



**FACULTAD  
DE INGENIERIA**

Universidad de Buenos Aires

**Organización de Datos (75.06)**  
**Curso 01 - L. Argerich**

**Trabajo Práctico**  
**Indexador de Textos**  
**1ro/2013**

---

Amarillo, Emilio	88343
Arias, Damián	89952
Gattei, Ignacio	91664
Tarsia, Guido	91456

---

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Definiciones básicas</b>	<b>3</b>
2.1. Término ( $T$ )	3
2.2. docID ( $D$ )	3
2.3. d-gap	3
2.4. Frecuencia absoluta ( $F_T$ )	3
2.5. Frecuencia relativa ( $F_{D,T}$ )	3
2.6. Lista de posiciones	3
<b>3. Generación de términos</b>	<b>4</b>
3.1. No se guardarán términos de más de una palabra	4
3.2. Se guardarán todos los términos en letra minúscula	4
3.3. Casos particulares de signos de puntuación	4
3.4. Números	5
<b>4. Construcción del índice: primera parte</b>	<b>6</b>
4.1. Lectura del directorio y asignaciones de docIDs	6
4.2. Extracción de términos y creación del índice en memoria	6
4.3. Creación de archivos temporales	7
<b>5. Construcción del índice: segunda parte</b>	<b>8</b>
5.1. Inversión basada en merge	8
5.2. Operatoria	8
<b>6. Métodos de compresión</b>	<b>10</b>
6.1. Rice coding	10
6.2. Frame of reference (FOR)	11
6.3. PFOR ( <i>patched-FOR</i> )	12
6.4. LZW	12
<b>7. Estructuras utilizadas en los archivos</b>	<b>13</b>
7.1. Los archivos flacos	13
7.2. El índice invertido	13
<b>8. Haciendo consultas</b>	<b>15</b>
8.1. Puesta en marcha	15
8.2. Ejemplo de consulta	16
8.2.1. Normalización y búsqueda	16
8.2.2. Algoritmo de intersección	16
<b>9. XR: experimental</b>	<b>18</b>
9.1. Corrector ortográfico	18

# 1. Introducción

El objetivo de este Trabajo Práctico es realizar un programa que permita buscar frases en un conjunto de documentos (la colección).

Para ello, luego de investigar sobre el tema, concluimos que la estructura más popular para este tipo de aplicación es el índice invertido, que consiste de dos grandes componentes: el vocabulario de términos (por ejemplo, de palabras) de la colección, y una lista invertida, la cual es una estructura que contiene información sobre la ocurrencia (o posición) del término.

Para la búsqueda de frases en índices invertidos se han propuesto diferentes enfoques. La primera estrategia, propuesta en Williams et al. [7] (1999) sería guardar palabras contiguas, en lo que se llama un índice de próxima palabra (nextword index). En la frase «hoy hace calor» se guardarían «hoy hace» y «hace calor» acompañado del documento en donde aparecen. Este enfoque podría utilizar hasta aproximadamente el doble de espacio en disco que la colección, porque se guardan todas las palabras de los documentos en dúos que pueden tener muy poca repetición. Se crean índices, aunque de tamaño considerable, de rápido accionar para encontrar frases.

Otra estrategia, la que utilizaremos, es almacenar la posición de cada término en cada documento de forma ordenada, así pueden utilizarse distintos algoritmos de compresión. Para buscar una frase, primero se buscan todas las palabras que contiene y se obtienen las listas invertidas para cada una, extrayendo las posiciones y documentos en donde aparecen las mismas. Luego, aplicando un algoritmo de intersección, se filtran los documentos en donde las palabras efectivamente existen en el orden deseado.

Una última estrategia, bastante más sofisticada, sería la presentada por Williams et al. [8] (2004) en donde se hace una combinación de las anteriores (un índice invertido con las palabras menos comunes y un índice de próxima palabra con los dúos más comunes) pero agregando un índice con las frases más comunes. Esta aproximación acelera un 400 % el tiempo de búsqueda sobre nuestra propuesta, pero ocupa el doble de espacio en disco, según el mismo artículo. Como priorizamos espacio antes que tiempo, la idea es descartada.

## 2. Definiciones básicas

Para diagramar el trabajo primero estableceremos las siguientes definiciones básicas:

- Documento: conjunto de datos en formato estándar de texto (Unicode).
- Carpeta a indexar: es la que se le pasa al programa creado para que procese todos sus documentos, y permita hacer consultas sobre su contenido
- Colección: conjunto de todos los documentos que aparecen en la carpeta a indexar.
- Diccionario: conjunto de todos los términos.

### 2.1. Término ( $T$ )

Se considera término  $T$  a todo conjunto de caracteres alfanumérico sumados al apóstrofo (o comilla simple: ' ). Cada término es, o bien, una cantidad numérica desde 1 (una) hasta una cantidad no definida de cifras, o bien palabras tanto simples como compuestas. Se desarrolla sobre este tema en la Sección 3.

### 2.2. docID ( $D$ )

(*document identification* = identificador de documento)

Este es un número asignado unívocamente a cada documento de la carpeta a indexar. Para hacer este proceso, se listan recursivamente todos los documentos (que se suponen *indexables*, es decir, de texto plano) y se los ordena (el criterio será develado más adelante). Este orden otorga el docID de cada documento, que comienzan desde el 1 (uno).

### 2.3. d-gap

(*document-gap* = distancia entre documentos)

Considerando una lista de docIDs en orden ascendente, se puede almacenar la misma con el primer elemento seguido de las diferencias entre los elementos siguientes, los *d-gaps*. Se utiliza la definición dada en Witten et al. [9, p. 115].

Por ejemplo  $\langle 1, 2, 4, 5, 8 \rangle$  se puede transformar en  $\langle 1, 1, 2, 1, 3 \rangle$ .

### 2.4. Frecuencia absoluta ( $F_T$ )

Cantidad de veces que aparece un término  $T$  en toda la colección.

### 2.5. Frecuencia relativa ( $F_{D,T}$ )

Cantidad de veces que aparece un término  $T$  en un documento  $D$ .

### 2.6. Lista de posiciones

También llamada lista de ocurrencias. Es aquella que almacena las posiciones  $p_i$  en un documento  $D$  en donde aparece el término  $T$ . Una posición en un documento corresponde a la cantidad de términos hasta el término en consideración, empezando por el 1.

### 3. Generación de términos

Todos los términos estarán compuestos de una única palabra y se guardarán en el Diccionario con letras minúsculas. Se tomaron decisiones basándonos en las siguientes directivas.

#### 3.1. No se guardarán términos de más de una palabra

Aunque los índices tipo Nextword (próxima palabra) son capaces de otorgar velocidades de acceso superiores a los índices invertidos tradicionales, ocupan en promedio un 60 % del espacio de la colección, según mencionan Williams et al. [7] (1999).

Para este Trabajo Práctico, se construirá un índice preparado para buscar frases, pero a través de sus posiciones. Esto quiere decir que se tendrán en cuenta la separación entre palabras dentro de un documento. Por ejemplo un sustantivo propio de más de una palabra como «San Francisco» se separará en dos términos «San» y «Francisco». Al hacer la consulta el programa calculará la distancia entre las dos palabras: («San», posición  $i$ ) y («Francisco», posición  $i + 1$ ), con las cuales se podrá intersectar con las listas invertidas de cada una y filtrar los documentos en donde en aparecen juntas en la posición requerida.

#### 3.2. Se guardarán todos los términos en letra minúscula

En un texto, puede haber palabras que contengan mayúsculas en los siguientes casos:

- Mayúscula al comenzar la palabra: porque es un sustantivo propio
- Mayúscula al comenzar la palabra: porque está precedida por un punto «.»
- Mayúscula en medio de la palabra: porque es un código o abreviatura («IQJ653», «PhD»)

Esto sería un problema si quisiéramos guardar términos de más de una palabra como en el ejemplo anterior ya que se complicaría enormemente para identificar «San Francisco» en el siguiente escenario: «... buenos momentos. San Francisco es ...». No hay forma de saber si «San» empieza con mayúscula porque es un sustantivo propio o porque está al lado de un punto. Por suerte, esto no será un problema, ya que se guardará todo en minúsculas.

Que todo se guarde en minúsculas, también ayuda en la búsqueda, ya que el usuario del programa podría escribir de forma incorrecta sin poner las mayúsculas e igualmente encontrar lo que busca.

#### 3.3. Casos particulares de signos de puntuación

- Guiones: si se encuentra un guion (medio o bajo) se toma como si fuese un espacio. Si el guion está dividiendo 2 palabras, se guarda cada palabra por separado como un término. También se reemplazan por espacios las '@' y los '/'.
- Apóstrofo (o comilla simple): por estar los textos de la colección en idioma inglés, conviene contemplar algunos casos del uso del apóstrofo y como se los tratará, a través de los siguientes ejemplos:
  - «Grey's Anatomy»: en este caso, se guardarán los términos «grey's» y «anatomy» por separado.
  - «Isn't it?»: se guardará «isn't» y «it».

- «Baba O'Riley»: se guardará «baba» y «o'riley».
- Caracteres que se ignoran: todos los caracteres que no son alfanuméricos y «encierran» conjuntos de palabras como son ¿?¡!()[], comillas dobles, acentos graves y agudos. También se ignoran . : , ; \* ^ +- \$ # .

### 3.4. Números

Los números se guardarán como términos comunes y silvestres. Por ejemplo:

- «1000»  $\implies$  «1000»
- «03/04/2004»  $\implies$  «03», «04» y «2004»

## 4. Construcción del índice: primera parte

Se mostrará la creación del índice a través de un ejemplo: se quiere indexar la carpeta «vainilla/» la cual contiene 3 documentos.

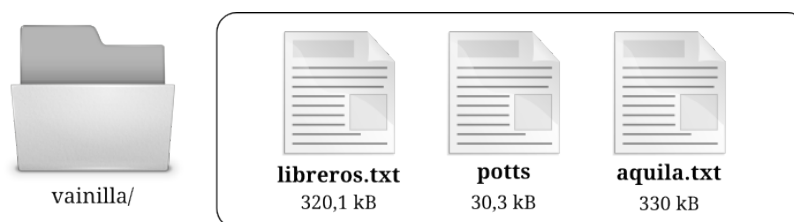


Figura 1: Ejemplo de directorio de trabajo

### 4.1. Lectura del directorio y asignaciones de docIDs

Luego de verificar que el directorio a indexar existe, se leen los nombres de **todos** los documentos del mismo (consideramos que el directorio es de *buena fe*), incluyendo los contenidos en carpetas interiores.

Para asignar docIDs, tomamos un consejo de Büttcher et al. [2, p. 216], (2010), pero no al pie de la letra. En la referencia, se ordenan los documentos según la cantidad de términos únicos que posea cada uno, porque cuántos más términos posea un documento, más probabilidad existe que los comparta con otro que también tiene muchos términos. Esto hace que las diferencias (*gaps*) entre documentos se achique a la hora de la compresión.

Seguir este enfoque para el TP demandaría potencialmente bastante más tiempo, por tener que procesar todos los documentos 2 veces. Por lo que nuestra propuesta es asignar docIDs según el tamaño del archivo, el archivo más pesado tiene docID=1, el siguiente, docID=2, y así hasta el último documento. No es lo mismo que la propuesta de [2], pero sigue el mismo lineamiento: los archivos más pesados *probablemente* compartan la mayoría, o por lo menos gran parte, de sus términos.

### 4.2. Extracción de términos y creación del índice en memoria

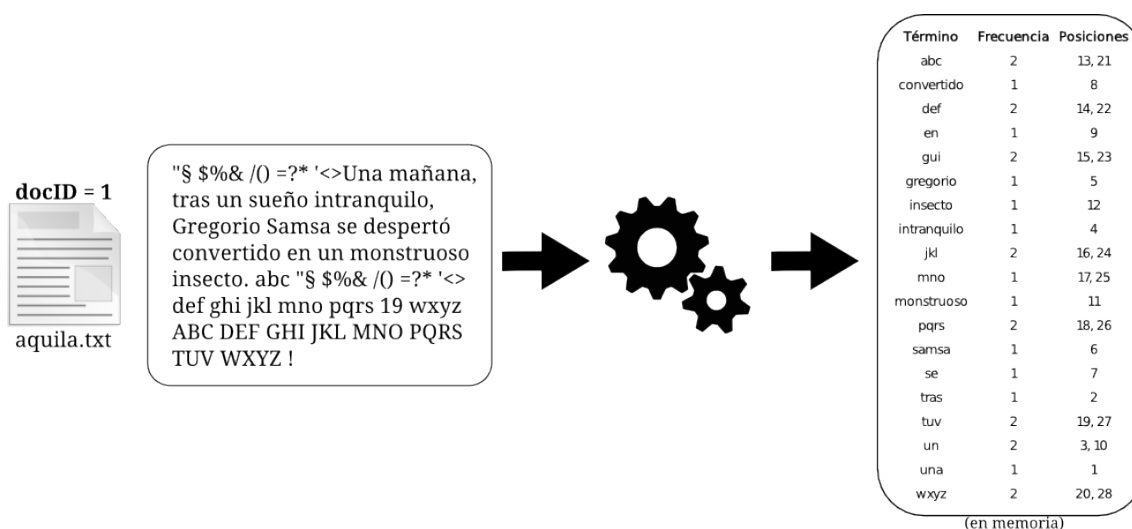


Figura 2: Un archivo procesado y cargados sus términos en memoria

Como mencionamos en la Sección 3 cada término es una palabra o cifra numérica. El programa crea listados en memoria principal con la siguiente información: para un término  $T_i$  se almacena que aparece en el documento con docID  $D_1$  en las posiciones  $p_1, p_2, p_3 \dots p_m$ , y listados equivalentes para los demás documentos. La estructura de estos listados es:

$$\{T_i; (D_1, \langle p_1, p_2, p_3 \dots p_m \rangle); (D_2, \langle p_1, \dots p_f \rangle); \dots; (D_j, \langle \dots \rangle)\}$$

Este procedimiento (generar términos y llevarlos a memorias principal) se hace constantemente hasta que ocurra uno de los siguientes hechos:

1. Se terminen de procesar todos los documentos
2. Se ocupe toda la memoria dedicada para el programa (se requerirán 512 MB de RAM dedicados, es decir, una PC con al menos 1 GB en total). En tal caso se crearán archivos temporales, se vaciará la memoria y se vuelve al paso 1.

### 4.3. Creación de archivos temporales

La primera vez que ocurra uno de estos dos momentos se creará un directorio temporal «vainillatemp/» que irá guardando archivos temporales con la información tal cual aparece en memoria. Sucesivas bajadas de memoria a disco, generarán archivos temporales de nombre incremental.

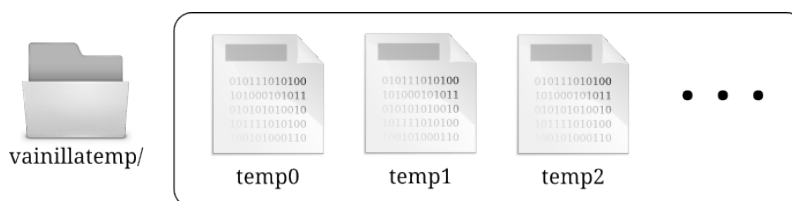



Figura 3: Un directorio temporal con archivos temporales

Cada archivo temporal contiene información suficiente para armar un índice por si mismo, ya que almacena para los documentos procesados, los términos, docIDs, frecuencias relativas y lista de posiciones como se muestra en la Figura 4.



Término	docID	Frecuencia	Posiciones
187	1	1	19
abc	1	2	13, 21
	2	2	68
	3	2	64, 192
en	1	1	9
...	..	...	..

Figura 4: Datos almacenados en un archivo temporal



## 5. Construcción del índice: segunda parte

### 5.1. Inversión basada en merge

Es inevitable citar a Zobel and Moffat [11, p. 14] (2006) y Witten et al. [9, p. 238] (1999), en donde encontramos que el método utilizado para llevar a cabo las operaciones descritas a continuación es un merge (mezcla) con la cantidad de documentos temporales que existan. Se muestra en la figura 5.

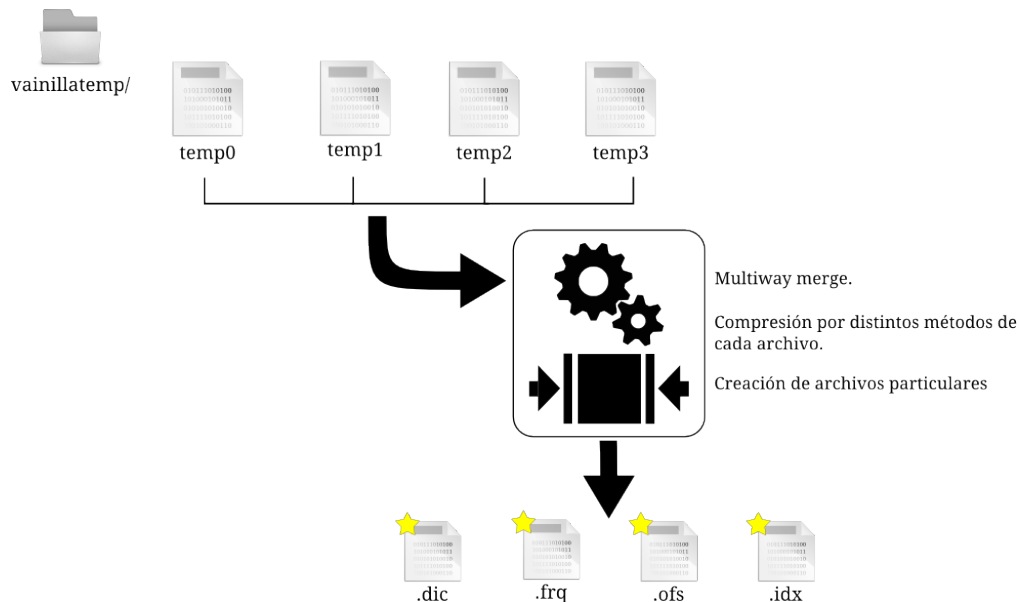


Figura 5: Multiway merge

### 5.2. Operatoria

La segunda parte de la creación del índice parte del mergeo de todos los archivos temporales, y resulta en la creación de 4 archivos: diccionario, frecuencias, offsets e índice, a partir de ahora, archivos dic., frq., ofs. e idx.

Al mergear registros del mismo término, se combinan las listas de docId's, frecuencias y posiciones de dicho término. Para cada término con sus datos obtenido, se realiza lo siguiente: -a

- En el archivo .dic se escribe el término codificado en LZW, avanza el puntero.
- En el archivo .frq se escribe la suma de frecuencias del término en Rice Coding, avanza el puntero.
- En el archivo índice, en la x posición del puntero, se escribe un bloque conteniendo dicho único término, y dentro de dicho bloque, cierta cantidad de fragmentos mayor o igual a 1. Cada fragmento va a guardar (por fragmento) cierta cantidad de posteos, siendo posteos lo siguiente: docIds que tengan el término (en forma de distancias), las frecuencias del término para doc en Rice Coding, y tantas listas de posiciones como docs codificadas en Pfor Delta.

Hay que tener en cuenta que en cada fragmento se guarda hasta una cierta cantidad de posteos, siendo esta cantidad una constante definida. Nuestra constante definida es 128. Quiere decir que si el término está en más de 128 docs, cuando quiera guardar los datos sobre las ocurrencias en el doc 129, va a generar un fragmento nuevo.

- Luego de escribir en el .idx, en el archivo .ofs se escribe la posición actual del puntero al archivo índice en Rice Coding, y se avanzan ambos punteros (.idx y .ofs)

Cabe destacar lo siguiente como corolario:

Sea  $t$  un término que se encuentra en al menos un documento de la colección a indexar:

$$\begin{aligned}
 .dic[x] &= t \\
 .frq[x] &= fg(t) \\
 .ofs[x] &= p \\
 .ind[p] &= bloque(t)
 \end{aligned}
 \tag{1}$$

Tener en cuenta que un bloque en .idx se guarda la siguiente información para un solo término: docs en los que se encuentra, frecuencias en cada doc., posiciones en cada doc.

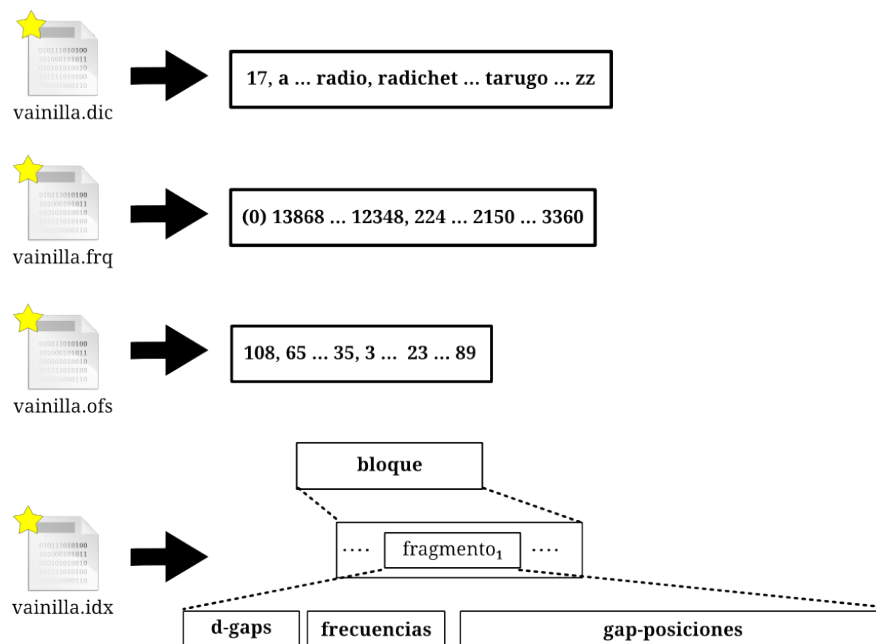


Figura 6: Archivos finales

## 6. Métodos de compresión

Para elegir métodos de compresión para listas de enteros se ha consultado:

- Trotman [6], (2003): se analizan los métodos Variable Byte («vByte»), Delta, Gamma y Golomb, extrayendo la conclusión que la tasa de compresión es exactamente inversa al listado de los nombres. Golomb obtiene un 50 % mayor compresión que vByte, mientras que este descomprime 7 veces más rápido que Golomb. Los otros métodos se ubican en el medio de estas cifras, sin alejarse demasiado de Golomb.
- Anh and Moffat [1], (2005): propone 3 algoritmos de rápida descompresión «Simple-9» (S9), «Relative-10» y «Carryover-12» y los compara con Golomb. Las conclusiones son que Golomb provee entre un 10 % y un 25 % más compresión que los propuestos, mientras que es hasta un 200 % más lento para recuperar información.
- Zhang et al. [10], (2008): analizan S9, vByte, PFor y Rice, y se extraen las conclusiones que Rice es el que mayor tasa de compresión ofrece, muy poco por encima de la compresión óptima, el método de PFor es alrededor de 4 veces rápido al descomprimir apenas ocupando un pequeño porcentaje de almacenamiento de más comparado a Rice.
- Büttcher et al. [2, p. 210], (2010): analizan Gamma, Delta, Golomb, Rice, vByte y S9. Como es el único que tiene a Golomb y Rice juntos, se extraen conclusiones importantes de aquí. Se puede observar Golomb tiene una ventaja insignificante en cuanto a tasa de compresión, pero es tiene una desventaja importante en cuanto a velocidad de descompresión. En datos duros, es como ganar un 2 % de espacio, pero perder un 30 % más de tiempo en descomprimir (la compresión demora lo mismo en ambos).

### 6.1. Rice coding

El primer elegido es Rice coding. Conocido hoy en día como un caso particular de Golomb coding, es uno de los métodos más antiguos de compresión, data del año 1979. Se trata de que un entero  $n$  es codificado en 2 partes: un cociente  $q = \lfloor \frac{n}{2^b} \rfloor$  y un resto  $r = n \bmod 2^b$ . El cociente es almacenado en forma unaria utilizando  $q + 1$  bits, mientras que el resto es almacenado en formato binario usando  $b$  bits.

La ventaja principal de este método es que tiene muy buena tasa de compresión. Sin embargo es en general un método lento en términos de velocidad de compresión y descompresión, siendo la causa principal que este método necesita manipular el código unario un bit a la vez, tanto en compresión, como en descompresión. Esto es muy demandante para el CPU, pero, por otro lado, como  $b$  es potencia de 2, la compresión y descompresión se hacen lo más eficientemente posible, necesitando operaciones bit a bit, como shifts y máscaras.

**b óptimo** Golomb comprime con un  $b_{opt} = 0,69 \times Promedio$  (promedio de todos los números de la lista a comprimir), y se necesitarán multiplicaciones y divisiones que enlentecen los procesos de compresión y descompresión. En teoría alcanza mejor tasa de compresión que nuestro elegido Rice, pero en la práctica la diferencia es muy pequeña [10], [2].

Para Rice coding, sabemos que  $b = 2^{\lfloor \log_2(b_{opt}) \rfloor}$  o  $b = 2^{\lceil \log_2(b_{opt}) \rceil}$ , y a través de la referencia [2, p. 200], encontramos que si llamamos  $N$  a la cantidad de documentos, tenemos que:

$$b_{opt} = \left\lceil \frac{\log_2(2 - \frac{F_T}{N})}{-\log_2(1 - \frac{F_T}{N})} \right\rceil$$

## 6.2. Frame of reference (FOR)

(Adaptado de Delbru et al. [3, p. 5] (2010))

El método FOR para una lista de enteros determina el rango de valores posibles, llamado «marco de referencia» y mapea cada valor adentro de este rango guardando suficientes bits para que los valores puedan ser distinguibles.

Por ejemplo, se tiene el siguiente listado (medianamente ordenado y con repetición):

⟨107, 108, 110, 115, 120, 125, 132, 132, 131, 135⟩

Si utilizáramos 8 bits para almacenar los cada uno de los 10 números, ocuparíamos 80 bits. En cambio, utilizando el punto de vista FOR, veríamos que el rango de número va de 107 a 135, por lo que podríamos restar 107 a todos los números de la secuencia, quedando ⟨0, 1, 3, 8, 13, 18, 25, 25, 24, 28⟩. Ahora cada diferencia puede codificarse con a lo sumo 5 bits. Por supuesto que debe guardarse el mínimo valor utilizando 8 bits, y también indicar con 3 bits más el hecho que solo son 5 bits por valor. En total se ocuparon  $(8 + 3 + 9 * 5 = 45)$  bits, ahorrando un 44 % con respecto a la opción sin compresión.

Hay un beneficio fundamental de este método relacionado con el guardado de grandes enteros: si quisiéramos el número 25312 y encontramos que el valor mínimo de la lista comprimida es 107 y se usan 5 bits para guardar los elementos, sabremos de entrada que esa lista no tiene el elemento que buscamos.

Dado un marco de referencia  $[0, max]$ , FOR necesita  $\lceil \log_2(max + 1) \rceil$  bits, llamados *bits del marco*, para comprimir cada entero en el listado.

La principal desventaja de FOR es que es sensible a valores atípicos en la lista de enteros. Por ejemplo, si un listado de 1024 enteros contiene 1023 números inferiores a 16 y un solo valor superior a 128, los *bits del marco* serán  $\lceil \log_2(128 + 1) \rceil = 8$ , desperdiciando 4 bits por cada otro valor.

Sin embargo la compresión y descompresión se pueden hacer muy rápidamente, utilizando operaciones de muy poco coste para el CPU. Se utilizan para ello algoritmos con operaciones lógicas bit a bit como AND o SHIFT, sin estructuras de selección que demoran más tiempo en ejecutarse. Esta es la clave de los algoritmos FOR. Se ve un ejemplo en la Tabla 1.

<pre>compress3 (int[] i, byte[] b) {b[0] = (i[0] &amp; 7)   ((i[1] &amp; 7) &lt;&lt; 3)   ((i[2] &amp; 3) &lt;&lt; 6); b[1] = ((i[2] &gt;&gt; 2) &amp; 1)   ((i[3] &amp; 7) &lt;&lt; 1)   ((i[4] &amp; 7) &lt;&lt; 4)   ((i[5] &amp; 1) &lt;&lt; 7); b[2] = ((i[5] &gt;&gt; 1) &amp; 3)   ((i[6] &amp; 7) &lt;&lt; 2)   ((i[7] &amp; 7) &lt;&lt; 5);} </pre>	<pre>decompress3(byte[] b, int[] i) {i[0] = (b[0] &amp; 7); i[1] = (b[0] &gt;&gt; 3) &amp; 7; i[2] = ((b[1] &amp; 1) &lt;&lt; 2)   (b[0] &gt;&gt; 6); i[3] = (b[1] &gt;&gt; 1) &amp; 7; i[4] = (b[1] &gt;&gt; 4) &amp; 7; i[5] = ((b[2] &amp; 3) &lt;&lt; 1)   (b[1] &gt;&gt; 7); i[6] = (b[2] &gt;&gt; 2) &amp; 7; i[7] = (b[2] &gt;&gt; 5) &amp; 7;} </pre>
Rutina de compresión que codifica 8 enteros usando 3 bits cada uno	Rutina de descompresión que decodifica 8 enteros representados por 3 bits cada uno

Cuadro 1: Compresión y descompresión FOR

Dado un listado de  $n$  enteros, FOR determina el marco de referencia y codifica el listado a través de pequeñas iteraciones de  $m$  enteros usando la misma rutina de compresión en cada

iteración. Por cuestiones de rendimiento,  $m$  es generalmente elegido múltiplo de 8 para que coincida con la frontera de los bytes.

**Listas delta** Como la de *d-gaps* presentada anteriormente. Debido a que la distribución de probabilidad generada tomando las diferencias tiende a ser monótonamente decreciente, una práctica común es elegir como marco de referencia  $[0, max]$ , en donde  $max$  es mayor valor entre los *deltas*.

### 6.3. PFOR (patched-FOR)

(Zukowski et al., 2006)

El segundo elegido. PFOR es una sencilla modificación al método FOR, con la cual se atenúa el problema de los valores atípicos mencionado anteriormente (aquí denominados *excepciones*).

Primero se determina el valor  $b$  para el cual la mayoría (digamos, el 80-90 %) de la lista de enteros a comprimir son menores a  $2^b$ . Entonces el listado comprimido se guarda en 2 partes: en la primera entran todos los enteros menores a  $2^b$  en  $b$  bits cada uno, y en la segunda se guardan las *excepciones*, con 8, 16 o 32 bits cada una, dependiendo la excepción más grande.

Al igual que FOR es rápido en la compresión y descompresión, salvo por un pequeño número de elementos, las excepciones. Pero tiene mayor tasa de compresión al no desperdiciar tantos bits en listados en donde los valores son en promedio uniformes salvo por algunos particulares.

**PFOR-Delta** Este método, modificado para *listas delta*, es llamado en la bibliografía ([10] y [12]), PFOR-Delta.

### 6.4. LZW

Como se verá más adelante, nuestra propuesta para resolver el TP necesita un archivo con palabras guardadas, el Diccionario.

Para comprimir dicho archivo utilizaremos unos de los métodos clásicos y más simples de compresión, el LZW. Básicamente lo que hace este método es leer una secuencia de símbolos, agruparlos en cadenas de caracteres y convertir las mismas en códigos. Como los códigos ocupan menos lugar que las cadenas de caracteres, se obtiene compresión.

No se hace ningún análisis del texto. En su lugar, solamente agrega cada nueva cadena de caracteres que encuentra a una tabla de que contiene todas las anteriores.

LZW es un algoritmo «voraz» (*greedy*) porque trata de encontrar la cadena de caracteres más larga posible para la cual tiene código asignado.

## 7. Estructuras utilizadas en los archivos

Como mencionamos en la Sección 5, para guardar toda la información referente a la colección de documentos, utilizaremos 4 archivos.

Combinando los enfoques de Zhang et al. [10], (2008) y de Büttcher et al. [2, p. 107], (2010), llegamos a las estructuras de archivos que describiremos.

### 7.1. Los archivos flacos

Estos archivos solamente almacenan un tipo de dato muy específico:

- **\*.dic**: aquí se guardan todos los términos que se encontraron en la colección, una larga cadena con los mismos, separados con barra «/» o espacio. Se los comprime con LZW, en bloques o bien se testeará con todo el archivo.
- **\*.frq**: correspondientemente a cada término existe una frecuencia absoluta asociada, que se guarda en este archivo. Lógicamente son enteros sin orden distribuidos aleatoriamente. Se utilizará Rice coding, que es que mejor comprime, para conjuntos de frecuencias. Este archivo tiene la finalidad de ayudar a hacer las búsquedas más rápidas, ya que se irá filtrando por la palabra menos frecuente.
- **\*.ofs**: además se necesita guardar el offset (distancia) que existe para cierto término en el índice invertido. Como son enteros crecientes que se van a cargar en memoria principal, se utilizan deltas. Hay 2 opciones, o se comprime aplicando Rice coding o PFOR-Delta, separando en conjuntos de offsets, o se deja sin comprimir. Analizaremos si vale la pena el espacio se pierde con la última opción.

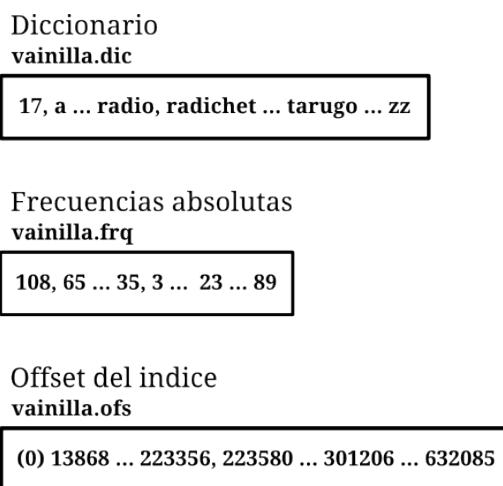


Figura 7: Los archivos flacos

### 7.2. El índice invertido

El llamado índice invertido de posiciones sigue con total naturalidad lo propuesto por [10], pero sin bloques de tamaño de fijo, si no variables (en la frontera de los bytes).

La estructura general de nuestro índice invertido se muestra en la Figura 8. Se crean particiones en el archivo llamadas *bloques*, de tamaño variable.

Definimos un «posteo» como el conjunto de un docID, frecuencia relativa y lista de posiciones para un término (lógicamente, en un documento).

Índice invertido  
vainilla.idx

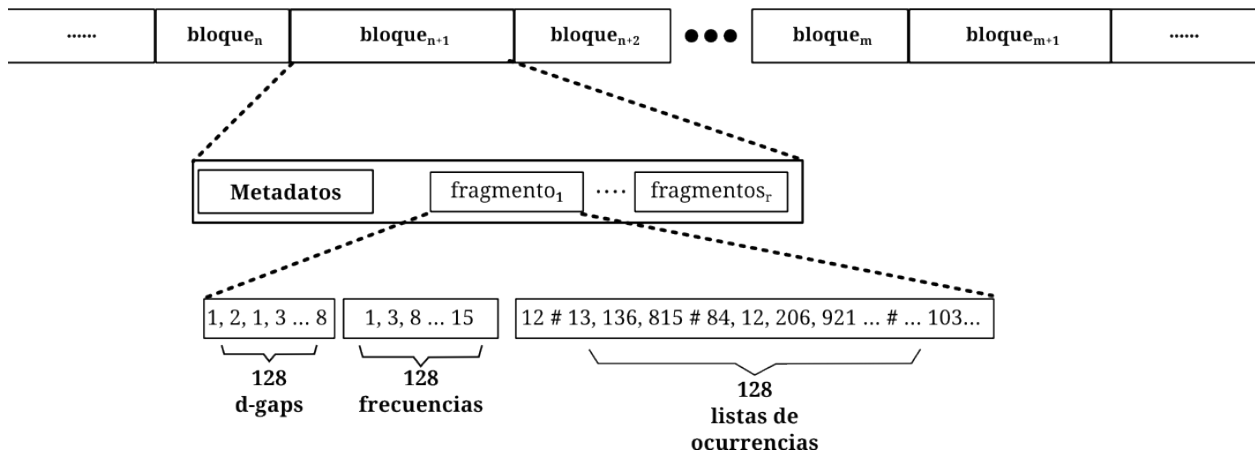


Figura 8: Estructura del índice invertido

## Un bloque

- Guarda información correspondiente a un término de la colección.
- Posee un gran número de listas invertidas correspondientes a las posiciones del término en consideración. Siguiendo con la referencia, dividimos estas listas en fragmentos con 128 posteos cada una. Este número podría cambiar para hacer más rápida la búsqueda en colecciones chicas o más grandes, pero siempre será múltiplo de 32.
- Contiene metadatos (datos adicionales) con información de cuántos fragmentos posee y dónde comienzan (offset desde el primer fragmento), y a continuación los fragmentos. Esto se hace porque, como la estructura de datos permite decodificar los docIDs primero que lo demás, se quiere ser capaz de saltar al siguiente fragmento sin pasar por todos los datos comprimidos. No se decidió si comprimir los metadatos aún.

## Un fragmento

- Es una unidad básica para descomprimir datos en el índice invertido. Cada fragmento se puede descomprimir muy rápidamente.
- Contiene 128 posteos almacenados como: 128 docIDs (almacenados como *d-gaps*), luego 128 frecuencias relativas correspondientes y al final 128 listas de ocurrencias (guardadas como *gaps* para cada docID). Los métodos de compresión serán, respectivamente, PFOR-Delta, Rice coding y PFOR-Delta nuevamente (a prueba, si no Rice coding).
- Se lo puede «saltar» si el docID que buscamos no se encuentra, por ello este listado se comprime con PFOR-Delta (que guarda la el rango máximo y el primer docID).

## 8. Haciendo consultas

En esta sección se ilustrará cómo funciona el algoritmo que utiliza el programa al momento de resolver una consulta.

### 8.1. Puesta en marcha

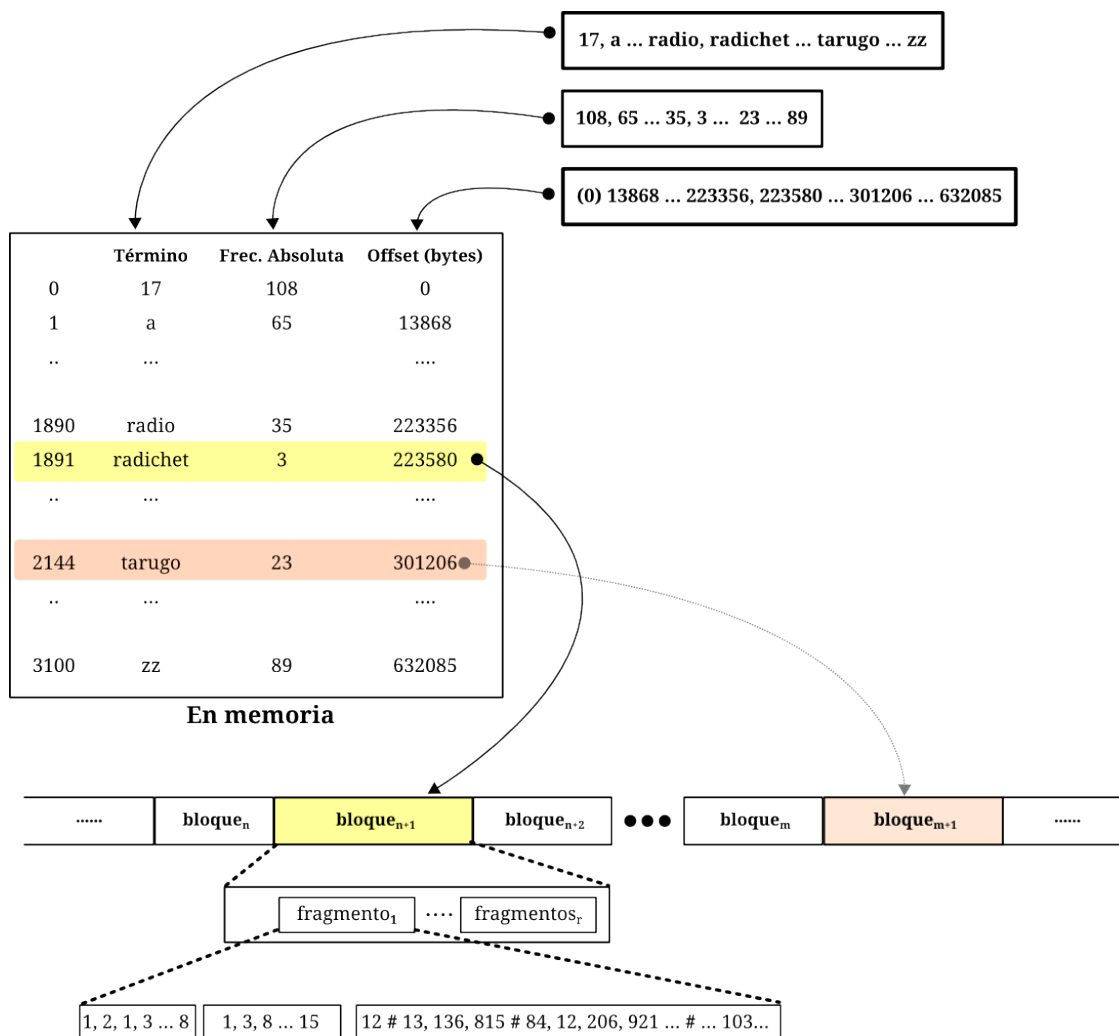


Figura 9: Armado en memoria de la tabla de términos y búsqueda en el índice invertido

Ni bien se ejecuta el programa, éste se prepara para resolver la consulta armando para ello una tabla en memoria. Esta tabla se compondrá de tres columnas: Término, Frecuencia Absoluta y Offset.

Cada columna se extrae descomprimiendo desde disco los archivos .dic, .frq y .ofs respectivamente. A primera vista, pareciera descabellado cargar el contenido de estos 3 archivos en memoria, pero haciendo algunas cuentas se ve que el espacio ocupado no es intimidante:

Se estima que para una colección de 1 millón de documentos habrá aproximadamente 500000 términos distintos. Si consideramos que en promedio, los términos ocupan 6 bytes aproximadamente. Si además le sumamos a esto 2 bytes por término (contemplando su frecuencia absoluta y su offset), se tiene que se requerirán 8 bytes por término. Haciendo la cuenta:  $500000 \times 8 = 4$  millones de Bytes  $\approx 3906$  KBytes  $\approx$  **3.8 MBytes**.



**Nota** Lo ideal sería cargar esta tabla en memoria por única vez y luego realizar todas las consultas que se deseen. Pero por restricciones del enunciado del presente trabajo práctico, se cargará la tabla de nuevo para cada consulta.

## 8.2. Ejemplo de consulta

### 8.2.1. Normalización y búsqueda

Supongamos por ejemplo que el usuario ingresó la siguiente frase: «Radichet, tarugo»

El primer paso será normalizar la consulta pasando todas las palabras a letra minúscula y eliminando los caracteres que el sistema ignora como son los signos de puntuación. La consulta quedará entonces así: «radichet tarugo». A partir de la consulta normalizada se obtienen los términos «radichet» y «tarugo». A continuación se los ubica en la tabla generada (ver Figura 9) y se obtienen sus frecuencias absolutas y offset.

Con el offset, se ubican los bloques que corresponden a cada término en el índice invertido (archivo .idx) y se los descomprime obteniendo los documentos donde aparecen y las posiciones dentro de los mismos como se muestra en la Figura 10.

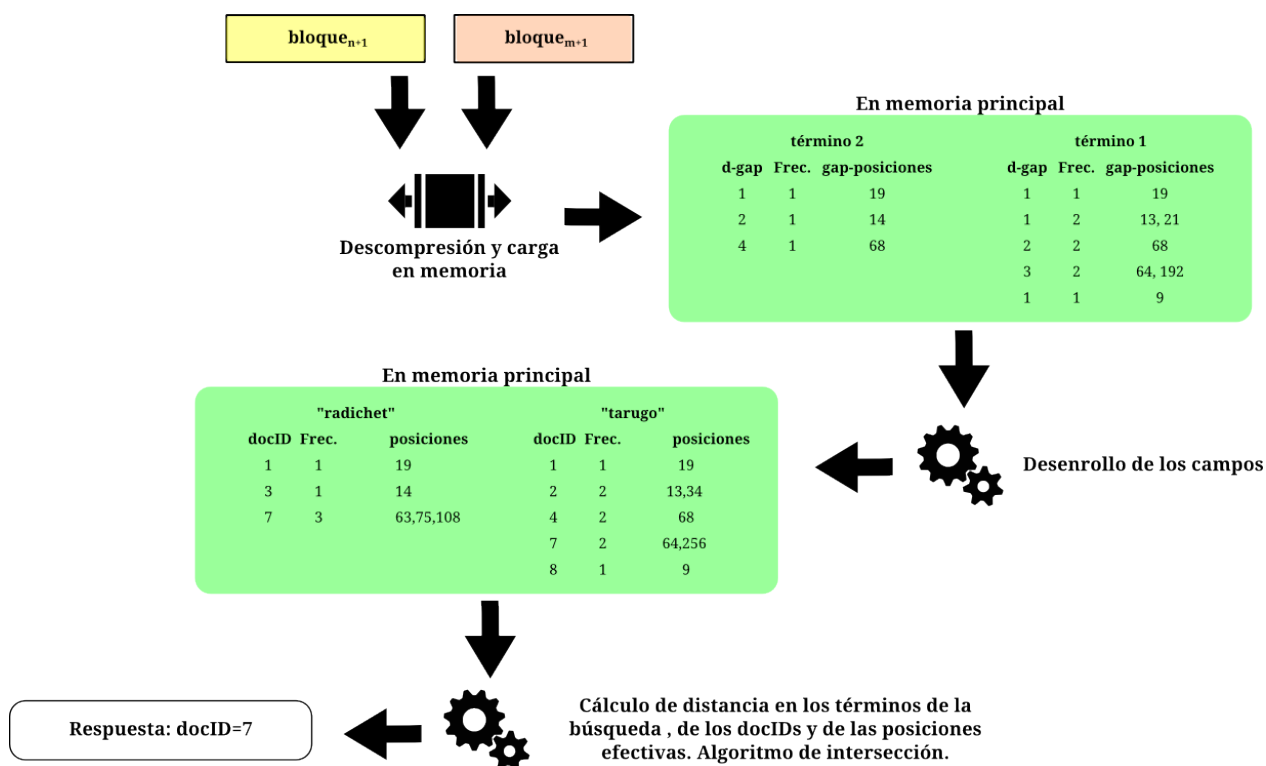


Figura 10: Obtención de las posiciones a partir de la descompresión de bloques

### 8.2.2. Algoritmo de intersección

La consulta de frases implica que la consulta es de tipo booleana, por lo que se la puede ver de la siguiente forma:

<< radichet >> AND << tarugo >>

Los documentos entregados como resultado de esta consulta serán:

1. los que contengan ambos términos y

2. los que tengan a «radichet» en la posición  $i$  y a «tarugo» en  $i+1$ .

Para resolver el primer punto se utiliza la frecuencia absoluta de los términos (que es la cantidad de documentos en los que aparece cada uno). En el ejemplo, el término «radichet» tiene frecuencia 3 y «tarugo» 23 por lo que, la máxima cantidad de documentos que podrán cumplir con los requisitos, serán 3. Finalmente se ve que solo 2 documentos (el 1 y el 7) contienen a ambos términos).

El segundo punto se resuelve recorriendo la lista de posiciones de los documentos 1 y 7 para cada término en simultaneo.

Finalmente, los documentos que cumplen con los dos puntos son presentados como resultado. En el ejemplo solo el documento de docID = 7 cumple con los requisitos («radichet» en posición 63 y «tarugo» en 64).

## 9. XR: experimental

### 9.1. Corrector ortográfico

La idea detrás de un corrector ortográfico es poder darle al usuario respuestas a sus consultas que pueden ser igual o más relevantes que lo que él buscó inicialmente.

Lo que haríamos, aprovechando que tenemos el Diccionario y sus frecuencias absolutas en memoria principal, es hacer las comparaciones utilizando las distancias de edición o de Levenshtein (Manning et al. [4, p. 58]), para cada palabra de la frase buscada y retornar una frase (o varias) con las más frecuentes, para darla como opción de búsqueda.

Utilizaremos para ello el algoritmo de Norvig, el cual tiene, hasta hoy en día, el balance perfecto entre eficiencia y sencillez.

## Referencias

- [1] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, January 2005. ISSN 1386-4564. doi: 10.1023/B:INRT.0000048490.99518.5c. URL <http://dx.doi.org/10.1023/B:INRT.0000048490.99518.5c>.
- [2] Stefan Büttcher, Charles Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. Mit Press, Cambridge, Mass., 2010. ISBN 9780262026512 0262026511. URL <http://books.google.de/books?id=UNNMQwAACAAJ>.
- [3] Renaud Delbru, Stephane Campinas, Krystian Samp, and Giovanni Tummarello. Adaptive frame of reference for compressing inverted lists, 2010.
- [4] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715. URL <http://nlp.stanford.edu/IR-book/>.
- [5] Peter Norvig. How to write a spelling corrector, 2007. URL <http://norvig.com/spell-correct.html>.
- [6] Andrew Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, January 2003. ISSN 1386-4564. doi: 10.1023/A:1022949613039. URL <http://dx.doi.org/10.1023/A:1022949613039>.
- [7] Hugh E. Williams, Justin Zobel, and Phil Anderson. What's next? - index structures for efficient phrase querying. In *Proc. Australasian Database Conference*, pages 141–152, 1999.
- [8] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, October 2004. ISSN 1046-8188. doi: 10.1145/1028099.1028102. URL <http://doi.acm.org/10.1145/1028099.1028102>.
- [9] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, CA, 2 edition, 1999. ISBN 978-1-55860-570-1.
- [10] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web, WWW '08*, pages 387–396, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. doi: 10.1145/1367497.1367550. URL <http://doi.acm.org/10.1145/1367497.1367550>.
- [11] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38:2006, 2006.
- [12] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE '06*, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. doi: 10.1109/ICDE.2006.150. URL <http://dx.doi.org/10.1109/ICDE.2006.150>.