

# TEMA 7. CONSTRUCCIÓN Y COMPRESIÓN DE ÍNDICES

---

Contenidos basados en el material del curso de Manning



# Contenidos

## 1. Construcción de índices

1.1 Introducción

1.2 La colección de documentos

1.3 Indexación basada en ordenación de bloques

1.4. Indexación en memoria de un paso

1.5. Indexación distribuida

1.6. Indexación dinámica

## 2. Compresión de índices

2.1 Introducción

2.2 Compresión de diccionarios

2.3 Compresión de los ficheros de postings

2.4 Compresión de texto

# Bibliografía

## ***A Introduction to Information Retrieval:***

*Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze.*  
*Cambridge University Press, 2009.*

***Capítulos 4 y 5***

## ***Managing Gigabytes: Compressing and Indexing Documents and Images***

*Ian H. Witten, Alistair Moffat and Tomothy C. Bell*  
*Morgan Kaufmann Publishers 1999*

***Capítulo 2***



# CONSTRUCCIÓN DE ÍNDICES

---

- 1.1 Introducción.
- 1.2 La colección de documentos.
- 1.3 Indexación basada en ordenación de bloques.
- 1.4. Indexación en memoria de un paso.
- 1.5. Indexación distribuida.
- 1.6. Indexación dinámica

# 1. Introducción

- Cuando se construye un sistema de Recuperación de Información muchas de las decisiones a tomar dependen del hardware sobre el que va a residir el sistema.
  - El acceso a los datos en memoria es mucho más rápido que el acceso a datos en disco.
  - No se transfieren datos del disco mientras el cabezal del disco está siendo posicionado.
  - La I/O a disco está basada en bloques: se leen o escriben bloques completos.
  - La transferencia de un segmento largo de datos del disco a memoria es más rápido que la de segmentos cortos no contiguos.
- Los servidores usados en sistemas de Recuperación de Información actualmente disponen de GBs de memoria.
- El espacio en disco suele ser varios (2–3) órdenes de magnitud mayor.



## 2. La colección de documentos RCV1

- Como ejemplo para aplicar la construcción de índices vamos a utilizar la colección de documentos Reuters RCV1, 1GB de texto aproximadamente.
- Consiste en un año de la agencia de noticias Reuters (parte de 1995 y 1996).
- Se toma como documento solamente la parte de texto de las noticias, ignorando la información multimedia.
- Se cubre un amplio espectro de temas internacionales: política, economía, deportes, ciencia, etc.

## 2. La colección de documentos RCV1



You are here: [Home](#) > [News](#) > [Science](#) > [Article](#)

Go to a Section: [U.S.](#) [International](#) [Business](#) [Markets](#) [Politics](#) [Entertainment](#) [Technology](#) [Sports](#) [Oddly Enough](#)

### Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

[Email This Article](#) | [Print This Article](#) | [Reprints](#)

[\[-\]](#) Text [\[+\]](#)



SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.



## 2. La colección de documentos RCV1

■ symbol	statistic	value
■ N	documents	800,000
■ L	avg. # tokens per doc	200
■ M	terms (= word types)	400,000
■	avg. # bytes per token (incl. spaces/punct.)	6
■	avg. # bytes per token (without spaces/punct.)	4.5
■	avg. # bytes per term	7.5
■ T	non-positional postings	100,000,000





### 3. Indexación basada en ordenación de bloques

**Recordemos**, en la construcción de un índice:

1. Los documentos son analizados para extraer pares (término, docID).

Doc 1

I did enact Julius  
Caesar I was killed  
i' the Capitol;  
Brutus killed me.

Doc 2

So let it be with  
Caesar. The noble  
Brutus hath told you  
Caesar was ambitious



Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



### 3. Indexación basada en ordenación de bloques

2. Después de analizar todos los documentos, el fichero invertido se ordena tomando como clave primaria los términos y como secundaria los docID.

100M ítems para ordenar.

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



### 3. Indexación basada en ordenación de bloques

2. Después de analizar todos los documentos, el fichero invertido se ordena tomando como clave primaria los términos y como secundaria los docID.

100M ítems para ordenar.

Term	Doc #	term	doc. freq.	→	postings lists
ambitious	2	ambitious	1	→	2
be	2	be	1	→	2
brutus	1	brutus	2	→	1 → 2
brutus	2				
capitol	1	capitol	1	→	1
caesar	1	caesar	2	→	1 → 2
caesar	2	did	1	→	1
caesar	2	enact	1	→	1
did	1	hath	1	→	2
enact	1	i	1	→	1
hath	1	i'	1	→	1
l	1	it	1	→	2
l	1	julius	1	→	1
i'	1	killed	1	→	1
it	2	let	1	→	2
julius	1	me	1	→	1
killed	1	noble	1	→	2
killed	1	so	1	→	2
let	2	the	2	→	1 → 2
me	1	told	1	→	2
noble	2	you	1	→	2
so	2	was	2	→	1 → 2
the	1	with	1	→	2
the	2				
told	2				
you	2				
was	1				
was	2				
with	2				



### 3. Indexación basada en ordenación de bloques

- Para colecciones pequeñas este proceso puede realizarse en memoria principal.
- Para grandes colecciones de documentos se requiere el uso de memoria secundaria.
- Para la ordenación de los elementos del índice cuando se requiere almacenar los datos en memoria secundaria se utilizan *algoritmos de indexación externa*.

#### OBJETIVO:

Minimizar el número de búsquedas/acceso a disco durante la ordenación



### 3. Indexación basada en ordenación de bloques: Algoritmo BSBI

- Segmenta la colección de documentos en bloques de la misma talla.
- Ordena los pares termID-docID de cada bloque en memoria principal.
- Almacena resultados intermedios en disco.
- Mezcla todos los resultados intermedios en un índice final.

8-byte (4+4) para (*termID*, *docID*).

100M elementos de 8-byte.

Se definen bloques de ~ 10M elementos:

Se pueden almacenar en memoria.

10 bloques para la colección RCV1

*BSBI blocked sort-based indexing*



### 3. Indexación basada en ordenación de bloques: Algoritmo BSBI

BSBIINDEXCONSTRUCTION()

```
1   $n \leftarrow 0$   
2  while (all documents have not been processed)  
3  do  $n \leftarrow n + 1$   
4       $block \leftarrow \text{PARSENEXTBLOCK}()$   
5      BSBI-INVERT( $block$ )  
6      WRITEBLOCKTODISK( $block, f_n$ )  
7  MERGEBLOCKS( $f_1, \dots, f_n; f_{\text{merged}}$ )
```



# 3. Indexación basada en ordenación de bloques:

## Algoritmo BSBI

- El algoritmo analiza documentos para identificar los pares termID-docID y los acumula en memoria hasta que se llena un bloque (PARSENEXTBLOCK).
- El bloque es invertido y almacenado en disco (BSBI-INVERT):
  - Ordenación de los pares termID-docID.
  - Detección de todos los pares con el mismo termID y construcción de su postings list.
  - El índice invertido para el bloque es almacenado en disco.
- Mezcla todos los resultados intermedios en un índice final (MERGEBLOCKS).

# 3. Indexación basada en ordenación de bloques:

## Algoritmo BSBI

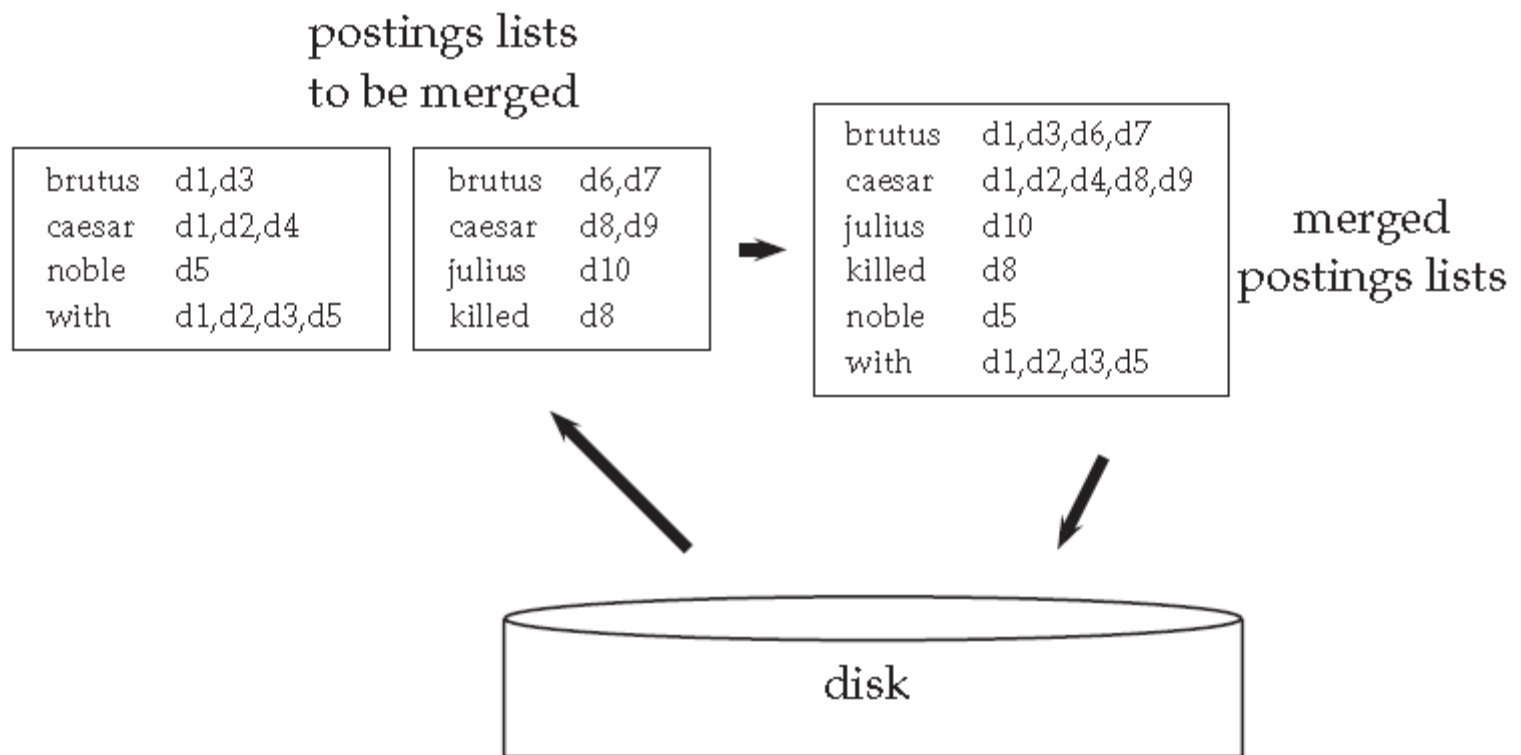
### MERGEBLOCKS:

- Se abren todos los ficheros de bloques simultáneamente.
- Se mantienen buffers pequeños de lectura para todos los bloques que se leen.
- Se mantiene un buffer de escritura para el índice final.
- En cada iteración:
  - Se selecciona el menor termID que no ha sido procesado todavía.
  - Todas las postings lists para este termID son leídas y mezcladas.
  - La lista resultante se escribe en disco.
  - Los buffers de lectura se rellenan desde disco cuando es necesario.





### 3. Indexación basada en ordenación de bloques: Algoritmo BSBI, MERGEBLOCKS





### 3. Indexación basada en ordenación de bloques: Algoritmo BSBI, costes

- Su complejidad temporal es  $\Theta(T \log T)$  ya que el paso más costoso es la ordenación y  $T$  es la cota superior de ítems a ordenar ( $T$  es el número de pares termID-docID).
- Sin embargo, el coste temporal real viene dominado por el coste del análisis de los documentos PARSENEXTBLOCK y de la mezcla final MERGEBLOCKS.
- Representa los términos a través de su termID (un número único de una serie)
  - Necesita una estructura de datos para representar la correspondencia entre términos y termID.
- Para colecciones grandes esta estructura de datos no cabe en memoria.



## 4. Indexación en memoria de un paso: Algoritmo SPIMI

- Utiliza términos en lugar de termID.
- Genera índices invertidos separados para cada bloque en disco
- No ordena, acumula los postings en las postings lists conforme van apareciendo.
- Empieza un nuevo índice para el bloque siguiente.
- Genera un índice invertido completo para cada bloque.
- Estos índices separados se mezclan en un gran índice.

*SPIMI single-pass in-memory indexing*



## 4. Indexación en memoria de un paso: Algoritmo SPIMI

```
SPIMI-INVERT(token_stream)
1  output_file = NEWFILE()
2  dictionary = NEWHASH()
3  while (free memory available)
4  do token  $\leftarrow$  next(token_stream)
5      if term(token)  $\notin$  dictionary
6          then postings_list = ADDTODICTIONARY(dictionary, term(token))
7          else postings_list = GETPOSTINGSLIST(dictionary, term(token))
8          if full(postings_list)
9              then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10         ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms  $\leftarrow$  SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

## 4. Indexación en memoria de un paso:

### Algoritmo SPIMI

- La parte del algoritmo que analiza los documentos y devuelve una secuencia de pares term-docID se ha omitido. SPIMI-INVERT es llamado repetidamente hasta que se ha procesado completamente la colección.
- Los tokens son procesados uno a uno (línea 4).
- Cuando un término aparece por primera vez, se añade al diccionario (generalmente una tabla hash) y se crea una nueva postings list (línea 6).



## 4. Indexación en memoria de un paso:

### Algoritmo SPIMI

- Cada posting es añadido directamente a su postings list, en lugar de esperar a tener todos los pares termID-docID y ordenarlos.
- Cada postings list es dinámica y está inmediatamente disponible para acoger postings.
- La talla de las postings list se adapta dinámicamente a las necesidades (líneas 8 y 9).
- Cuando se agota la memoria, se escribe el índice invertido del bloque (el diccionario y las postings lists) en disco (línea 12).
- Antes de este volcado en disco hay que ordenar los términos (línea 11), ya que se quiere escribir las postings lists en orden lexicográfico para facilitar el mezclado final.



## 4. Indexación en memoria de un paso:

### Algoritmo SPIMI

- Cada llamada escribe un bloque en disco.
- El último paso del SPIMI, no mostrado en el algoritmo de la figura, es la mezcla de los bloques para la construcción del índice invertido final.
- Tanto los postings como el diccionario de términos pueden almacenarse de forma compacta empleando compresión.
- El coste del SPIMI es  $\Theta(T)$  ya que no se realiza ordenación de tokens (sino de términos), y todas las operaciones son como mucho lineales con la talla de la colección.



## 5. Indexación distribuida

- Para grandes colecciones de documentos la construcción de índices no puede realizarse en un único ordenador.
- Especialmente cierto para indexación de la web.
  - Los centros de datos de búsqueda en web tienen sedes distribuidas a lo largo del mundo (Google, Bing, Baidu).
  - A pesar de que no se conocen las cifras exactas, se estima que Google mantiene más de 2.000.000 de servidores, distribuidos en varias ciudades del mundo.

Se hace necesario el uso de la indexación distribuida para la construcción de los índices





## 5. Indexación distribuida

- El resultado del proceso de construcción es un índice distribuido entre varios ordenadores estándar.
- Los índices se distribuyen en base a:
  - **Términos**: una máquina se ocupa de un rango de términos
  - **Documentos**: una máquina se ocupa de un rango de documentos.

- En un cluster de ordenadores se resuelve un problema distribuyéndolo entre ordenadores estándar (*nodos*).
- Un *nodo maestro* dirige el proceso de asignar y reasignar tareas a los nodos.



## 5. Indexación distribuida

- Se reparte la indexación en conjuntos de tareas paralelas
- El nodo maestro asigna cada tarea a una máquina desocupada
- Se definen dos conjuntos de tareas paralelas:
  - Análisis
  - Inversión de índices
- Se corta la colección de documentos de entrada en bloques.
- Cada bloque es un subconjunto de documentos (correspondiente a un bloque en BSBI/SPIMI)



## 5. Indexación distribuida

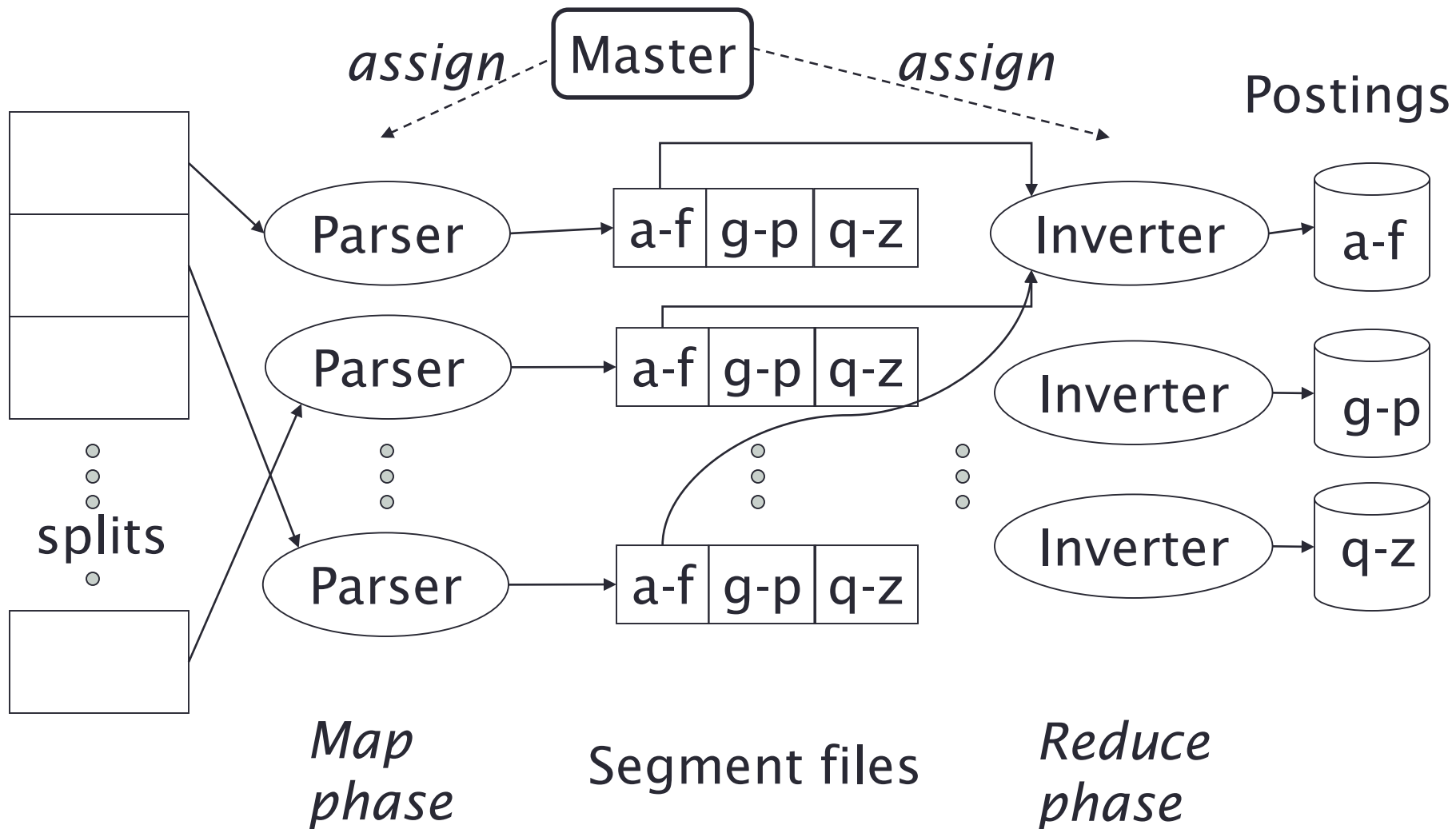
### Análisis

- El nodo maestro asigna un bloque a una máquina desocupada.
- El analizador lee el documento a su vez y emite los pares (término, documento).
- El analizador escribe los pares en  $j$  particiones.
- Cada partición corresponde a un rango de la primera letra del término
  - (e.g., **a-f**, **g-p**, **q-z**) en este caso  $j = 3$ .

### Inversión de índices

- Recoge todos los pares (término, documento) para una de las particiones basada en términos.
- Genera el índice invertido

## 5. Indexación distribuida





## 6. Indexación dinámica

- ¿Son las colecciones de documentos estáticas?
  - Nuevos documentos aparecen en la colección y han de ser insertados.
  - Otros documentos son borrados o modificados
- Ello implica que tanto el diccionario como las postings lists han de ser modificados:
  - Para términos presentes en el diccionario hay que actualizar las postings lists.
  - Los términos nuevos hay que añadirlos al diccionario.



## 6. Indexación dinámica: una solución simple

- Mantener dos índices en paralelo:
  - un gran índice principal y
  - un índice auxiliar pequeño que se encarga de los nuevos documentos (almacenado en memoria,  $n$  postings).
- Las búsquedas se realizan en ambos índices y los resultados se mezclan.
- Los borrados se almacenan en un vector de bits, que se utiliza para filtrar los resultados antes de mostrarlos.
- Las modificaciones se implementan como un borrado y una inserción.
- Periódicamente se mezclan los dos índices.



## 6. Indexación dinámica: mezcla logarítmica

- Mantener una serie de índices ( $I_0, I_1, I_2, \dots$ ), cada uno el doble de grande que su antecesor ( $2^0 \times n, 2^1 \times n, 2^2 \times n, \dots$ )
  - Cada vez, se instancia una potencia de 2
- Los postings son procesados sólo una vez en cada nivel.
- Mantiene un índice auxiliar pequeño ( $Z_0$ ) en memoria ( $|Z_0|=n$ )
- Mantiene índices grandes ( $I_0, I_1, \dots$ ) en disco
- Si  $Z_0$  se pasa de la talla máxima ( $> n$ ),
  - escribe los  $2^0 \times n$  postings en disco como  $I_0$
  - o es mezclado con  $I_0$  (si  $I_0$  ya existe) y crea  $Z_1$  de talla  $2^1 \times n$
  - Se escribe  $Z_1$  en disco como  $I_1$  (si no existe  $I_1$ )
  - o se mezcla con  $I_1$  para formar  $Z_2$
  - ...

## 6. Indexación dinámica: mezcla logarítmica

LMERGEADDTOKEN(*indexes*,  $Z_0$ , *token*)

```
1   $Z_0 \leftarrow \text{MERGE}(Z_0, \{\text{token}\})$ 
2  if  $|Z_0| = n$ 
3    then for  $i \leftarrow 0$  to  $\infty$ 
4      do if  $l_i \in \text{indexes}$ 
5        then  $Z_{i+1} \leftarrow \text{MERGE}(l_i, Z_i)$ 
6          ( $Z_{i+1}$  is a temporary index on disk.)
7           $\text{indexes} \leftarrow \text{indexes} - \{l_i\}$ 
8        else  $l_i \leftarrow Z_i$  ( $Z_i$  becomes the permanent index  $l_i$ .)
9           $\text{indexes} \leftarrow \text{indexes} \cup \{l_i\}$ 
10         BREAK
11      $Z_0 \leftarrow \emptyset$ 
```

LOGARITHMICMERGE()

```
1   $Z_0 \leftarrow \emptyset$  ( $Z_0$  is the in-memory index.)
2   $\text{indexes} \leftarrow \emptyset$ 
3  while true
4  do LMERGEADDTOKEN(indexes,  $Z_0$ , GETNEXTTOKEN())
```





# Ejercicio

Para  $n = 2$  y  $1 \leq T \leq 16$ , haz una traza del algoritmo de la Mezcla Logarítmica .

Crea una tabla que muestre, para cada tiempo en el cual han sido procesados  $T = 2 * k$  tokens ( $1 \leq k \leq 8$ ), cuál de los índices  $l_0, \dots, l_3$  está en uso.



# Ejercicio

T	$I_0$	$I_1$	$I_2$	$I_3$
2				
4				
6				
8				
10				
12				
14				
16				



# Ejercicio

T	$I_0$	$I_1$	$I_2$	$I_3$
2	1	0	0	0
4	0	1	0	0
6	1	1	0	0
8	0	0	1	0
10	1	0	1	0
12	0	1	1	0
14	1	1	1	0
16	0	0	0	1



## 6. Indexación dinámica: comparación

- Índices auxiliar y principal: el coste temporal de la construcción de índices es  $O(T^2/n)$  ya que cada posting se toca en cada mezcla.
- Mezcla logarítmica: cada posting es mezclado  $O(\log T/n)$  veces, por tanto la complejidad es  $O(T \log T/n)$
- Pero el procesamiento de una consulta requiere la mezcla de  $O(\log T/n)$  índices
  - Mientras que en el primer caso es  $O(1)$
- Algunos grandes motores de búsqueda realizan una construcción de índices a partir de cero de forma periódica.

# Otros tipos de índices

- Índices posicionales

- El volumen de datos es mayor
  - (termID, docID, (position1, position2, . . . )),
  - en lugar de (termID, docID)
- Los postings contienen información adicional además de docID.

to, 993427:

```
<1, 6: {7, 18, 33, 72, 86, 231};  
2, 5: {1, 17, 74, 222, 255};  
4, 5: {8, 16, 190, 429, 433};  
5, 2: {363, 367};  
7, 3: {13, 23, 191}; ... >
```

be, 178239:

```
<1, 2: {17, 25};  
4, 5: {17, 191, 291, 430, 434};  
5, 3: {14, 19, 101}; ... >
```

- Restricciones de acceso a documentos:

- Listas de control de accesos (ACLs): representan cada documento como una lista de usuarios con acceso, matriz usuario-documento.
- Se invierte la matriz de forma que para cada usuario se mantiene una postings list con los docID a los que tiene acceso.
- El resultado de la búsqueda se intersecta con la ACL del usuario, de forma que se filtran los documentos sin acceso.



# COMPRESIÓN DE ÍNDICES

---

2.1 Introducción

2.2 Compresión de diccionarios

2.3 Compresión de los ficheros de postings

2.4 Compresión de texto

# 1. Introducción

BRUTUS	→	1	2	4	11	31	45	173	174	
CAESAR	→	1	2	4	5	6	16	57	132	...
CALPURNIA	→	2	31	54	101					

- Estadísticas de la colección (RCV1)
  - ¿Cómo de grandes serán el diccionario y las postings lists?
- Compresión de diccionarios
- Compresión de postings lists



# 1. Introducción: ¿por qué compresión?

- Uso de menor espacio en disco.
- Almacenamiento de más información en memoria.
  - Aumenta la velocidad de la búsqueda
- Incrementa la velocidad de transferencia de datos de disco a memoria.



# 1. Introducción: ¿por qué compresión de índices?

- **Diccionario**

- Conseguir que quepa en memoria principal y
- que quepan algunas postings lists en memoria

- **Ficheros de Postings**

- Se reduce las necesidades de espacio en disco
- Se reduce el tiempo de lectura de las postings lists de disco
- Los grandes motores de búsqueda almacenan una parte importante de los postings en memoria.



# 1. Introducción: la colección RCV1

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52



# 1. Introducción: compresión con y sin pérdida de información

Efecto del preproceso sobre el número de términos, el número de postings no posicionales y posicionales en la colección Reures-RCV1.

- El número de términos determina la talla del diccionario.
- El número de postings determina la talla del índice
- “ $\Delta\%$ ” indica la reducción en talla respecto de la línea anterior, excepto para “30 stop words” y “150 stop words” que se calcula en referencia a “case folding”.
- “cumul %” es la reducción total respecto de “unfiltered”.



# 1. Introducción: compresión con y sin pérdida de información

- El preproceso afecta en gran medida la talla tanto del diccionario como de las postings lists.
- “Stemming” y “case folding” reduce el número de términos en un 17% y el número de nonpositional postings en 4% y 3%, respectivamente.
- El tratamiento de las palabras más frecuentes también aporta cambios importantes:
  - La eliminación de “30 stopwords” reduce en un 31% el número de postings posicionales.
  - La eliminación de “150 stopwords” reduce en un 30% el número de nonpositional postings.

# 1. Introducción: compresión con y sin pérdida de información

- Compresión **sin** pérdida de información: se preserva toda la información.
  - Lo más usual en RI.
- Compresión **con** pérdida de información: se descarta cierta información
- Ciertos preprocesos pueden ser vistos como compresión con cierta pérdida de información: case folding, stop words, stemming, number elimination.



# 1. Introducción: vocabulario versus talla de la colección

- ¿Cómo de grande es el vocabulario de términos?
  - Cuántas palabras distintas contienen los documentos?
- ¿Podemos asumir una cota superior?
  - Aunque los diccionarios como el Oxford English Dictionary definen más de 600.000 palabras, las grandes colecciones contienen muchas más (incluyen nombres de gente, lugares, productos, etc.)
- En la práctica, el vocabulario crece conforme crece la talla de la colección



# 1. Introducción: dos leyes empíricas

## La ley de Heaps:

¿Cuántos términos diferentes se esperan en una colección de documentos?

## La ley de Zipf:

¿Cuál es la distribución de frecuencias de los términos?



# 1. Introducción: vocabulario versus talla de la colección, la ley de Heaps

- La ley de Heaps: el número de palabras diferentes  $M$  en un texto es proporcional a  $T$ :

$$M = kT^b$$

donde  $T$  es el número de tokens de la colección.

- Valores típicos:  $30 \leq k \leq 100$  y  $b \approx 0.5$
- En una gráfica con escalas logarítmicas de la talla del vocabulario  $M$  vs.  $T$ , la ley de Heaps predice una línea con pendiente  $\frac{1}{2}$ .



# 1. Introducción: vocabulario versus talla de la colección, la ley de Heaps

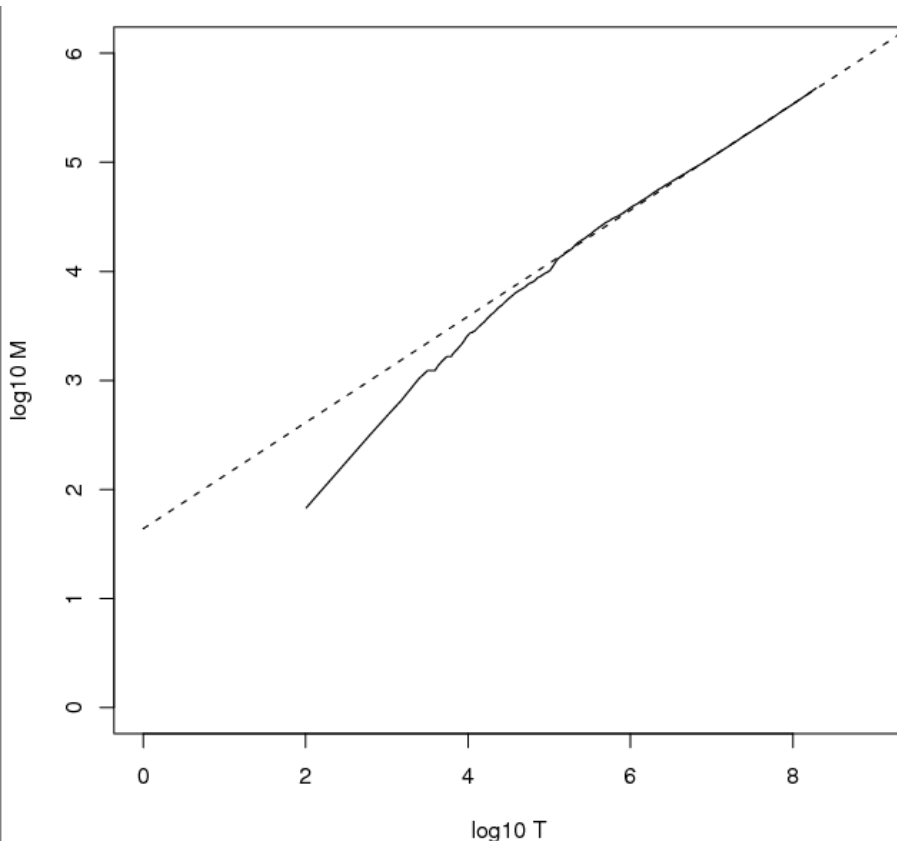
- Para RCV1, la línea discontinua

$\log_{10} M = 0.49 \log_{10} T + 1.64$  el  
mejor por mínimos cuadrados

$M = 10^{1.64} T^{0.49}$  por tanto

$k = 10^{1.64} \approx 44$  y  $b = 0.49$ .

- Para los primeros 1,000,020 tokens, la ley predice 38,323 términos
- Realmente hay 38,365



# 1. Introducción: vocabulario versus talla de la colección, la ley de Heaps

Independientemente de los valores de los parámetros de una determinada colección, la ley de Heaps sugiere que:

- La talla del diccionario crece según va creciendo la talla de la colección de documentos, no tiende a una talla máxima
- La talla del diccionario es bastante grande para grandes colecciones.

La compresión del diccionario es importante para un sistema de RI.



# 1. Introducción: ley de Heaps, ejercicio

Se nos proporciona una muestra random de 10.000 documentos de una colección de 1.000.000 docs. Hay 5.000 palabras diferentes en la muestra. Suponiendo que la colección satisface la ley de Heaps con exponente 0.5, se pide calcular una estimación de las palabras diferentes  $M$  en la colección completa.

Ley de Heaps:  $M = kT^b$

Donde  $M$  es la talla del vocabulario y  $T$  el número de palabras total. En este caso  $b=0,5$

# 1. Introducción: ley de Heaps, ejercicio

Sea  $n$  el número medio de tokens por documento, de forma que el número de tokens en la muestra es  $T_m = 10.000 \times n$ , y en la colección es,  $T = 1.000.000 \times n$ .

Las constantes de la ley de Heaps,  $K$  y  $b$ , serán las mismas para la muestra y la colección, por lo que:

$$K = M / T^b = 5.000 / T_m^b$$

$$M / (1.000.000 \times n)^{0,5} = 5.000 / (10.000 \times n)^{0,5}$$

Por lo que

$$M = 5.000 \times (1.000.000 \times n)^{0,5} / (10.000 \times n)^{0,5} = 50.000$$



# 1. Introducción: vocabulario versus talla de la colección, la ley de Zipf

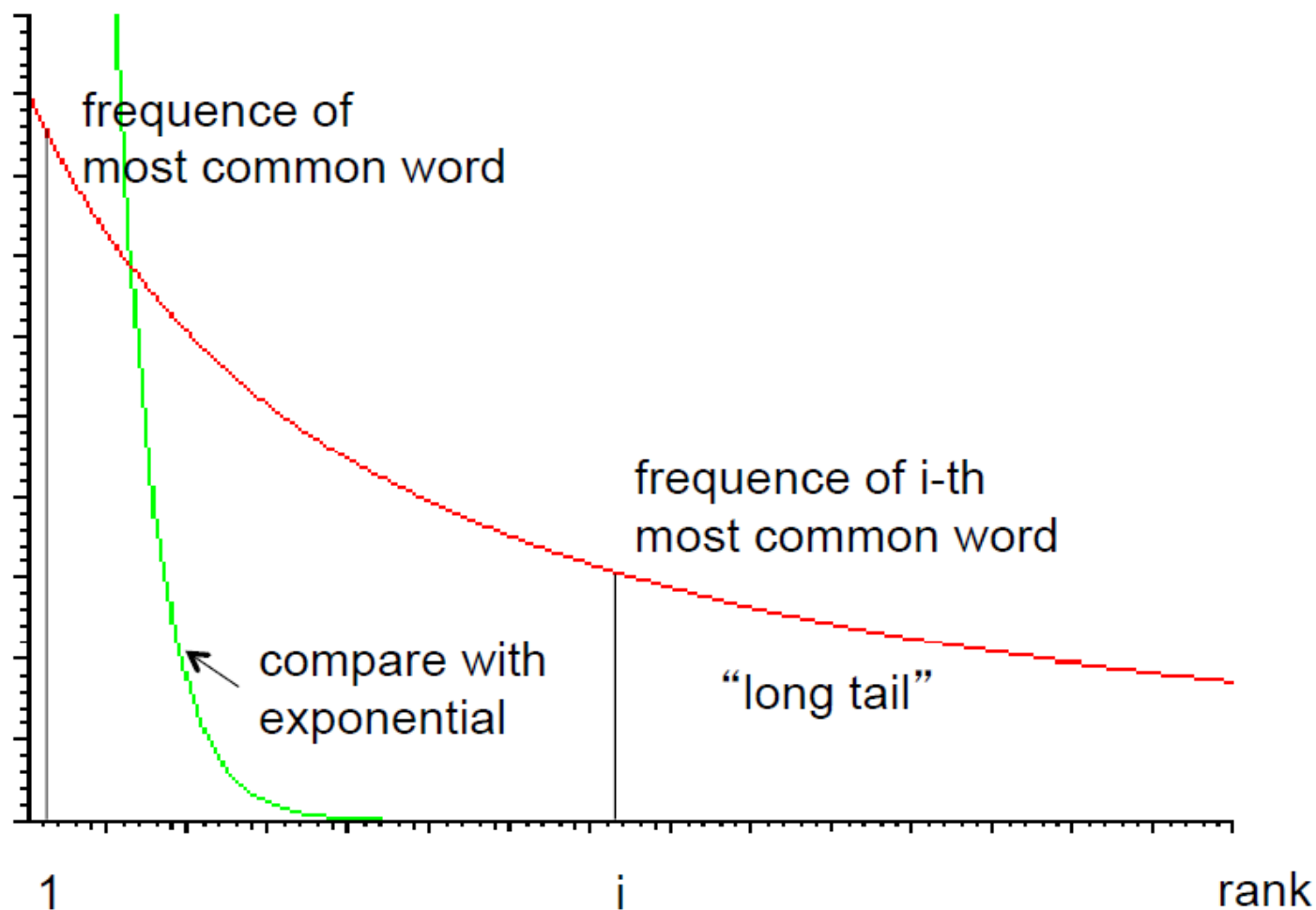
Estudio de las frecuencias relativas de los términos en la colección.

- En lenguaje natural, hay unos pocos términos muy frecuentes y muchos términos que aparecen con baja frecuencia.
- Ley de Zipf: El *i*ésimo término más frecuente tiene una frecuencia proporcional a  $1/i$ .

$$cf_i \propto 1/i$$

$cf_i$  es la *frecuencia de colección*: el número de ocurrencias del término  $t_i$  en la colección.

# 1. Introducción: vocabulario versus talla de la colección, la ley de Zipf



# 1. Introducción: vocabulario versus talla de la colección, la ley de Zipf

- Si el término más frecuente ocurre  $cf_1$  veces
  - el segundo más frecuente ocurre  $cf_1/2$  veces
  - el tercero más frecuente ocurre  $cf_1/3$  veces ...
- De forma equivalente:

$$cf_i = K/i$$

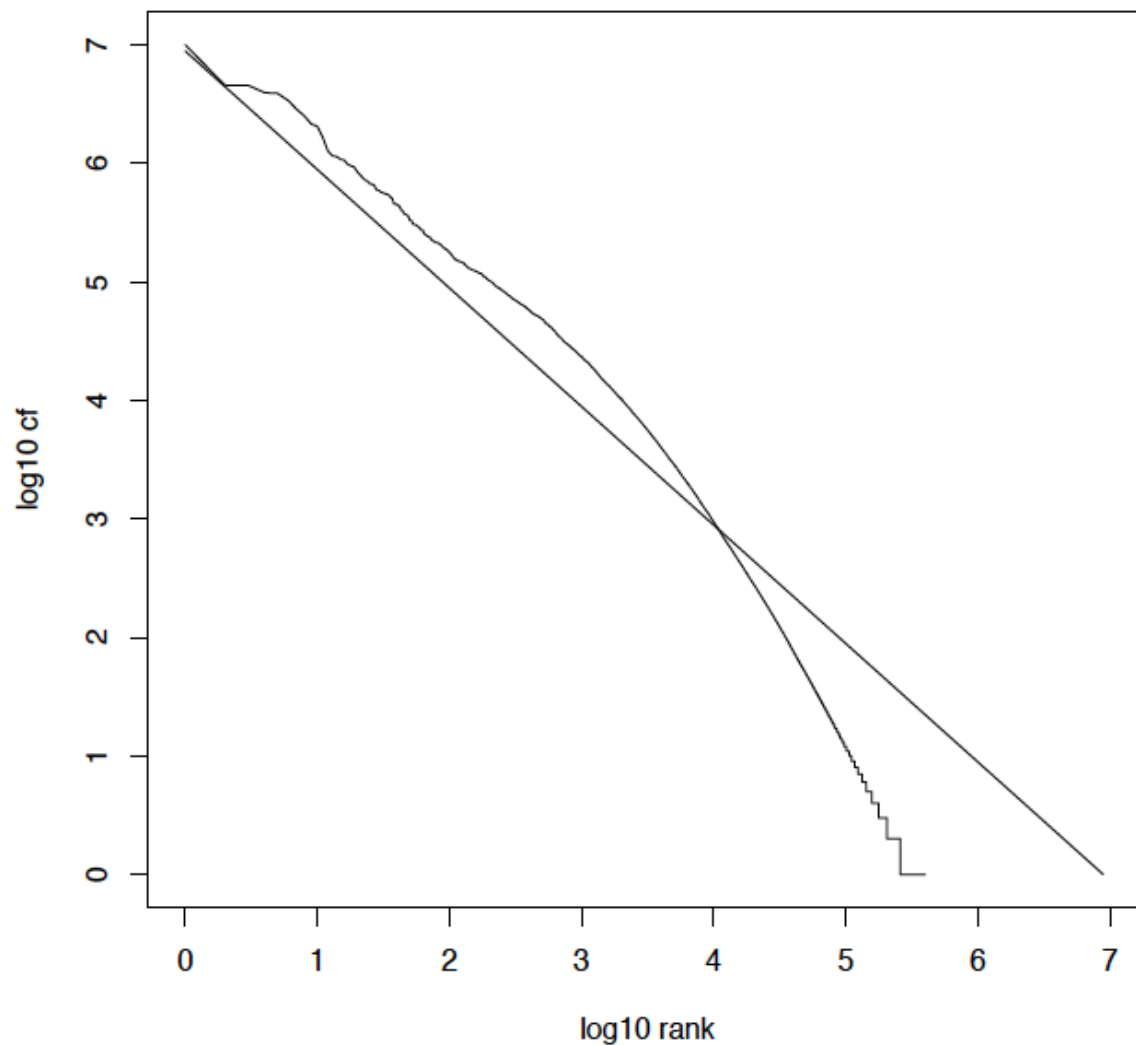
donde  $K$  es un factor de normalización,

- Por tanto:

$$\log cf_i = \log K - \log i$$

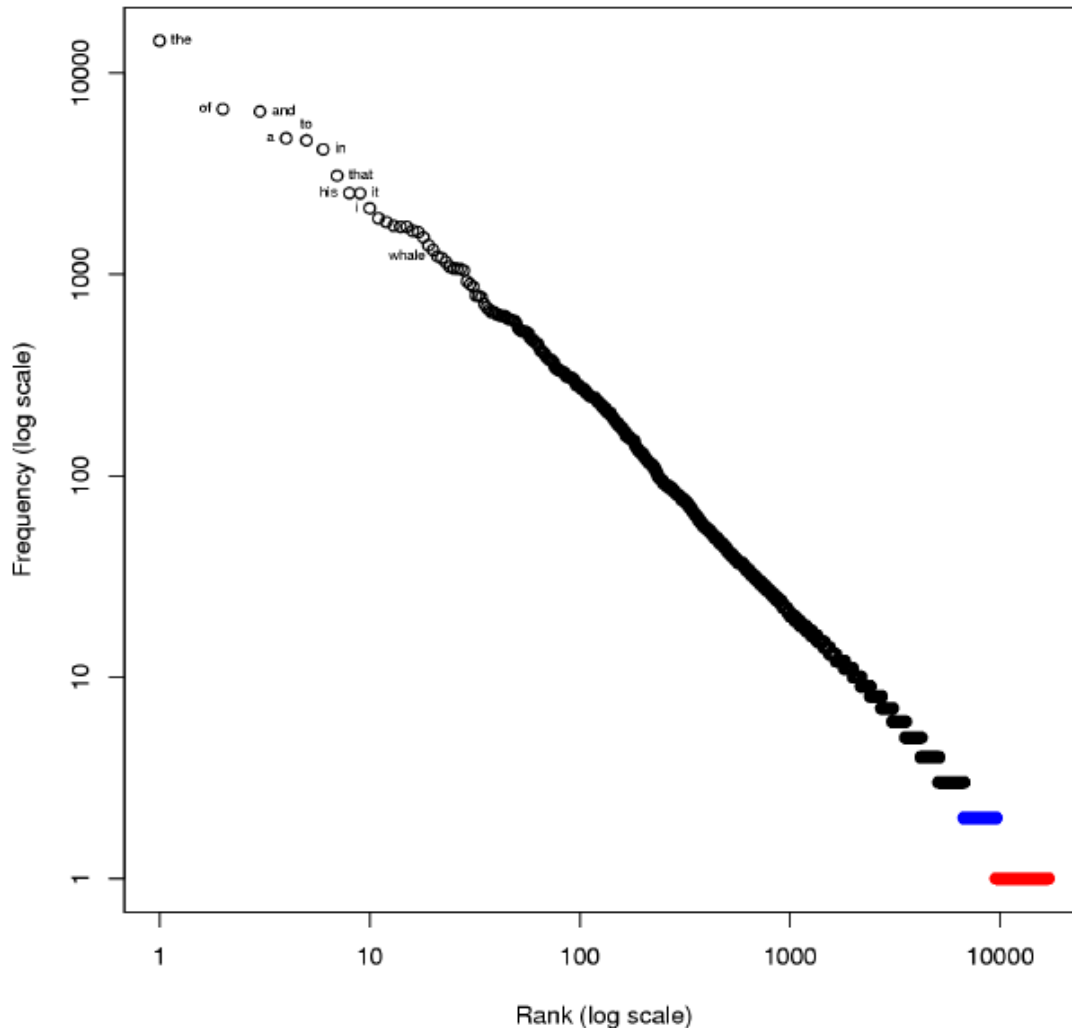
$\Rightarrow$  hay una relación lineal entre  $\log cf_i$  y  $\log i$

# 1. Introducción: vocabulario versus talla de la colección, la ley de Zipf para RCV1





# 1. Introducción: vocabulario versus talla de la colección, la ley de Zipf



Distribution of word frequencies in Melville's "Moby Dick"

Log-log plot

[Wikipedia, "Hapax Legomenon", Sept. 24th, 2011]

## 1. Introducción: ley de Zipf, ejercicio

Asumiendo que la longitud de la postings list de una colección de documentos sigue una ley de Zipf  $\sim i^{-2}$ , y que la lista más larga tiene 10.000 doc ID, cuántas listas se espera que tengan una longitud  $\geq 100$  ?

La ley de Zipf:  $cf_i = K/i^2$

# 1. Introducción: ley de Zipf, ejercicio

Asumiendo que la longitud de la postings list de una colección de documentos sigue una ley de Zipf  $\sim i^{-2}$ , y que la lista más larga tiene 10.000 doc ID, cuántas listas se espera que tengan una longitud  $\geq 100$  ?

La ley de Zipf:  $cf_i = K/i^2$

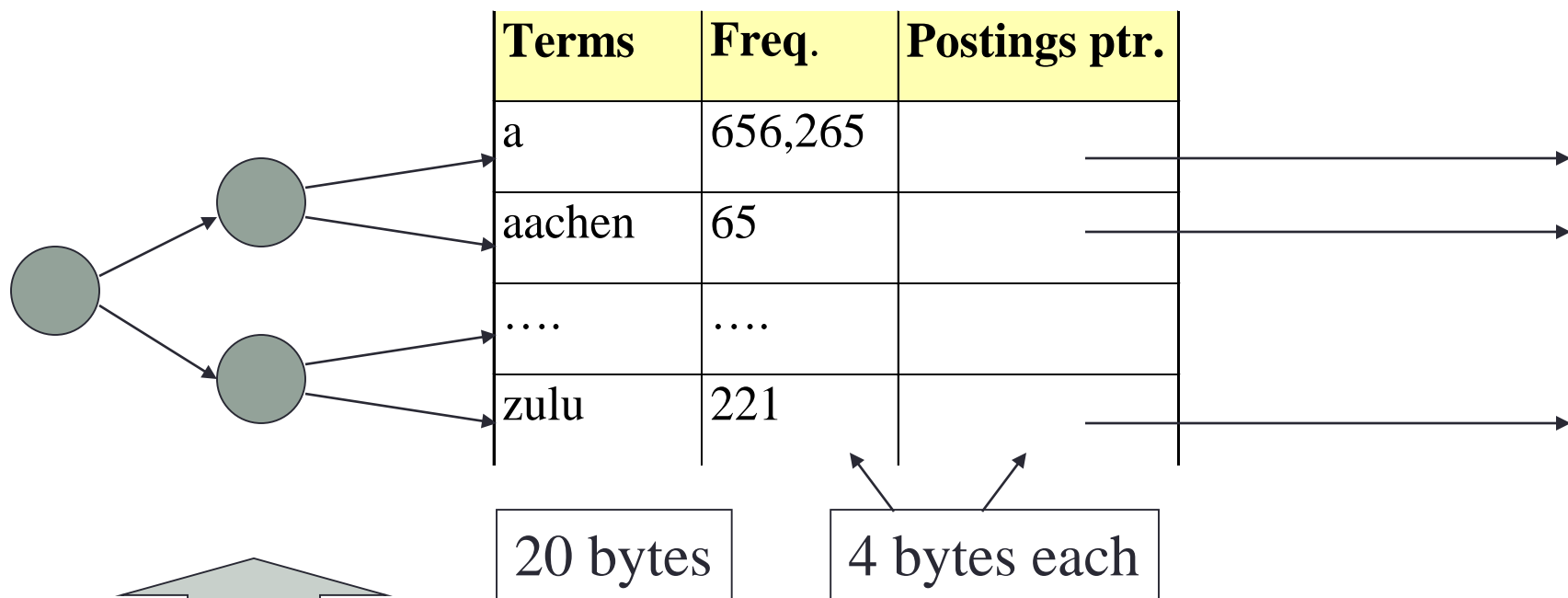
Para  $i=1$ ,  $cf_1=10.000$  por lo que  $K=10.000$ .

¿Qué valor ha de tener la  $i$  para que la frecuencia sea 100? Aplicamos la ley con el valor de  $k$ :

$$100 = K / i^2, \text{ por lo que } i = 10.$$

Por tanto **las primeras 10 listas más frecuentes** tendrán una longitud superior a 100.

## 2. Compresión de diccionarios: un vector de talla fija

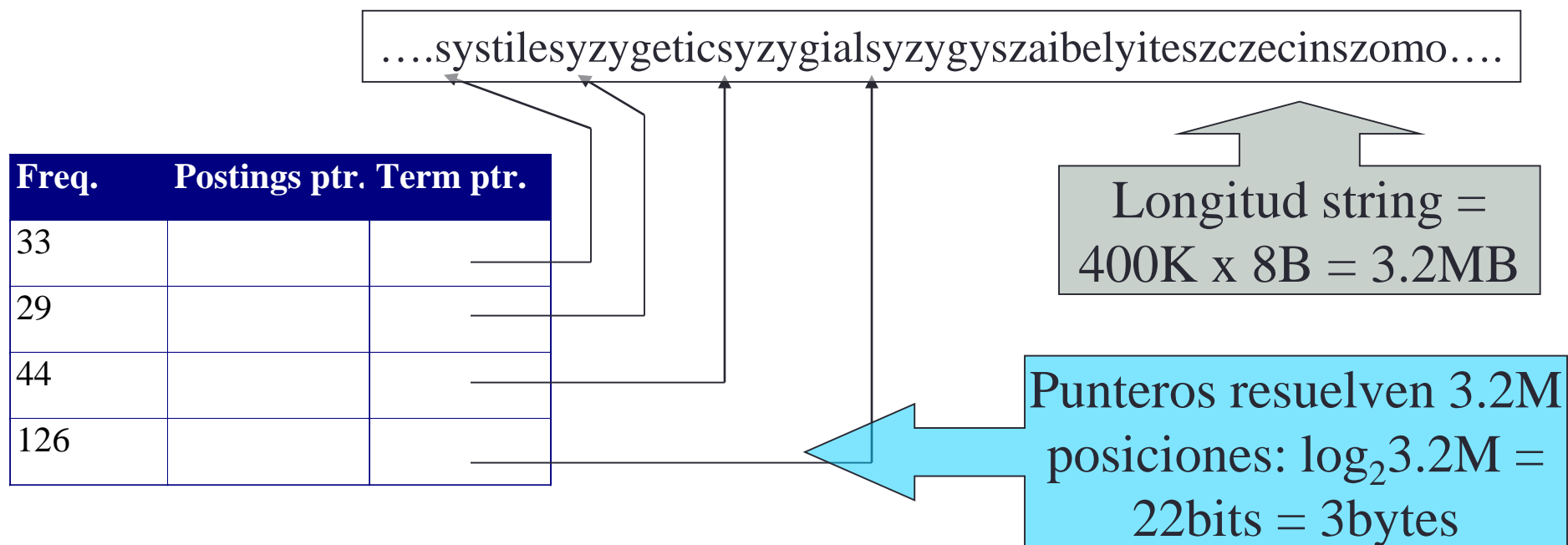


Estructura de búsqueda  
en el diccionario

~400,000 térmns; 28 bytes/térm = 11.2 MB

## 2. Compresión de diccionarios: un string de caracteres

El puntero a la palabra siguiente muestra el final de la actual  
Esperamos ahorrar hasta un 1/3 de espacio en el diccionario.



~400,000 términos; (4+4+3+8) bytes/término = 7.6 MB

## 2. Compresión de diccionarios: blocking

Guarda punteros para cada bloque de términos de talla  $k$   
( $k=4$  en el ejemplo)

Necesita almacenar la longitud de los términos (1 extra byte).

....**7***systile***9***syzygetic***8***syzygial***6***syzygy***11***szaibelyite***8***szczecin***9***szomo*....

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

Ahorra 9 bytes  
en 3 ( $k-1$ )  
punteros.

Pierde 4 bytes en  
Longitud térm.

$\sim 400,000 \times (1/4) \times 5B \text{ reduc.} = 0.5 \text{ MB reduc.}$

## 2. Compresión de diccionarios: blocking+ front coding

La palabras ordenadas suelen compartir largos prefijos comunes.  
⇒ Almacenar sólo las diferencias (para las últimas  $k-1$  en un bloque de  $k$ )

**8***automata***8***automate***9***automatic***1****0***automation*

→ **8***automat*\***a****1**◇**e****2**◇**ic****3**◇**ion**

Codifica *automat*

Longitud extra  
Sobre *automat*.



## 2. Compresión de diccionarios: resumen

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
Also, blocking $k = 4$	7.1
Also, Blocking + front coding	5.9





### 3. Compresión de los ficheros de postings

- Los ficheros de postings son más grandes que los de diccionario, al menos en un factor 10.
- Es deseable almacenar los postings de la forma más compacta posible.
- En este apartado consideraremos que un posting es un docID.
- Para Reuters (800,000 documentos), podríamos usar  $\log_2 800,000 \approx 20$  bits por docID.

Índice NO comprimido para Reuters-RCV1 es 250MB ( $100M \times 20b/8$ )



### 3. Compresión de los ficheros de postings

- Un término como ***arachnocentric*** ocurre tal vez en un documento de un millón.  $\Rightarrow$  Necesitaremos para almacenar su posting 20 bits.
- Un término como ***the*** ocurre posiblemente en cada documento, de manera que 20 bits/posting es demasiado caro.

### 3. Compresión de los ficheros de postings

- Almacenamos la lista de documentos que contienen un término en orden creciente de docID.
  - **computer**: 33,47,154,159,202 ...
- $\Rightarrow$  Sería suficiente almacenar los *gaps*.
  - 33,14,107,5,43 ...
- Siempre que almacenar los gaps sea más barato que 20 bits.



### 3. Compresión de los ficheros de postings

	encoding	postings list					
THE	docIDs	...	283042	283043	283044	283045	...
	gaps		1	1	1		...
COMPUTER	docIDs	...	283047	283154	283159	283202	...
	gaps		107	5	43		...
ARACHNOCENTRIC	docIDs	252000	500100				
	gaps	252000	248100				

Para una representación económica de la distribución de gaps, necesitamos un método que use pocos bits para gaps cortos.



### 3. Compresión de los ficheros de postings: codificación de longitud variable

- Objetivo:
  - Para **arachnocentric**, usaremos ~20 bits/gap por entrada.
  - Para **the**, usaremos ~1 bit/gap por entrada.
- Si el gap medio por término es  $G$ , queremos usar  $\sim \log_2 G$  bits/gap por entrada.
- Idea: codificar cada entero (gap) con el menor número posible de bits que necesitaríamos para ese entero.
- Esto requiere una **codificación de longitud variable**.



### 3. Compresión de los ficheros de postings: codificación variable en bytes (VB)

- Uso de un número entero de bytes para codificar un gap.
- Para un valor de gap  $G$ , se usa el número de bytes menor que permite representar  $\log_2 G$  bits.
- De cada byte se usan los 7 bits del final para la codificación y el bit inicial es un *bit de continuación*  $c$ .
- Si  $G \leq 127$ , se codifica en binario en los 7 bits disponibles y se fija  $c = 1$
- Si es mayor, de la codificación en binario de  $G$  los 7 bits de menor orden se almacenan en ese primer byte y se usan bytes adicionales para almacenar los de mayor orden, siguiendo el mismo algoritmo.
- El bit de continuación del último byte se fija a 1 ( $c = 1$ ), y para el resto  $c = 0$ .

### 3. Compresión de los ficheros de postings: codificación variable en bytes, ejemplo

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings almacenados como concatenación de bytes

000001101011100010000101000011010000110010110001

La codificación VB de postings es  
decodificable sin ambigüedad

Para un gap pequeño (5),  
usa un byte.

### 3. Compresión de los ficheros de postings: codificación variable en bytes

```
VBENCODENUMBER(n)  
1  bytes  $\leftarrow \langle \rangle$   
2  while true  
3  do PREPEND(bytes, n mod 128)  
4      if n < 128  
5          then BREAK  
6      n  $\leftarrow n \text{ div } 128$   
7  bytes[LENGTH(bytes)] += 128  
8  return bytes
```

```
VBENCODE(numbers)  
1  bytestream  $\leftarrow \langle \rangle$   
2  for each n  $\in$  numbers  
3  do bytes  $\leftarrow$  VBENCODENUMBER(n)  
4      bytestream  $\leftarrow$  EXTEND(bytestream, bytes)  
5  return bytestream
```



### 3. Compresión de los ficheros de postings: codificación variable en bytes

VBDECODE(*bytestream*)

```
1  numbers  $\leftarrow \langle \rangle$ 
2  n  $\leftarrow 0$ 
3  for i  $\leftarrow 1$  to LENGTH(bytestream)
4  do if bytestream[i] < 128
5      then n  $\leftarrow 128 \times n + \textit{bytestream}[\textit{i}]$ 
6      else n  $\leftarrow 128 \times n + (\textit{bytestream}[\textit{i}] - 128)$ 
7          APPEND(numbers, n)
8          n  $\leftarrow 0$ 
9  return numbers
```

Índice comprimido para Reuters-RCV1 es 116MB (~50% red)



### 3. Compresión de los ficheros de postings: codificación variable en bytes

Se pide decodificar la siguiente secuencia de bits  
codificada con codificación variable (CV) en bytes:  
000000111000010010001010



### 3. Compresión de los ficheros de postings: codificación variable en bytes

Se pide decodificar la siguiente secuencias de bits  
codificada con codificación variable (CV) en bytes:  
0000001111000010010001010

**Solución:**

00000011 10000100 10001010,  
que una vez decodificados son 388 ( $3 \cdot 128 + 4$ ), 10

La postings list seria: 388, 398



### 3. Compresión de los ficheros de postings: códigos gamma

- Vamos a comprimir a nivel de bit,
  - El código Gamma es el más conocido.
- Representa un gap  $G$  con un par *length* y *offset*
- *offset* es  $G$  en binario, con el bit líder eliminado
  - Por ejemplo  $13 \rightarrow 1101 \rightarrow 101$
- *Length* es la *longitud* del offset
  - Para 13 (offset 101), es 3.
- Se codifica *length* en unario: 1110.
- El código gamma para 13 es la concatenación de *length* y *offset*: 1110101

### 3. Compresión de los ficheros de postings: códigos gamma

number	length	offset	$\gamma$ -code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001



### 3. Compresión de los ficheros de postings: códigos gamma

Dar la secuencia de bits correspondiente a la compresión por códigos gamma de la siguiente posting list:

[10,15,22,23,34,44,50,58]

### 3. Compresión de los ficheros de postings: códigos gamma

Dar la secuencia de bis correspondientes a la compresión por códigos gamma de la siguiente posting list:

[10,15,22,23,34,44,50,58]

La secuencia de números (gaps) a codificar es:

[10,5,7,1,11,10,6,8]

La codificación usando códigos gamma es:

GAP	binario	Cog gamma
10	1010	1110010
5	101	11001
7	111	11011
1	1	0
11	1011	1110011
10	1010	1110010
6	110	11010
8	1000	1110000



### 3. Compresión de los ficheros de postings: códigos gamma

Se pide decodificar la siguiente secuencia de bits  
codificada con códigos gamma:

10110011100011100111100100





### 3. Compresión de los ficheros de postings: códigos gamma

Se pide decodificar la siguiente secuencia de bits  
codificada con códigos gamma:


10110011100011100111100100

**Solución:**

101 100 1110001 11001 1110010 0

que una vez decodificados se corresponde con la  
secuencia de gaps: 3, 2, 9, 5, 10, 1

Por lo que la postings list es : 3, 5, 14, 19, 29, 30



### 3. Compresión de los ficheros de postings: códigos gamma, propiedades

- $G$  es codificado usando  $2 \lfloor \log G \rfloor + 1$  bits
  - La longitud de offset es  $\lfloor \log G \rfloor$  bits
  - La longitud de length es  $\lfloor \log G \rfloor + 1$  bits
- Todos los códigos gamma tienen un número impar de bits
- Los códigos gamma son decodificados de forma no ambigua en base a prefijos (prefix-free).
- Pueden utilizarse para cualquier distribución
- Los códigos gamma no requieren fijar parámetros (parameter-free)



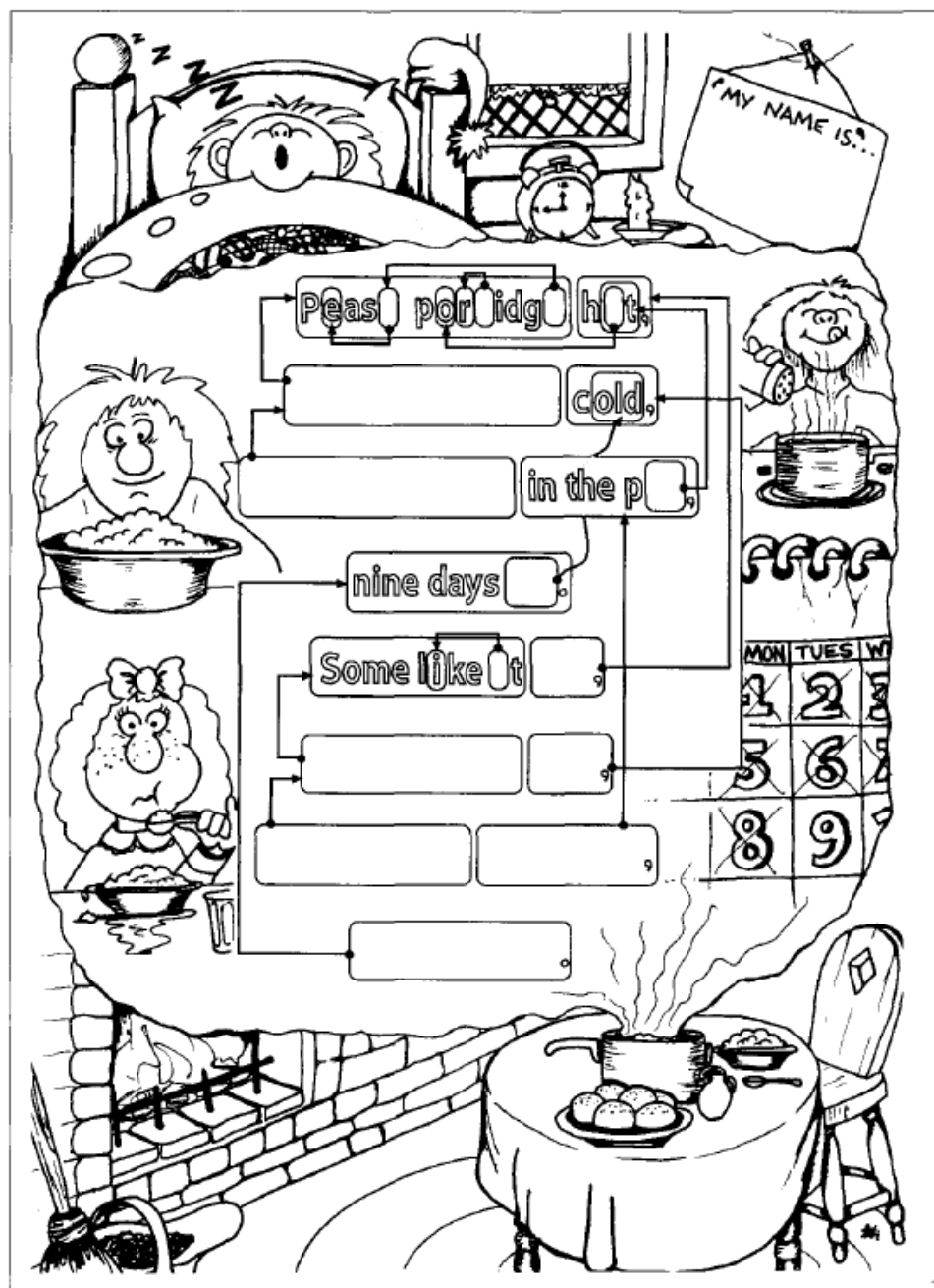
### 3. Compresión de diccionarios y de ficheros de postings

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, $k = 4$	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, $\gamma$ -encoded	101.0



## 4. Compresión de texto

- La compresión de texto implica un cambio de representación de un fichero de forma que:
  - Ocupe menos espacio de almacenamiento
  - Requiera menos tiempo de transmisión
- El fichero original puede ser reconstruido exactamente a partir de la representación comprimida.





## 4. Compresión de texto: clasificación

- **Métodos simbólicos:** estiman las probabilidades de los símbolos y codifican con códigos más cortos los símbolos más probables. **Códigos de Huffman.**
- **Métodos basados en diccionarios:** se reemplazan palabras o segmentos de texto por un índice a una entrada de un diccionario (ejemplo anterior). **Códigos LZ77 y LZ78.**



## 4. Compresión de texto: código de Huffman

- Codificar es la tarea de determinar la representación de salida de un símbolo, en base a una distribución de probabilidad proporcionada por un modelo.
- Uno de los métodos de compresión más conocidos es el código de Huffman.
- Es un método simbólico:
  - Los símbolos más comunes se codifican con pocos bits
  - Los símbolos menos frecuentes se codifican con códigos más largos.

## 4. Compresión de texto: código de Huffman

**Table 2.1 Codewords and probabilities for a seven-symbol alphabet.**

Symbol	Codeword	Probability
<i>a</i>	0000	0.05
<i>b</i>	0001	0.05
<i>c</i>	001	0.1
<i>d</i>	01	0.2
<i>e</i>	10	0.3
<i>f</i>	110	0.2
<i>g</i>	111	0.1

La secuencia *eefggfed* es codificada como 10101101111111101001

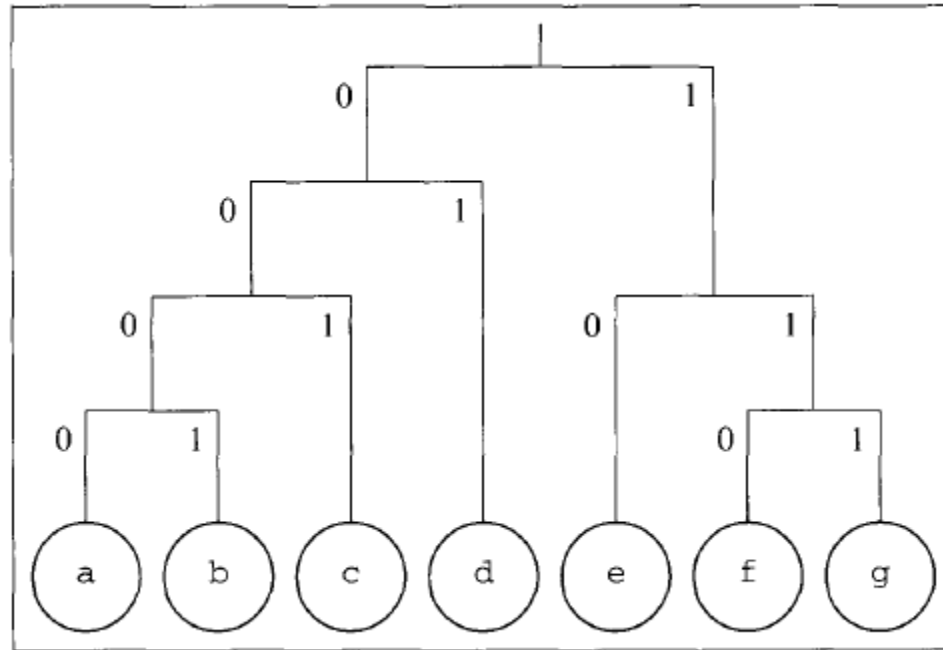
Notación:

*Codeword*: cada secuencia que se utiliza para codificar un símbolo.

*Codebook*: el diccionario de codewords



# Árbol para decodificar el código de Huffman



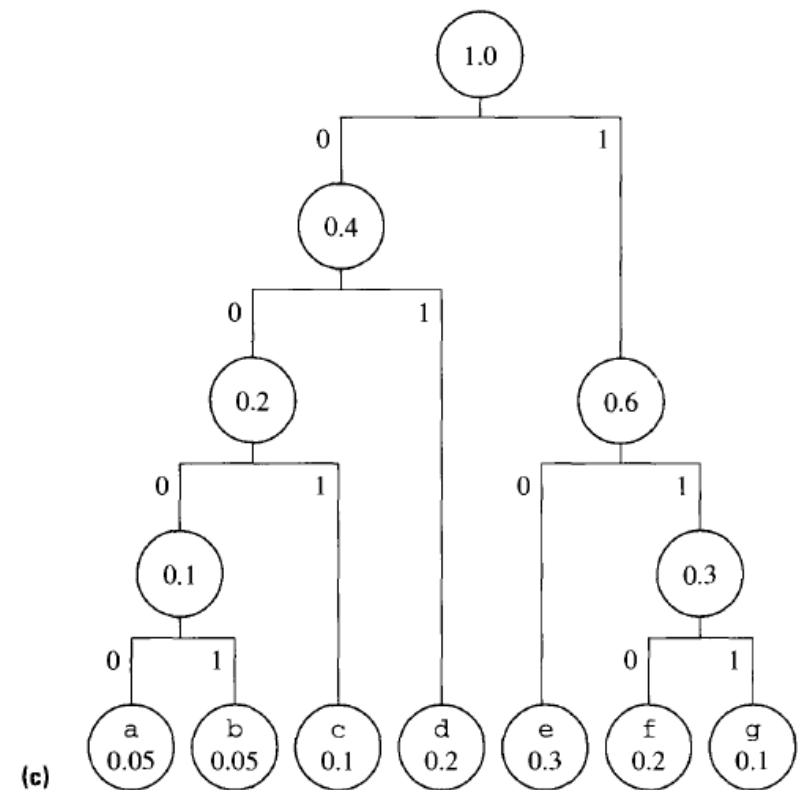
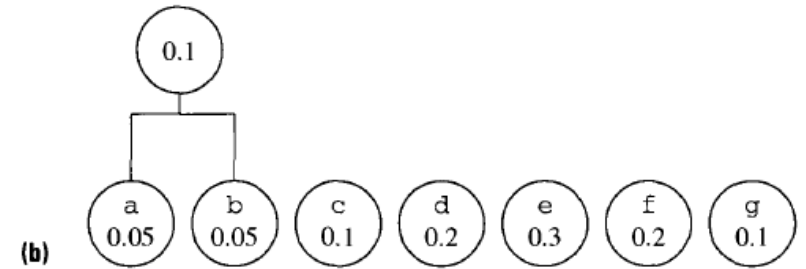
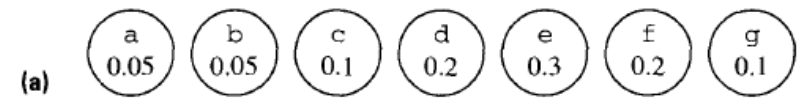
Es un código *prefix-free*: ningún codeword es prefijo de ningún otro. Ello permite una decodificación sin ambigüedades. Ej: **10101101111111101001** (*eefggfed*)

## Construcción del *árbol de Huffman*:

(a) Nodos hoja

(b) Combinación de nodos

(c) El árbol completo



# Cálculo del código de Huffman

To calculate a Huffman code,

1. Set  $T \leftarrow$  a set of  $n$  singleton sets, each containing one of the  $n$  symbols and its probability.
2. Repeat  $n - 1$  times
  - (a) Set  $m_1$  and  $m_2 \leftarrow$  the two subsets of least probability in  $T$ .
  - (b) Replace  $m_1$  and  $m_2$  with a set  $\{m_1, m_2\}$  whose probability is the sum of that of  $m_1$  and  $m_2$ .
3.  $T$  now contains only one item, which corresponds to the root of a Huffman tree; the length of the codeword for each symbol is given by the number of times it was joined with another set.



# Código de Huffman: ejemplo

Character	Frequency
'b'	3
'e'	4
'p'	2
' '	2
'o'	2
'r'	1
'i'	1





# Código de Huffman: ejemplo

Character	Code
'b'	00
'e'	11
'p'	101
' '	011
'o'	010
'r'	1000
'!'	1001

Cadena de entrada: beep boop beer!

Representación en binario: 0110 0010 0110 0101 0110 0101 0111 0000 0010  
0000 0110 0010 0110 1111 0110 1111 0111 0000 0010 0000 0110 0010 0110 0101 0110  
0101 0111 0010 0010 0001

Cadena codificada: 0011 1110 1011 0001 0010 1010 1100 1111 1000 1001



# Código de Huffman: ejercicio

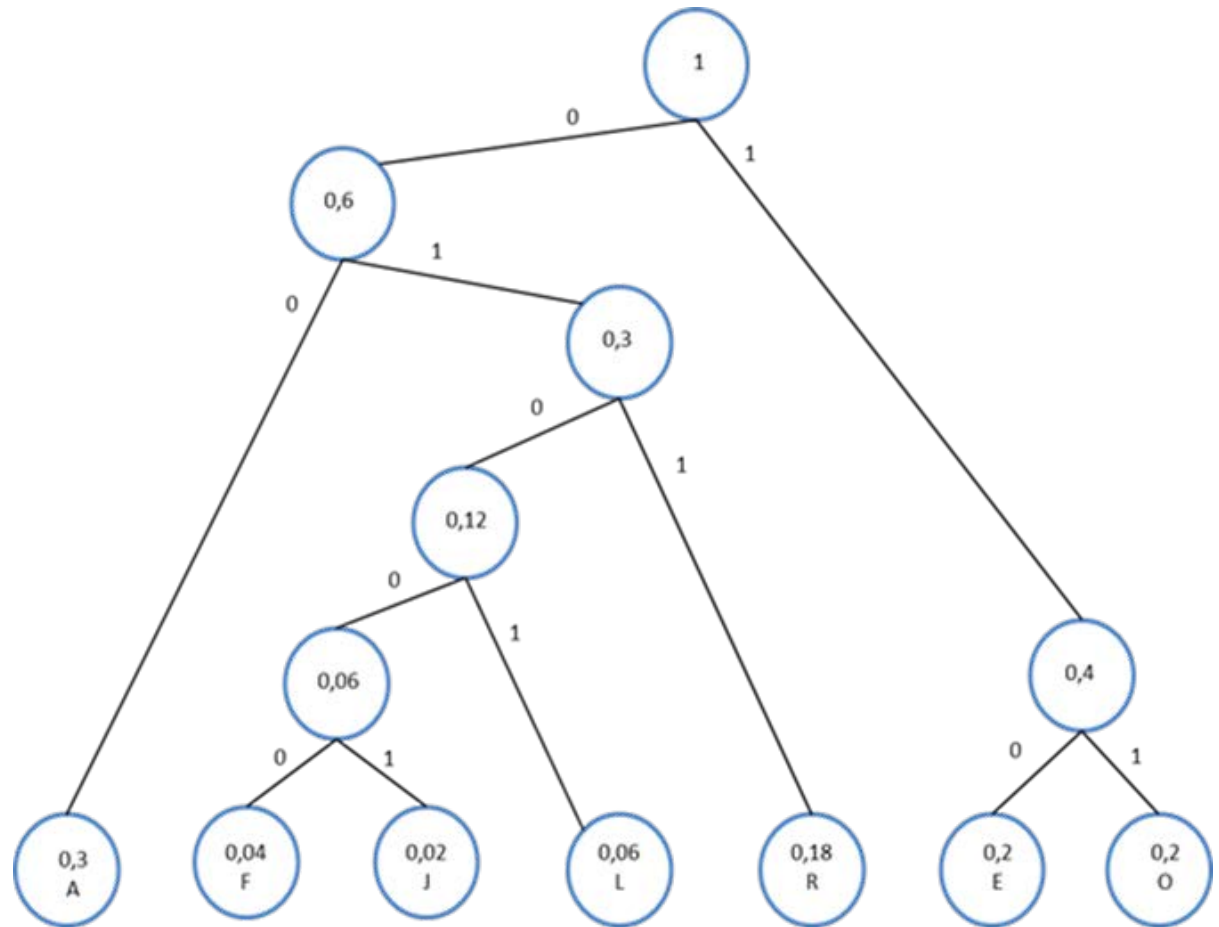
1. Dada la siguiente lista de símbolos y sus probabilidades se pide construir el **árbol de Huffman**

Símbolo	Probabilidad
A	0.30
E	0.20
F	0.04
J	0.02
L	0.06
O	0.20
R	0.18

2. Utilizando el árbol anterior se pide codificar la cadena "RELOJ"

# Código de Huffman: ejercicio

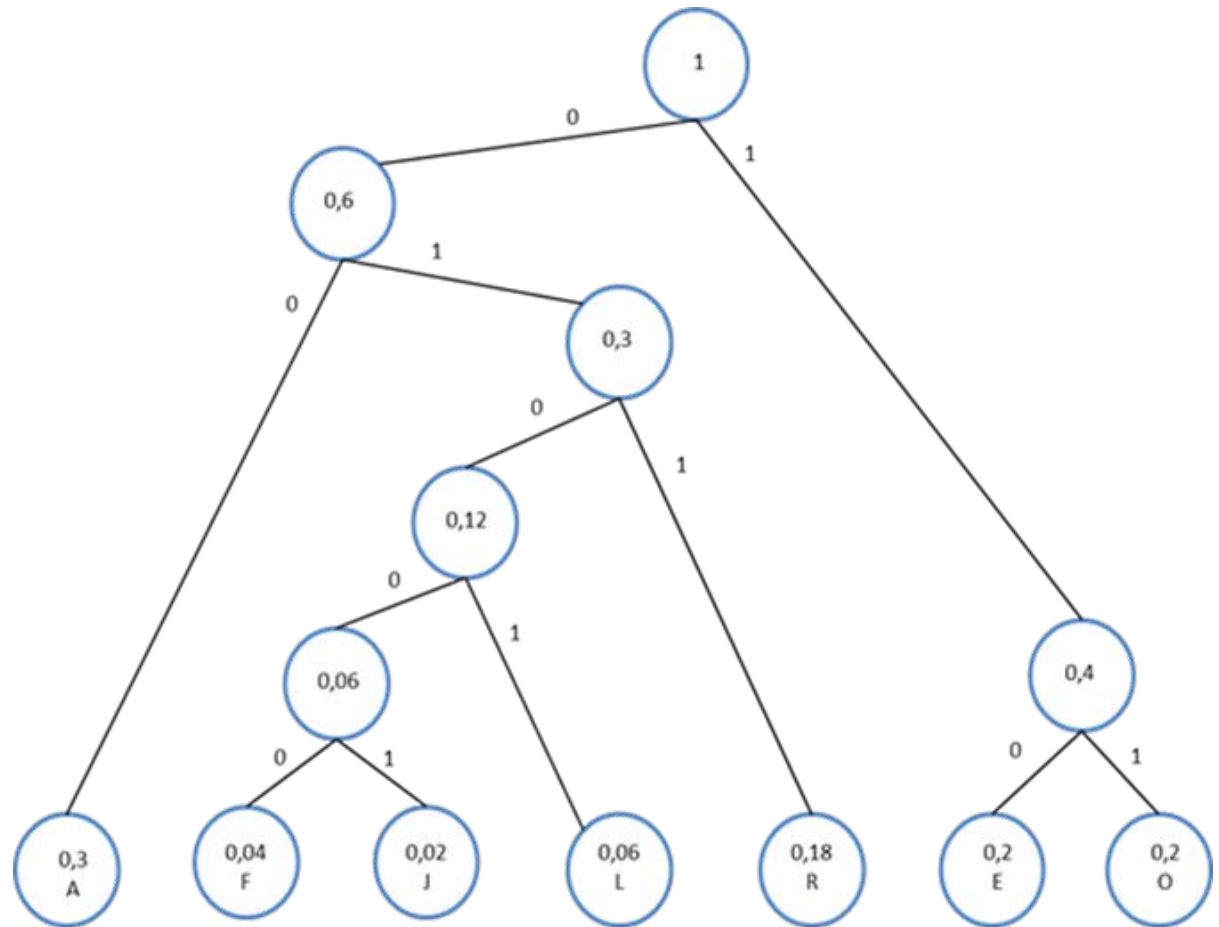
Símbolo	Probabilidad
A	0.30
E	0.20
F	0.04
J	0.02
L	0.06
O	0.20
R	0.18





# Código de Huffman: ejercicio

Símbolo	Probabilidad
A	0.30
E	0.20
F	0.04
J	0.02
L	0.06
O	0.20
R	0.18



Codificación de la cadena "RELOJ": 011 10 0101 11 01001

## Código de Huffman canónico:

Parte del código para un libro, donde el alfabeto está construido por las palabras que aparecen en el libro.

Symbol	Codeword	
	Length	Bits
100	17	000000000000000000
101	17	000000000000000001
102	17	000000000000000010
103	17	000000000000000011
...	...	...
yopur	17	00001101010100100
young	17	00001101010100101
youthful	17	00001101010100110
zeed	17	00001101010100111
zephyr	17	00001101010101000
zigzag	17	00001101010101001
11th	16	0000110101010101
120	16	0000110101010110
...	...	...
were	8	10100110
which	8	10100111
as	7	1010100
at	7	1010101
for	7	1010110
had	7	1010111
he	7	1011000
her	7	1011001
his	7	1011010
it	7	1011011
s	7	1011100
said	7	1011101
she	7	1011110
that	7	1011111
with	7	1100000
you	7	1100001
I	6	110001
in	6	110010
was	6	110011
a	5	11010
and	5	11011
of	5	11100
to	5	11101
the	4	1111



# El código de Huffman canónico

- La palabra representada por un codeword puede determinarse rápidamente a partir de la longitud de dicha secuencia y del codeword correspondiente de la primera palabra de dicha longitud.
- Por ejemplo, la palabra *said* es la *décima* de los codeword de longitud 7.
- Dada esta información, y que el primer codeword de longitud 7 es 1010100, podemos obtener el codeword para *said* incrementando 1010100, es decir, añadiendo *nueve* a su representación binaria.

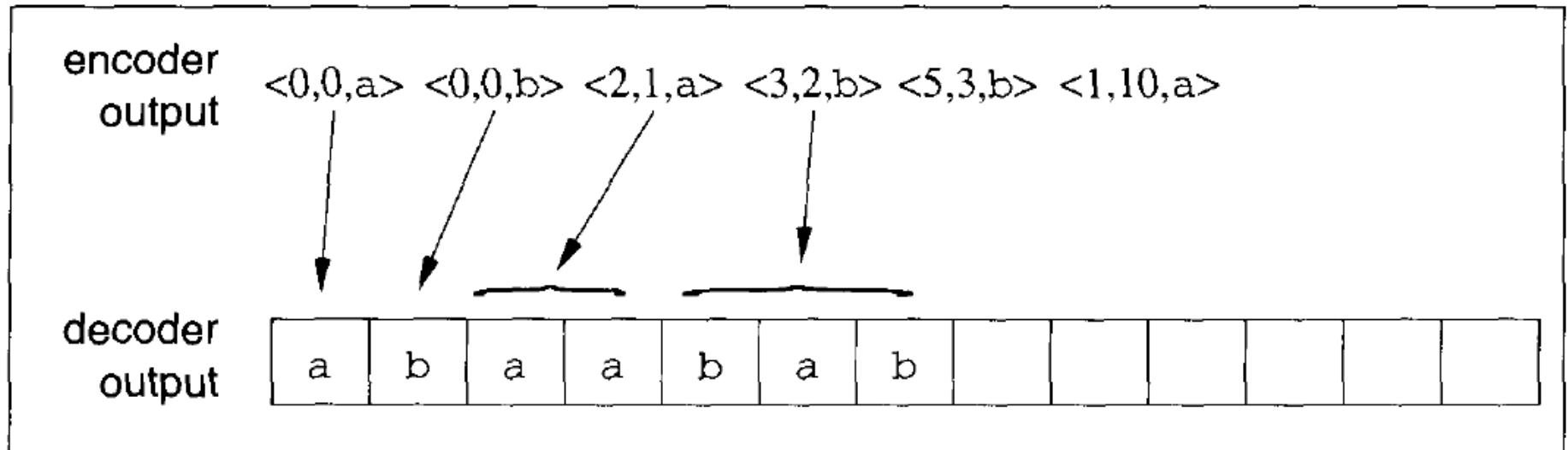


## 4. Compresión de texto: códigos LZ77 y LZ78

Una subcadena es remplazada por un puntero a una ocurrencia anterior.

- El codebook es esencialmente todo el texto anterior a la posición actual
  - Los codewords estan representados por punteros.
- 
- El texto previo constituye un buen diccionario, ya que usualmente es del mismo estilo y lenguaje que el texto a procesar.
  - El diccionario es transmitido implícitamente sin ningún coste adicional, ya que el decodificador tiene acceso al texto anterior.
  - Desarrollados por Jacob Ziv y Abraham Lempel

## 4. Compresión de texto: ejemplo código LZ77



**<posición, longitud, símbolo>**

- **Posición**: número de posiciones que hay que retroceder en el texto decodificado para encontrar la siguiente subcadena.
- **Longitud**: longitud de la subcadena referenciada.
- **Símbolo**: sólo es necesario cuando el símbolo a codificar no ha aparecido anteriormente.

El alfabeto contiene a's y b's.

# Algoritmos de codificación y decodificación LZ77

To encode the text  $S[1 \dots N]$  using the LZ77 method, with a sliding window of  $W$  characters,

1. Set  $p \leftarrow 1$ . /\* the next character of  $S$  to be coded \*/
2. While there is text remaining to be coded do
  - (a) Search for the longest match for  $S[p \dots ]$  in  $S[p - W \dots p - 1]$ . Suppose that the match occurs at position  $m$ , with length  $l$ .
  - (b) Output the triple  $\langle p - m, l, S[p + l] \rangle$ .
  - (c) Set  $p \leftarrow p + l + 1$ .

To decode the text  $S[1 \dots N]$  using the LZ77 method, with a sliding window of  $W$  characters,

1. Set  $p \leftarrow 1$ . /\* the next character of  $S$  to be decoded \*/
2. For each triple  $\langle f, l, c \rangle$  in the input do
  - (a) Set  $S[p \dots p + l - 1] \leftarrow S[p - f \dots p - f + l - 1]$ .
  - (b) Set  $S[p + l] \leftarrow c$ .
  - (c) Set  $p \leftarrow p + l + 1$ .

## 4. Compresión de texto: código LZ77

- La codificación LZ77 implica una búsqueda en la ventana de texto anterior de la secuencia más larga que coincide con la subcadena entrante.
- Una búsqueda lineal es demasiado costosa en tiempo y puede ser acelerada usando una estructura de datos adecuada: un trie, una tabla hash, o un árbol binario de búsqueda.
- El método de decodificación para LZ77 es muy rápido ya que cada símbolo decodificado requiere una búsqueda en un vector.
- El programa de decodificación es muy simple y puede ser incluido con los datos a muy bajo coste.



## 4. Compresión de texto: código LZ77

Sea la siguiente secuencia resultado de una codificación con códigos LZ77:

(0,0,\$) (0,0,érase) (2,1,un) (2,1,hombre) (2,1,a) (2,1,una)  
(2,1,nariz) (2,1,pegado) (14,2,\$) (8,3,\$) (0,0,superlativa)  
(8,7,sayón) (2,1,y) (2,1,escriba)

Se pide el texto que resulta de decodificar la secuencia anterior. Los elementos que se han utilizado como diccionario para codificar son palabras o secuencias de palabras (consideramos el separador “\$” como una entrada más en el diccionario).



## 4. Compresión de texto: código LZ77

Sea la siguiente secuencia resultado de una codificación con códigos LZ77:

(0,0,\$) (0,0,érase) (2,1,un) (2,1,hombre) (2,1,a) (2,1,una)  
(2,1,nariz) (2,1,pegado) (14,2,\$) (8,3,\$) (0,0,superlativa)  
(8,7,sayón) (2,1,y) (2,1,escriba)

Se pide el texto que resulta de decodificar la secuencia anterior. Los elementos que se han utilizado como diccionario para codificar son palabras o secuencias de palabras (consideramos el separador “\$” como una entrada más en el diccionario).

**Solución:**

\$érase\$un\$hombre\$a\$una\$nariz\$pegado\$érase\$una\$nariz\$superlativa\$érase  
\$una\$nariz\$sayón\$y\$escriba

## 4. Compresión de texto: código gzip

- Basado en los códigos LZ77, *Gzip* usa una tabla hash para localizar las ocurrencias previas de la subcadena.
- Se aplica la función hash a los tres siguientes símbolos a ser codificados, y el valor resultante es usado para buscar una entrada en la tabla.
- Esta entrada contiene una lista enlazada con las posiciones en las que aparecen esos tres caracteres en la ventana.
- Debido al rápido algoritmo de búsqueda y a la representación de salida basada en los códigos de Huffman, *gzip* supera en compresión y efectividad a la mayor parte de los métodos basados en Ziv-Lempel.