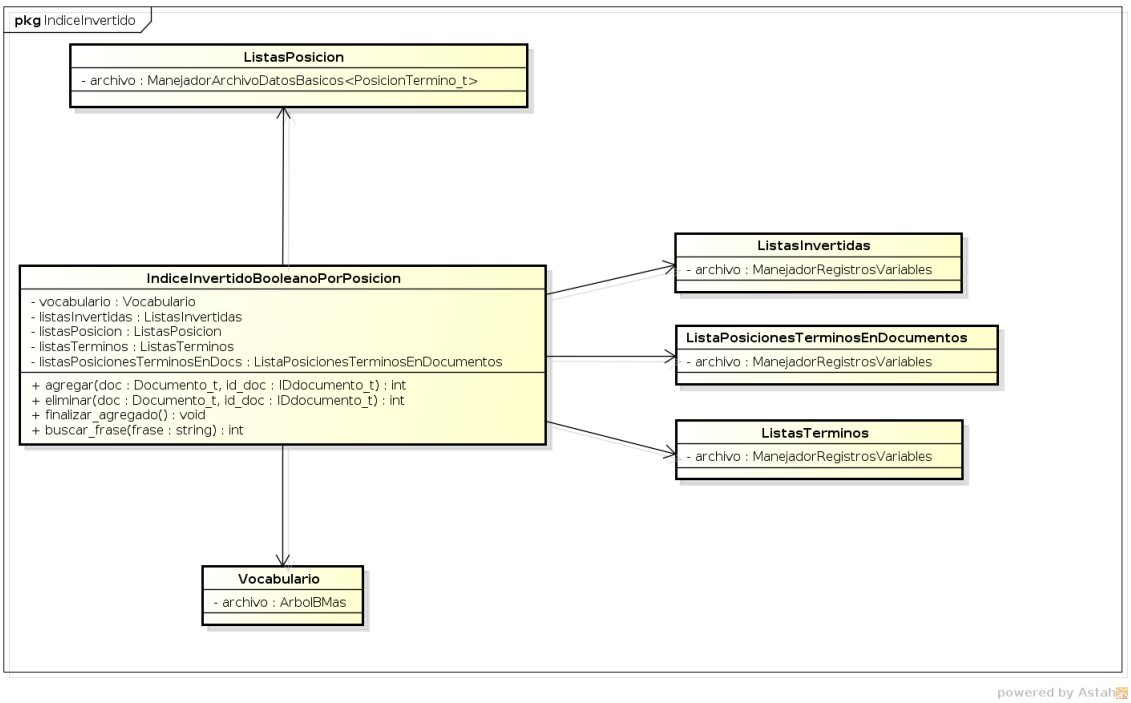

Figura 5: Diagrama de clases de Indice Invertido.

Las clases que son necesarias para implementar este índice invertido son:

▷ `ListaTerminos`: es un archivo secuencial de registros variables donde se guarda cada término distinto que se encuentra en los archivos a indexar. A cada término se lo puede identificar mediante su posición relativa en este archivo, este valor lo llamaremos `IDtermino`. Sus registros tienen la siguiente estructura:

`(término : string)`

▷ `Vocabulario`: es un archivo de tipo árbol B+ con el término completo como identificador del registro. Cada término tendrá almacenado una referencia a su propia `ListaInvertida`. Sus registros tienen la siguiente estructura:

`(i(término) : string, IDtermino : int, refListaInvertida : int)`

▷ `ListasInvertidas`: es un archivo que indica, para algún termino: en qué documento aparece, cuántas veces, y dónde está almacenada la lista que contiene las posiciones donde aparece ese término en ese documento. Sus registros tienen la siguiente estructura:

`(IDdocumento : int, cantidadPosiciones : int, refListaPosiciones : int)`

▷ `ListasPosiciones` : es un archivo que contiene nada más que números. Cada número es la posición de algun término en algún documento. Este archivo tiene la siguiente estructura:

`(posicion : int)`

▷ `ListaPosicionesTerminosEnDocumentos`: es un archivo que se va actualizando a medida que se indexan los documentos. Para cada término encontrado en cada documento, se registra el identificador del término, el identificador del documento, y la posición dentro del mismo donde aparece dicho término.

`(IDtermino : int, IDdocumento : int, posicion : int)`

El algoritmo de indexación consta de dos fases:

1. La fase de ***scanning***:

```
Por cada término relevante t encontrado en un documento idD
en la posición relativa dentro del documento p:
  Se busca t en el Vocabulario (archivo B+).
    Si está:
      Se obtiene su idT.
    Si no:
      Se registra t al final del ListaTerminos por orden de aparición,
        con su posición relativa determinando su idT.
      Se agrega (t, idT, REF_NULL) al vocabulario.
  Se registra (idT, idD, p) en el archivo de posiciones de términos en documentos.
```

2. La fase de ***indexing***:

```
Ordenar ListaPosicionesTerminosEnDocumentos por idT y idD.
Para cada registro en ListaPosicionesTerminosEnDocumentos:
  Para cada idT distinto:
    Para cada idD distinto:
      Guardar las posiciones en ListasPosiciones, obteniendo RefListaPos
      Guardar una nueva lista en ListasInvertidas con RefListaPos
    Actualizar la entrada del Vocabulario con la referencia a la lista invertida
```

Para encontrar los documentos que poseen una frase dada, se utiliza un algoritmo de intersección de conjuntos.

Para poder agregar canciones al índice, se realiza la fase de ***scanning*** sólo sobre los nuevos documentos, y luego se realiza completa la fase de ***indexing***.

Compresor

Este paquete del trabajo contiene las siguientes clases:

▷ `BufferBits` es una clase de tipo `template` que maneja la lectura, manipulación y emisión de bits. El propósito de esta clase es permitir la escritura de bits cuya cantidad no complete el octeto.

▷ `ModeloProbabilistico` representa el alfabeto con el que se está trabajando y la frecuencia de cada caracter dentro de ese alfabeto. Esto es un vector de frecuencias enteras, del cual se pueden obtener las probabilidades de cada símbolo.

- El tamaño del alfabeto puede ser de 257 (256 caracteres ASCII) u otro valor que tenga sentido para el usuario. En el caso particular del compresor PPMC, el modelo -1 utiliza un vector de 256 posiciones y el resto de los modelos un vector con 257 posiciones, para incluir el caracter especial "escape".

▷ `Intervalo` representa el segmento en el cual se está trabajando para comprimir. Contiene los atributos "piso", "techo" y "rango", los cuales se van modificando a medida que se comprime.

- El método `normalizar()` es el encargado de devolver un vector de bits que representa lo que se debería emitir en la compresión luego de resolver los underflow y overflow. También devuelve cuántos overflow y underflow se produjeron en una normalización. Estos dos datos se utilizan en otras clases.
- El método `actualizar_piso_techo()` recibe los valores de `low_count` y `high_count` de un símbolo (que representan la probabilidad de un símbolo), y actualiza el piso y el techo del intervalo de acuerdo a estos valores.

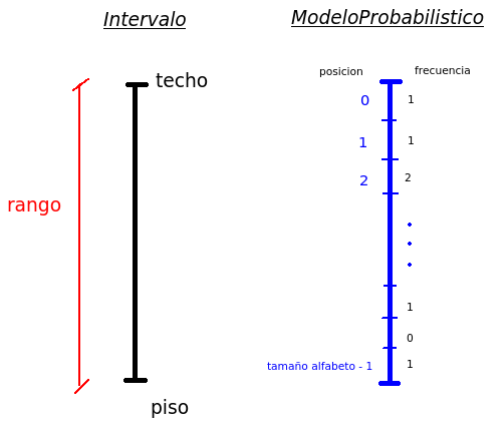

Figura 6: Relación entre `ModeloProbabilistico` e `Intervalo`.

▷ `Arimetico` implementa tanto el compresor como el descompresor aritmético dinámico de orden 0.

▷ `PPMC` implementa tanto el compresor como el descompresor, con un orden definido en 5 (que se puede modificar en el archivo `Constantes.h`[2]).

- Esta clase utiliza la clase `Orden`, que representa mediante un mapa a los `ModeloProbabilistico` contenidos en cada contexto del orden. El orden -1 tiene un único contexto, y el modelo 0 también. El modelo 1 tiene 257 objetos de tipo `ModeloProbabilistico`, el modelo 2 tiene 257^2, y así sucesivamente.

[2]Para cambiar esta constante es necesario recompilar el proyecto dentro del directorio `src`, ejecutando `make clean` y luego `make`.

- En la carpeta `ppmc_logs` se generarán archivos de log para cada ejecución, que indicarán, para cada compresión y descompresión, lo siguiente:
  - Caracter emitido,
  - Orden en el que el caracter es emitido,
  - Probabilidad con que el caracter es emitido.

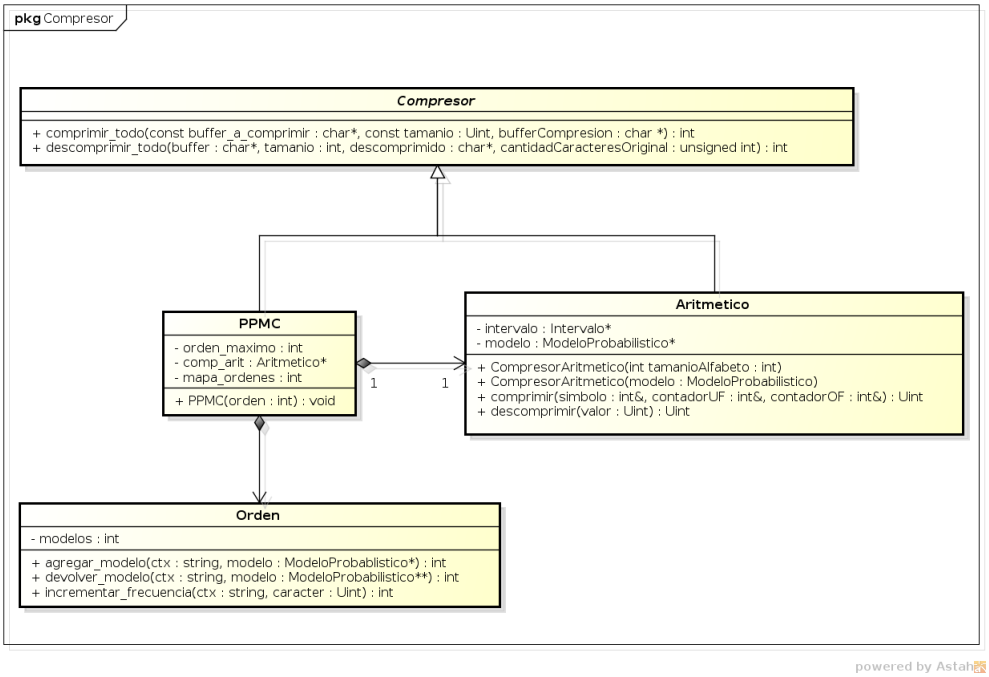

Figura 7: Diagrama de clases del compresor PPMC.

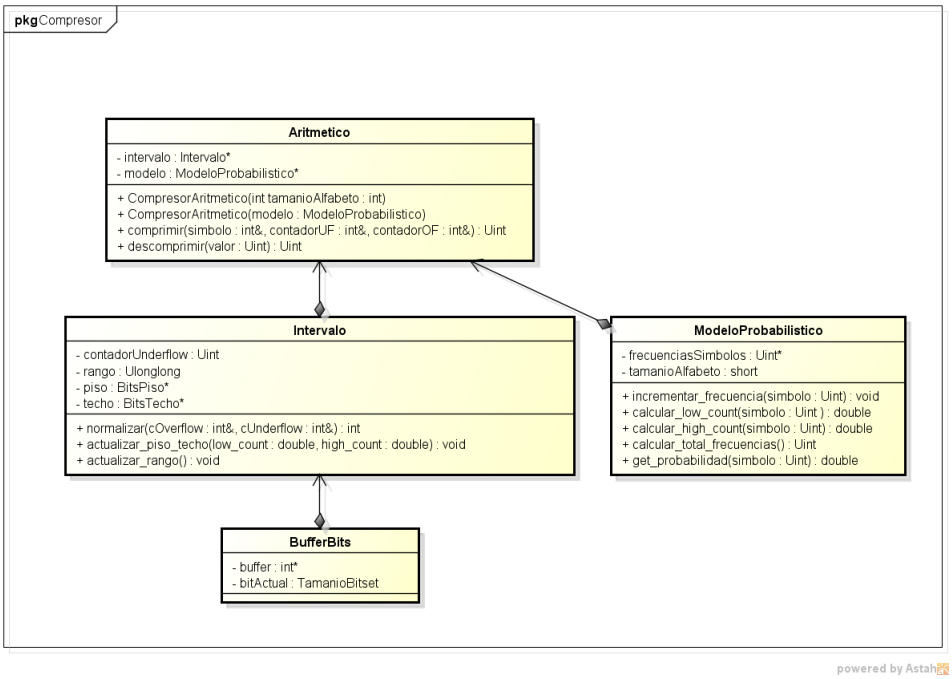

Figura 8: Diagrama de clases del compresor aritmético.

Parser

El paquete Parser contiene una única clase `ParserCanciones`. Esta clase es la encargada de extraer la información de los archivos y devolverlos en objetos de tipo `RegistroCancion`, para poder almacenarlos en el disco.

- ▷ El método `obtener_proxima_cancion()` devuelve dos registros: uno "normalizado" y otro "no normalizado". Esto es necesario ya que necesitamos el primero para crear los índices secundarios, pero necesitamos el segundo para almacenarlo en el archivo maestro.

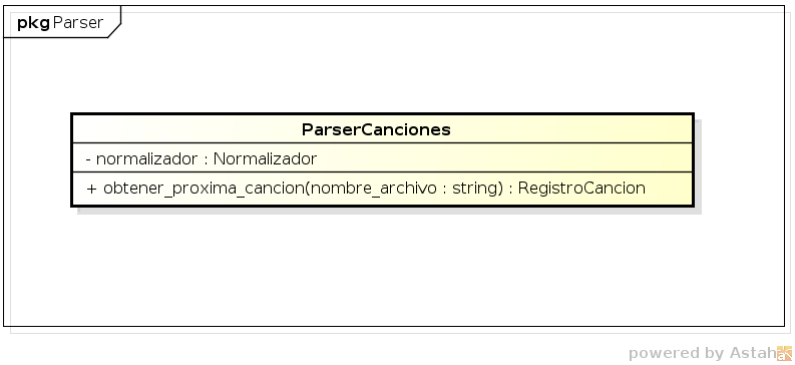

Figura 9: Clase ParserCanciones.

Capa Interfaz

El diagrama ilustra el diseño de los componentes de esta capa:

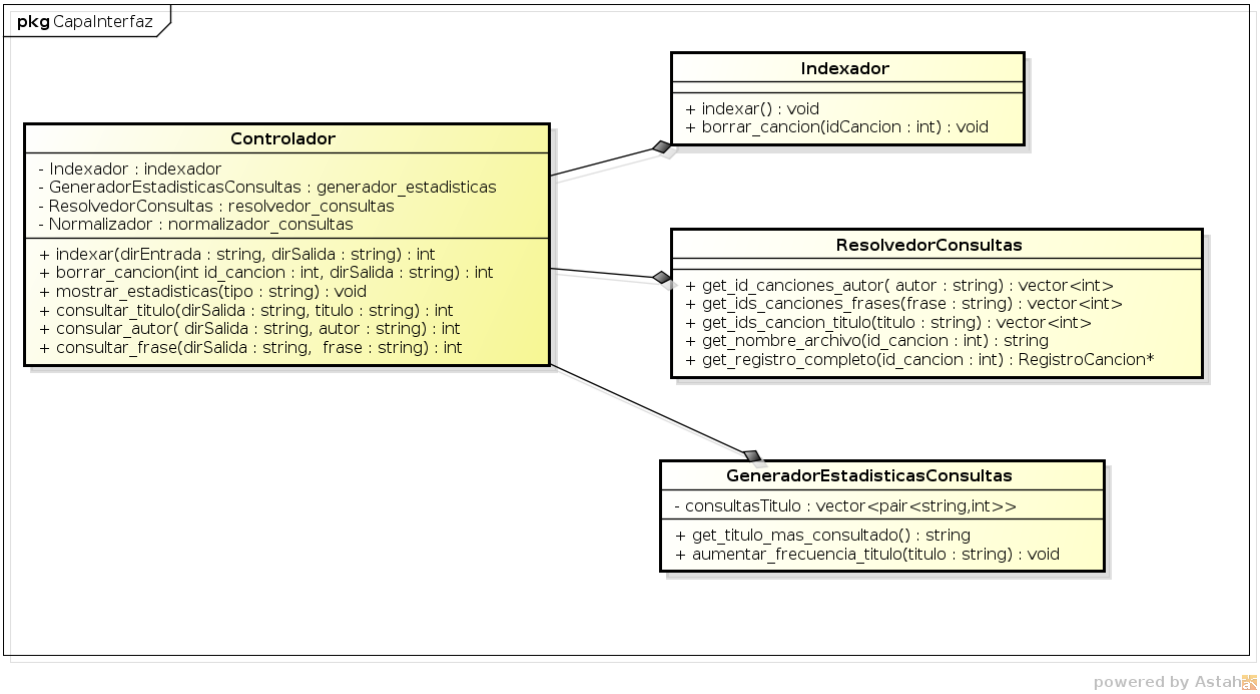

Figura 10: Diagrama de clases de la Interfaz

La clase `ResolvedorConsultas` es la encargada de recibir consultas y devolver datos asociados a esas consultas (IDs de canciones, Registros, nombres de archivos, etc.).

La clase `Indexador` es la encargada de modificar los archivos de las estructuras ante pedidos del usuario.

La clase `GeneradorEstadisticasConsultas` es la encargada de llevar cuenta de la estadística de las consultas de los usarios en una corrida del programa.

Las estructuras que se utilizan para mantener el índice son los siguientes:

Tipo de estructura	**Nombre de la estructura**	**Estructura de los registros**
`IndiceInvertido`	`indiceSecundarioFrases`	Explicada anteriormente.
`HashingExtensible`	`indicePrimario`	`(i(ID canción) , offset de canción en archivo maestro)`
`ManejadorRegistrosVariables`	`archivoMaestro`	`((autor)+, (anio)?, idioma, titulo, letra)`
`ArbolBMas`	`indiceSecundarioAutor`	`(i(autor canción + id canción))`
`IndicePorTitulo`	`indiceSecundarioTitulo`	`(i(titulo canción), (id canción)+)`
`HashingExtensible`	`documentos`	`(i(id canción), nombre archivo original)`

6 Manual de usuario

Pasos previos

Para poder indexar canciones en idioma español, se requiere que el usuario ejecute por consola el siguiente comando:

```
user@notebook:~/organizacion-datos-2013-grupo-2$ sudo
user@notebook:~/organizacion-datos-2013-grupo-2$ sh install_locale_es.sh
```

Compilación

La aplicación se compila dentro del directorio "`src`", ejecutando por consola el comando "`make`".

Ejecución

La aplicación se ejecuta por consola, dentro del directorio "`src`". de la siguiente forma:

```
./main
```

El programa posee las siguientes opciones de ejecución:

```
(0) SALIR.
(1) Indexar carpeta.
(2) Consultar título.
(3) Consultar autor.
(4) Consultar frase.
(5) Borrar canción por ID.
(6) Buscar titulo más consultado.
(7) Buscar autor más consultado.
(8) Buscar frase más consultado.
(9) Buscar tema más consultado.
```

Este programa no terminará la ejecución hasta que el usuario no escoja la opción de salir del mismo.

Ejemplos

- ▷ Para indexar la carpeta `songs` dentro de `organizacion-datos-2013-grupo 2` y guardar los índices dentro de `output`:

```
user@notebook:~/organizacion-datos-2013-grupo-2/src$ ./main
(Menú)
Ingrese operacion: 1
Ingrese directorio de entrada: ../songs
Ingrese directorio de salida: ../salida
AVISO: El directorio de salida no existe.
Se creó el directorio de salida.
Se indexó ../songs/joaquin sabina - y nos dieron las diez.txt correctamente!
No se indexó ../songs/at the drive in -invalid.txt porque no cumple el estándar especificado.
(...)
user@notebook:~/organizacion-datos-2013-grupo-2/src$
```

- ▷ Para realizar consultas por título:

```
user@notebook:~/organizacion-datos-2013-grupo-2/src$ ./main
(Menú)
Ingrese operacion: 2
Ingrese directorio de salida: ../salida
Ingrese consulta: helter skelter
-----------------------------------------------
- ID  = 3
- AUTOR/ES  = aerosmith,
- TITULO = helter skelter
(Letras)
-----------------------------------------------
-----------------------------------------------
- ID  = 5
- AUTOR/ES  = oasis,
 - TITULO = helter skelter
(Letras)
Ingrese operacion:
```

▷ Para realizar consultas por autor:

```
Ingrese operacion: 3
Ingrese directorio de salida: ../salida
Ingrese consulta: u2
-----------------------------------------------
- ID  = 8
- AUTOR/ES  = u2,
- TITULO = pride (in the name of love)
(Letras)
Ingrese operacion:
```

▷ Para saber el título que más fue consultado en una ejecución del programa:

```
Ingrese operacion: 6
Título más consultado: 'helter skelter'
Ingrese operacion:
```

Tests

Si se desea, se pueden ejecutar los tests dentro del directorio "tests", ejecutando el comando "make" por consola y luego ejecutando "./tests".