TEMA 7. CONSTRUCCIÓN Y COMPRESIÓN DE ÍNDICES

Contenidos basados en el material del curso de Manning

Contenidos

Este tema contiene dos grandes apartados, en el primero estudiaremos diferentes métodos de indexación y en el segundo estudiaremos métodos de compresión de datos.

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

Vamos a estudiar cómo construir un índice invertido.
Estudiaremos la indexación basada en ordenación de bloques, un algoritmo eficiente para un sólo ordenador diseñado para colecciones estáticas, en una versión más escalable que la introducida en el tema1.
Estudiaremos también la indexación en memoria de un paso, con mejores propiedades de escalabilidad que la anterior.
Estudiaremos la indexación distribuida en clusters de cientos o miles de ordenadores.
Finalmente, estudiaremos la indexación dinámica para aquellas colecciones que soportan frecuentes cambios.

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.

Vamos a utilizar una colección de documentos de referencia para ilustrar un caso concreto de colección sobre la que razonar las necesidades de memoria en el proceso de indexación.
Aunque es una colección de tamaño reducido nos sirve para el estudio que llevaremos a cabo en este tema.

2. La colección de documentos RCV1

REUTERS

You are here: Home > News > Science > **Article**

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enouc

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

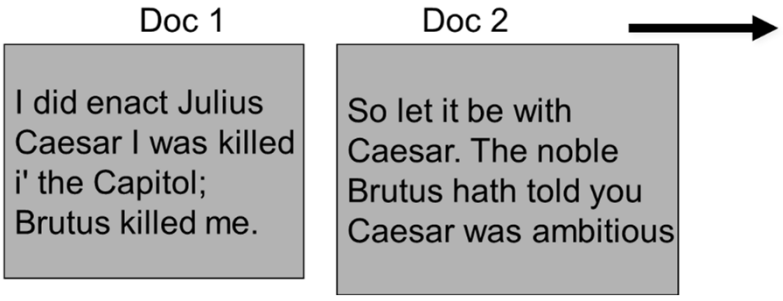
La diapositiva muestra los valores redondeados que se utilizarán para hacer las estimaciones.
Los valores reales son:
806,791 documentos,
222 tokens por documento,
391,523 términos distintos,
6.04 bytes por token, contando espacios y signos de puntuación,
4.5 bytes por token, sin contar espacios y signos de puntuación,
7.5 bytes por término, se incluye información adicional (frecuencias, etc.)
96,969,056 tokens no posicionales.
Tokens posicionales son realmente 179.158.000.

Estos valores se corresponden con los valores de la tercera línea de la tabla de compresión de índices (case-folding) para las columnas de términos y postings no posicionales que veremos en una diapositiva más adelante (41).

3. Indexación basada en ordenación de bloques

Recordemos, en la construcción de un índice:

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



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

Recordemos el método simple que estudiamos en un tema anterior para la construcción de un índice invertido a partir del análisis de los documentos.
Se analiza documento a documento y se extraen los pares término-docID.

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

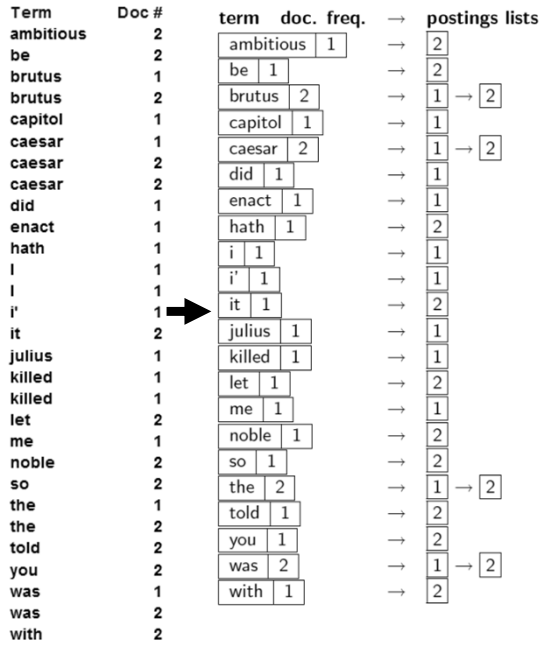
Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Posteriormente se ordena la lista anterior en base a los términos. Reuters-RCV1 tiene unos 100 millones de tokens no posicionales, por lo que en la construcción de un índice no posicional siguiendo esta estrategia de indexación habría que ordenar 100M de pares.

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.



Posteriormente se ordena la lista anterior en base a los términos. Reuters-RCV1 tiene unos 100 millones de tokens no posicionales, por lo que en la construcción de un índice no posicional siguiendo esta estrategia de indexación habría que ordenar 100M de pares.

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

La recopilación de todos los pares termID-docID de la colección utilizando 4 bytes para termID y docID, requiere 0,8 GB de almacenamiento.
Las colecciones típicas de hoy son varios órdenes de magnitud mayores que Reuters-RCV1.
Con memoria principal insuficiente, se necesita usar un algoritmo de indexación externo, es decir, uno que use disco.
Para una velocidad aceptable, el requisito central de dicho algoritmo es que minimice el número de accesos a disco durante la indexación.

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

Una solución es el algoritmo de indexación basado en ordenación de bloques (BSBI).
Para hacer que la construcción de índices sea más eficiente, se representan los términos como termIDs, donde cada termID es un número único de una serie.
Se puede construir la correspondencia de términos a termID sobre la marcha mientras se procesa la colección y guardarla en memoria.

Aplicando esta estrategia a la colección Reuters-RCV1 y asumiendo que se pueden ajustar 10 millones de termID-docID
pares en memoria, se necesitan 10 bloques, cada uno se convertirá en un índice invertido de una parte de la colección completa.

3. Indexación basada en ordenación de bloques: Algoritmo BSBI

```
BSBIndexConstruction()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4     block ← ParseNextBlock()
5     BSBI-Invert(block)
6     WriteBlockToDisk(block, f_n)
7  MergeBlocks(f_1, ..., f_n; f_merged)
```

El algoritmo BSBI se explica en las diapositivas siguientes.

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).

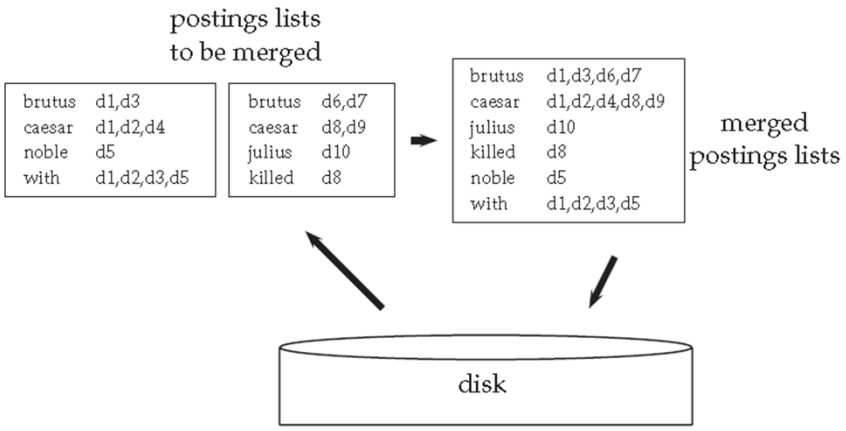
Cada bloque es procesado (BSBI-Invert) y almacenado en disco. Es importante que ese almacenamiento secuencial
se haga de forma ordenada en base al termID para poder realizar el posterior proceso de mezcla de forma sencilla,
como se explicará a continuación.

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



Un ejemplo con dos bloques se muestra en la figura, donde usamos ‘di’ para denotar el i-ésimo documento de la colección.
Para hacer la fusión, se abren todos los ficheros de bloque simultáneamente.
Se mantienen pequeños buffers de lectura para los diez bloques que se leen y un buffer de escritura para el
índice mezclado que se genera.
En cada iteración, se selecciona el term-ID más bajo que aún no ha sido procesado.
Todas las postings lists para este termID de todos los buffers de lectura se leen y se mezclan, y la lista mezclada se escribe en el buffer de escritura.

Cada buffer de lectura se rellena desde su fichero cuando es necesario.

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

- Su complejidad temporal es Θ(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.

El mejor algoritmo de ordenación trabaja en O(TlogT)

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

Otra solución para indexar usando memoria secundaria es el algoritmo de indexación en memoria de un paso (SPIMI).
En este método de indexación se utilizan directamente los términos.

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 ← next(token_stream)
 5     if term(token) ∉ 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 ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

El algoritmo se explica en las diapositivas siguientes.

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 (línea12).
- 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.

La gran diferencia respecto del método anterior estriba en que no se realiza la ordenación de la lista de pares término-docID que resulta del análisis de los documentos. Este método va tratando directamente cada par y realiza la inserción (o localización) en el diccionario y en la postings list correspondiente sobre la marcha.
Es por esto que se ahorra el costoso proceso de ordenación de los pares.

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

El proceso de indexación de una gran colección de documentos no puede ser abordada con un único ordenador.
Vamos a hablar de indexación distribuida.

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.

Como resultado de una indexación distribuida se obtiene un índice distribuido entre varios ordenadores.
Esta distribución se realiza en base a los términos o en base a los documentos.
En nuestro caso, construiremos un índice distribuido particionado en base a términos.
Dispondremos de un nodo que asigna tareas, el llamado nodo maestro, y del resto de nodos del cluster.

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

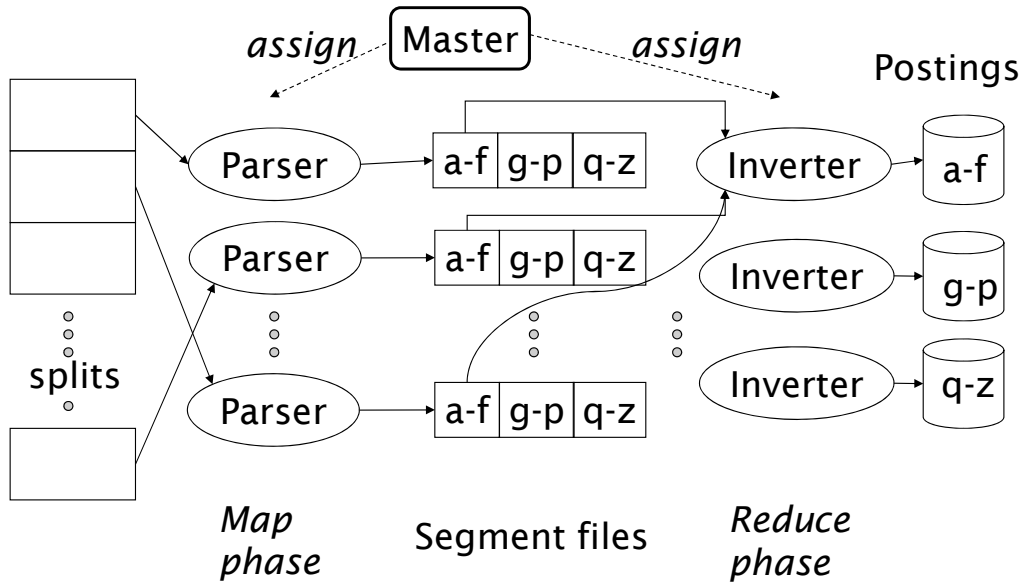
- El nodo maestro asigna un bloque a una máquina desocupada.
- El analizador lee el documento a su vez y emite los pares (término, documento).
- El analizador escribe los pares en j particiones.
- Cada partición corresponde a un rango de la primera letra del término
  - (e.g., ***a-f, g-p, q-z***) en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

Como hemos indicado, vamos a aplicar una partición en base a los términos.
Se definen 'j' particiones en el conjunto de términos, en este caso j=3, y la salida del proceso de análisis
se escribe en cada una de estas particiones atendiendo al término del par analizado.
Por su parte el proceso de construcción de índices recoge todos los pares para una determinada partición y genera el índice invertido.

5. Indexación distribuida



Descripción del proceso de análisis e indexación distribuido.
Como hemos explicado, cada 'inverter' toma la entrada de la misma partición del diccionario de términos de cada una de las salidas de todos los analizadores.
Con ello, el índice invertido que construye solamente contiene términos dentro de esa partición del diccionario.

MapReduce (una arquitectura para computación distribuida)

1. La fase *map* construye la correspondencia entre los datos de entrada y los pares clave-valor (termID,docID). Cada *analizador* escribe la salida en ficheros intermedios, los *segment files* de la figura.
2. En la fase *reduce* se persigue que los valores para una determinada clave se almacenen cerca. Para ello se distribuye las claves en j particiones y se guardan los pares para cada partición en ficheros separados.
3. En la tarea de la inversión de índices se recuperan todos los valores (docID) para una clave dada (termID). Esto se lleva a cabo en la fase *reduce*

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.

Un factor a considerar en el mantenimiento de un índice invertido para una determinada colección de documentos es su carácter dinámico. Tanto el diccionario como las postings lists han de ser actualizados cuando la colección de documentos se actualiza.

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).
- La 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.

Una solución sencilla consiste en mantener dos índices en paralelo de forma que el índice auxiliar pequeño
recoge las actualizaciones más recientes.
La resolución de una consulta implica el acceso a ambos índices y una mezcla de los dos resultados.
Esta solución comporta una mezcla periódica de ambos índices.

El coste de la construcción de este índice es $O(T^2/n)$ donde n es la talla del índice auxiliar y T el número total de postings.

6. Indexación dinámica: mezcla logarítmica

- Mantener una serie de índices (I_0,I_1,I_2, …), cada uno el doble de grande que su antecesor (2^0xn, 2^1xn, 2^2xn, …)
  - 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, …) en disco
- Si Z_0 se pasa de la talla máxima (> *n*),
  - escribe los 2^0xn postings en disco como I_0
  - o es mezclado con I_0 (si I_0 ya existe) y crea Z_1 de talla 2^1xn
  - Se escribe Z_1 en disco como I_1 (si no existe I_1)
  - o se mezcla con I_1 para formar Z_2
  - …

Podemos hacerlo mejor introduciendo log_2 (T / n) índices: I_0, I_1, I_2,. . .
del tamaño que se indica en la diapositiva, donde n es un valor fijo que indica el tamaño del índice auxiliar Z_0 que se mantiene en memoria, y los tamaños de los índices que se mantienen en disco son proporcionales a n en diferentes potencias de 2.
Se definen también Z_i (i>1) que son índices temporales que se almacenan en disco.

6. Indexación dinámica: mezcla logarítmica

```
LMERGEADDTOKEN(indexes, Z_0, token)
 1  Z_0 ← MERGE(Z_0, {token})
 2  if |Z_0| = n
 3    then for i ← 0 to ∞
 4         do if I_i ∈ indexes
 5               then Z_{i+1} ← MERGE(I_i, Z_i)
 6                    (Z_{i+1} is a temporary index on disk.)
 7                    indexes ← indexes − {I_i}
 8               else I_i ← Z_i     (Z_i becomes the permanent index I_i.)
 9                    indexes ← indexes ∪ {I_i}
10                    BREAK
11         Z_0 ← ∅

LOGARITHMICMERGE()
1  Z_0 ← ∅     (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z_0, GETNEXTTOKEN())
```

El algoritmo de mezcla logarítmica va llamando de forma indefinida al procedimiento LMERGEADDTOKEN
que se encarga de hacer la inserción en el índice.
GETNEXTTOKEN en este contexto token hace referencia a una entrada en el índice invertido, es decir, un término y su postings list.

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 $I_0, \ldots, I_3$ está en uso.

Veamos un ejemplo de aplicación del algoritmo de mezcla logarítmica.

Ejercicio

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

Cada fila indica qué índices en memoria secundaria están activos cuando se ha procesado una lista de n (n=2 en este ejemplo) tokens adicionales, es decir, cuando el índice Z_0 en memoria principal se ha llenado.

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

Solución del ejemplo.
Cada fila indica qué índices en memoria secundaria están activos cuando se ha procesado una lista de n (n=2 en este ejemplo) tokens adicionales, es decir, cuando el índice Z_0 en memoria principal se ha llenado.

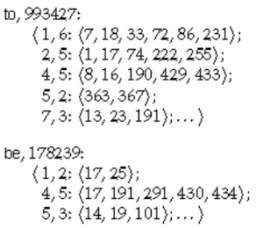
6. Indexación dinámica: comparación

- 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.

Con la aproximación simple cada posting se toca en cada mezcla entre índice principal e índice auxiliar, el coste es $O(T^2/n)$ donde n es la talla del índice auxiliar y T el número total de postings.

Con la mezcla logarítmica el coste es
$\Theta(T \log(T/n))$ ya que cada posting es procesado solamente una vez en cada uno de los $\log(T/n)$ niveles.

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.

- 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.

Este tema describe la construcción de índices NO posicionales.
Excepto por el volumen de datos mucho más grande que necesitamos acomodar, la principal diferencia para los índices posicionales es que hay que insertar tripletes (termID, docID, (position1, position2,...))
en lugar de pares (termID, docID)
y que los postings contienen información posicional además de los docID.
Con este cambio, los algoritmos discutidos aquí pueden aplicarse a los índices posicionales.

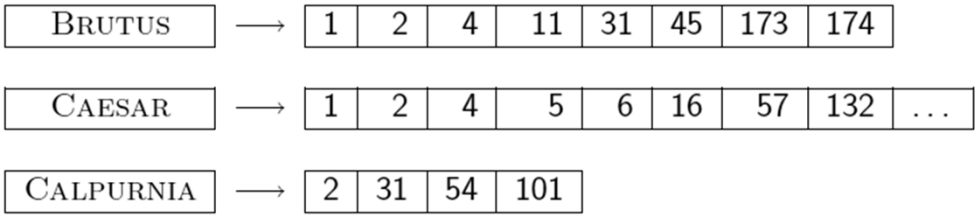
Adicionalmente, podemos añadir filtros a los resultados de una consulta para implementar restricciones de acceso a
las colecciones de documentos. Para ello se construye una matriz usuario-documento de forma que para cada usuario se indica la lista de documentos a los que puede acceder.

COMPRESIÓN DE ÍNDICES

2.1 Introducción
2.2 Compresión de diccionarios
2.3 Compresión de los ficheros de postings
2.4 Compresión de texto

En el tema 1 se introdujo el índice invertido como la estructura de datos central en RI.
En este tema vamos a estudiar técnicas para la compresión de esta estructura de datos.
Se pueden alcanzar ratios de compresión de 1:4, lo que comporta un ahorro en el almacenamiento del índice de 75%.

1. Introducción



- 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.

Con la compresión podemos almacenar en memoria principal más información.
Los algoritmos de descompresión funcionan tan rápidos sobre el hardware actual que el tiempo total de transferencia de un chunk comprimido de datos desde disco y una posterior descompresión resulta menor que transferir el mismo chunk de datos en formato descomprimido.
De forma que almacenar en disco las postings lists comprimidas es una buena práctica.

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)	Δ%	cumul %	Size (K)	Δ %	cumul %	Size (K)	Δ %	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

Estudiaremos el ejemplo de la colección RCV1.
En la primera columna se describe el proceso realizado sobre los términos
de la colección: ningún proceso (unfiltered), eliminación de números (no numbers), eliminación de mayúsculas (case folding), eliminación de 30 stopwords, eliminación de 150 stopwords y stemming.

La segunda columna contiene el tamaño del diccionario tras el proceso indicado en la fila, así como el incremento relativo y absoluto.
La tercera columna contiene la misma información para los postings no posicionales y la cuarta para los posicionales.
Los tamaños están expresados en miles (K).
Las diapositivas siguientes describen la tabla detalladamente.

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
- "Δ%" 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".

El número de términos es el factor principal para determinar el tamaño del diccionario (columna 2).
El número de postings no posicionales (columna 3) es un indicador del tamaño esperado del índice no posicional de la colección.
El tamaño esperado de un índice posicional está relacionado con el número de posiciones que debe codificar (columna 4).

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.

Aunque la eliminación de stop words reduce bastante el número de postings, esta reducción se puede compensar usando técnicas de compresión de índices, ya que las palabras muy frecuentes necesitan poca memoria para almacenar los “gaps” (se verá más adelante).

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

La segunda edición del Oxford English Dictionary (OED) define más de 600,000 palabras.
Pero el vocabulario de las colecciones más grandes es mucho más grande que el OED.
El OED no incluye la mayoría de los nombres de personas, ubicaciones, productos o elementos científicas como los genes.
Estos nombres deben incluirse en el índice invertido, para que los usuarios puedan buscarlos.

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?

Vamos a estudiar dos leyes empíricas que nos ayudarán a hacer una estimación del número de términos esperados en una colección de documentos así como de la longitud de las postings lists.

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 ½.

La ley de Heaps es útil para la estimación del tamaño del diccionario de términos de la colección.

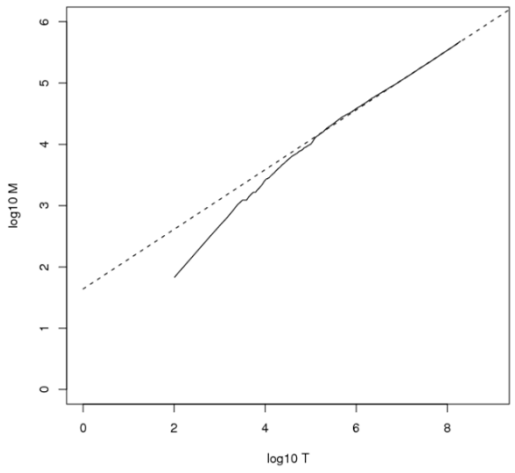
1. Introducción: vocabulario versus talla de la colección, la ley de Heaps

- 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



Resulta una ley empírica muy ajustada para Reuters RCV1.
De hecho para $T > 10^5 = 100{,}000$, y para los valores $b = 0.49$ y $k = 44$, se ajusta bien.

El parámetro k es bastante variable ya que el crecimiento del vocabulario depende mucho de la naturaleza de la colección de documentos.

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

Como estamos trabajando con la misma colección de documentos, las constantes K y b, que dependen de la colección, son las mismas para ambas.

Aplicaremos la ley de Heaps con esas constantes a la muestra y a la colección.

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 x n, y en la colección es, T=1.000.000 x 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

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

Conocemos el número de documentos en la muestra y en la colección, pero necesitamos el número de tokens.

Expresamos T y T_m en función de un número medio de palabras por documento n, que asumimos es el mismo para la muestra y la colección.

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

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

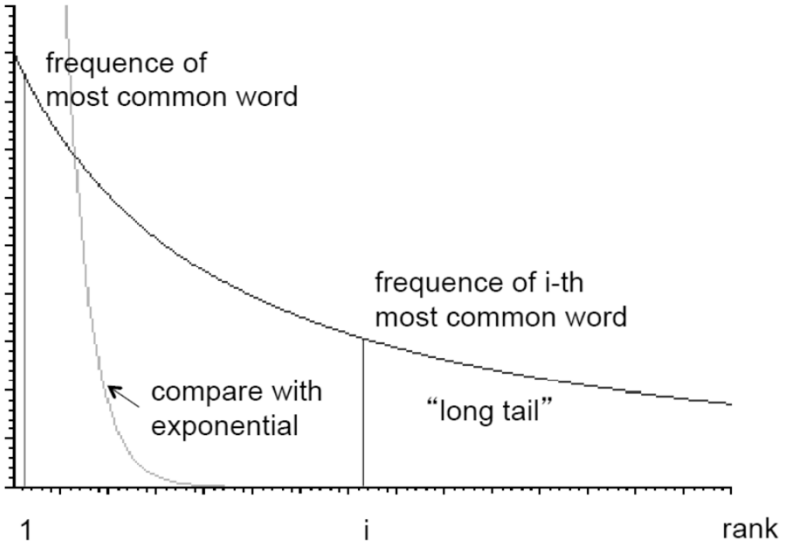
- En lenguaje natural, hay unos pocos términos muy frecuentes y muchos términos que aparecen con baja frecuencia.
- Ley de Zipf: El *i*ésimo término más frecuente tiene una frecuencia proporcional a 1/*i* .

$$cf_i \propto 1/i$$

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

La ley de Zipf puede ayudar en la estimación de los tamaños de las postings lists.

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



La distribución de frecuencias de frecuencias suele seguir la forma conocida como 'long tail', con un número muy grande de elementos con frecuencias muy bajas.

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:

$$cf_i = K/i$$

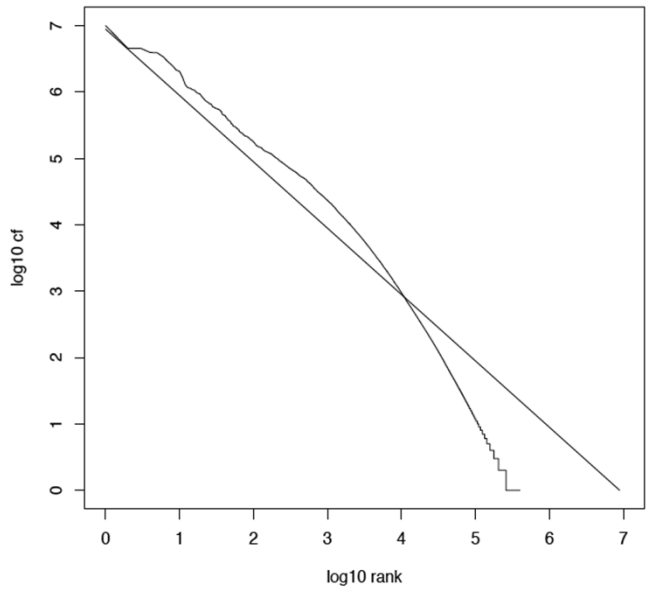
donde K es un factor de normalización,

- Por tanto:

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

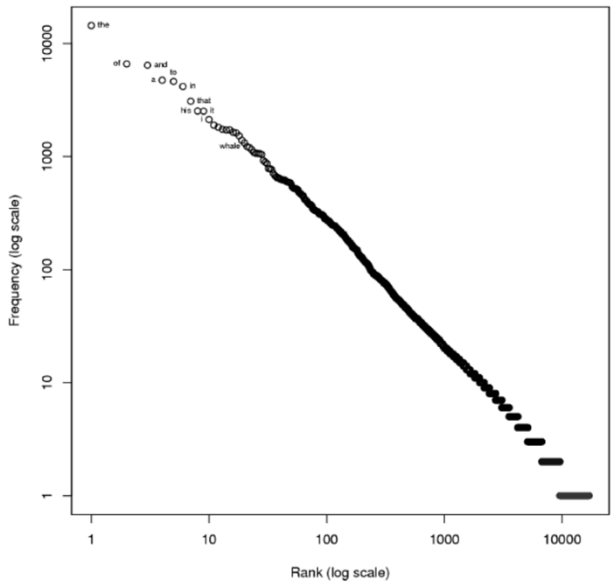
$\Rightarrow$ hay una relación lineal entre $\log cf_i$ y $\log i$

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



Se muestra cómo se ajusta la colección Reuters RCV1 a la recta que describe la ley de Zipf.
En el eje horizontal se representan las postings lists ordenadas de mayor a menos longitud, y en el eje vertical se representan esas longitudes. En ambos ejes se han tomado logaritmos.

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"]

Esta es la gráfica para la novela Moby Dick.

1. Introducción: ley de Zipf, ejercicio

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

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

Vamos a resolver un ejercicio relacionado con la ley de Zipf.

1. Introducción: ley de Zipf, ejercicio

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

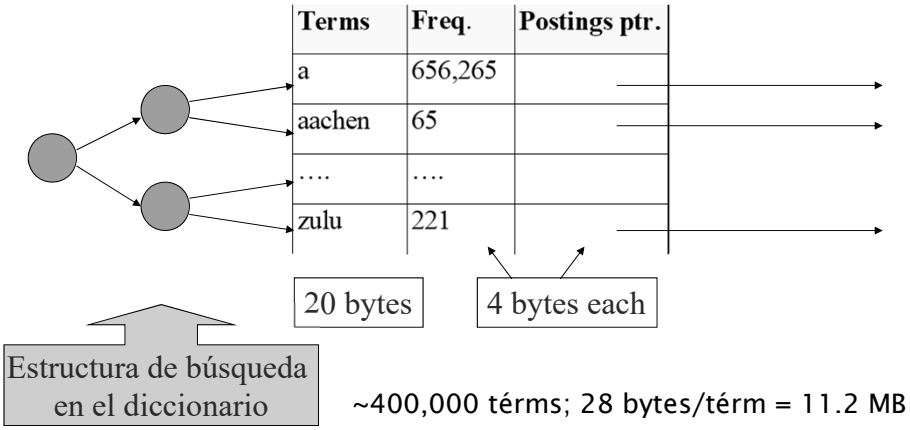
La ley de Zipf: $cf_i = K/i^2$

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:

$100 = K / i^2$, 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



La estructura de datos más simple para el diccionario es realizar una ordenación lexicográfica del vocabulario y almacenarlo en un array de talla fija (fixed-width storage).

Para el inglés se necesita 20 bytes para el término (ya que muy pocos términos en inglés tienen más de 20 caracteres),

4 bytes para su frecuencia de documento y 4 bytes adicionales para el puntero a su postings list.

Punteros de 4 bytes permiten direccionar 4 gigabytes (GB).

Para grandes colecciones de documentos se necesita reservar más bytes por puntero.

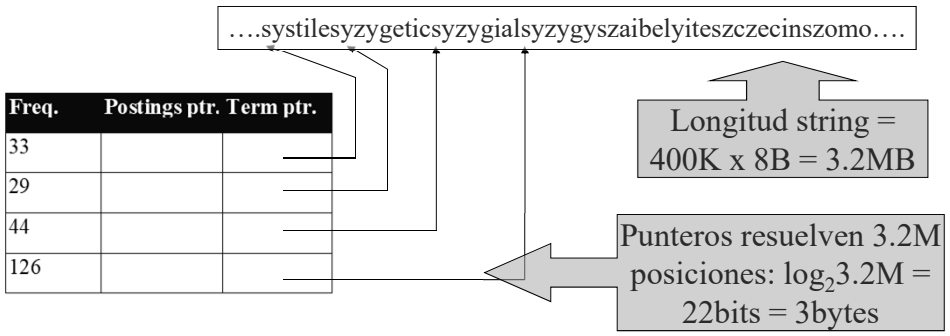
Se localizan los términos en el array a través por ejemplo de búsqueda binaria.

Para Reuters-RCV1, se necesitan $M \times (20 + 4 + 4) = 400{,}000 \times 28 = 11.2$ megabytes (MB) para almacenar el diccionario con este esquema.

2. Compresión de diccionarios: un string de caracteres

El puntero a la palabra siguiente muesta el final de la actual

Esperamos ahorrar hasta un 1/3 de espacio en el diccionario.



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

La longitud media de los términos en inglés es de unos 8 caracteres, por lo que se está malgastando una media de 12 caracteres con el esquema anterior. Un esquema alternativo para almacenar el diccionario consiste en guardar los términos ordenados lexicográficamente como una única secuencia, un string largo de caracteres, tal como muestra la diapositiva. El puntero al próximo término marca el final de término anterior. Los primeros términos en el ejemplo son *systile*, *syzygetic*, y *syzygial*.

Con este esquema, el array es mucho menor, ahorra un 60% comparado con el esquema anterior (fixed-width storage). Se usan en promedio – 12 bytes sobre los 20 bytes por término. Sin embargo, se necesita almacenar adicionalmente los punteros a los términos.

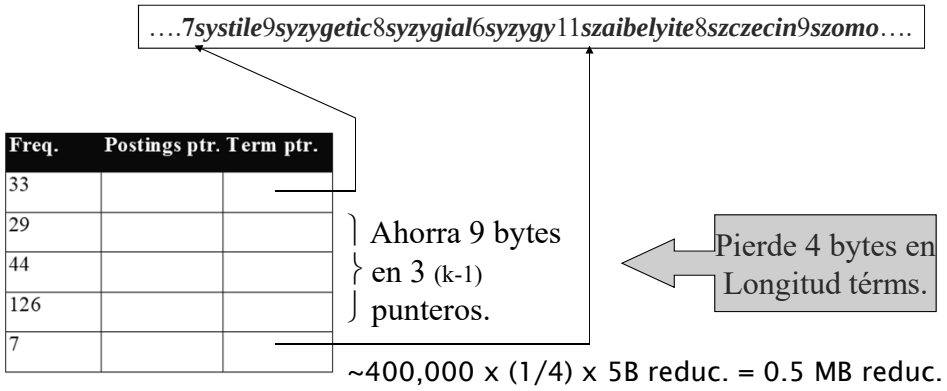
Estos punteros direccionan $400{,}000 \times 8 = 3.2 \times 10^6$ posiciones, por lo que deben ser de $\log_2 (3.2 \times 10^6) \approx 22$ bits o 3 bytes.

Con este esquema se necesitan $400{,}000 \times (4 + 4 + 3 + 8) = 7.6$ MB para el diccionario Reuters-RCV1: 4 bytes para la frecuencia y para el puntero de la postings list, 3 bytes para el puntero al término, y 8 bytes en promedio por término. Se consigue una reducción espacial de un tercio, de 11.2 a 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).



Se puede comprimir más la anterior representación del diccionario agrupando los términos en el string en bloques de talla *k* y usando un único puntero por bloque.

Se almacena también la longitud de cada término con un byte adicional al principio del mismo.

Se eliminan $k - 1$ punteros a términos, pero se necesitan k bytes adicionales para almacenar las longitudes de los términos.

Para $k = 4$, se ahorra $(k - 1) \times 3 = 9$ bytes, pero se necesitan $k = 4$ bytes para las longitudes.

El requerimiento total de espacio para con este esquema para el diccionario de Reuters-RCV1 se reduce en 5 bytes por cada bloque, es decir, un total de $400{,}000 \times 1/4 \times 5 = 0.5$MB. El requerimiento especial es 7.1 MB.

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***10***automation***

→8***automat*****a***1◊***e***2◊***ic***3◊***ion***

Codifica ***automat***

Longitud extra
Sobre ***automat.***

Una fuente de redundancia en el diccionario es el hecho de que entradas consecutivas en una listado ordenado lexicográficamente comparten prefijos. Esta observación lleva a una nueva alternativa: *front coding*.

Un prefijo común es identificado por una secuencia acabada con un carácter especial ('*' en el ejemplo).

En el caso de la colección Reuters, front coding ahorra 1.2 MB adicionales.

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

Resumen de la reducción en las necesidades espaciales para el diccionario.

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 (100Mx20b/8)

Hacemos un cálculo de cuánto necesitamos para almacenar cada docID, suponiendo que es un valor entre 0 y 800.000, que es el tamaño de la colección.
En este apartado, cuando hablemos de posting nos referiremos realmente a su contenido, el docID.

3. Compresión de los ficheros de postings

- Un término como ***arachnocentric*** ocurre tal vez en un documento de un millón. ⇒ 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.

Para el caso de términos poco frecuentes no parece problemático usar 20 bits por posting.
Sin embargo, cuando un término aparece en muchos documentos, su postings lists es muy larga, por lo que usar esa memoria para cada uno de sus postings no parece una buena solución.

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 …
- ⇒ Sería suficiente almacenar los *gaps*.
  - 33,14,107,5,43 …
- Siempre que almacenar los gaps sea más barato que 20 bits.

Con el objetivo de encontrar una representación más eficiente de las postings lists, una que utilice menos de 20 bits por posting, observamos que las postings de términos frecuentes están muy juntas.

Por ejemplo en el caso del término muy frecuente como ‘computer’ podemos encontrar segmentos de la postings list en los que los postings son números muy cercanos.
De ahí la idea de almacenar los gaps entre postings consecutivos, en lugar del valor del docID.

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.

La idea clave es que los gaps entre las postings son cortos y se requiere mucho menos espacio que 20 bits para almacenarlos.
De hecho, los gaps para los términos más frecuentes como "el" y "para" son en su mayoría iguales a 1.
Los gaps para términos poco frecuentes serán sin embargo tan grandes como los valores absolutos.

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 ~$\log_2 G$ 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.

Además de codificar gaps en lugar de postings, la codificación que se va a usar pretende utilizar el menor número de
bits posible. Es lo que se conoce como codificación de longitud variable. Vamos a estudiar un par de métodos de codificación: codificación variable en bytes y códigos gamma.

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 $c = 0$.

Vamos a estudiar la codificación variable en bytes.
Trata de utilizar el menor número de bytes posible para codificar el gap.
De cada byte, se utilizan los 7 bits del final para codificar.
Con 7 bits podemos codificar desde el valor 0 a 127.
El bit más significativo se utiliza como bit de continuación.

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
0000011010111000100001010000110100001100101100 01

La codificación VB de postings es decodificable sin ambigüedad

Para un gap pequeño (5), usa un byte.

Para decodificar un código variable en bytes, leemos una secuencia de bytes con el bit de continuación a 0,
terminado con un byte con el bit de continuación 1.
Esta secuencia de bytes determina la codificación de un gap.
Luego extraemos y operamos con las partes de 7 bits restantes.
Con el uso del bit de continuidad la decodificación NO es ambigua.
Como vemos en el ejemplo, los números bajos requieren menos bytes que los números más altos.

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

```
VBENCODENUMBER(n)
1  bytes ← ⟨⟩
2  while true
3  do PREPEND(bytes, n mod 128)
4     if n < 128
5        then BREAK
6     n ← n div 128
7  bytes[LENGTH(bytes)] += 128
8  return bytes

VBENCODE(numbers)
1  bytestream ← ⟨⟩
2  for each n ∈ numbers
3  do bytes ← VBENCODENUMBER(n)
4     bytestream ← EXTEND(bytestream, bytes)
5  return bytestream
```

Algoritmo de codificación.
La función ‘div’ hace referencia a la división entera y la función ‘mod’ al resto de la división entera.

PREPEND añade un elemento al inicio de una lista,
PREPEND(<1, 2>, 3) = <3, 1, 2>.
EXTEND concatena listas
EXTEND(<1,2>, <3, 4>) = <1, 2, 3, 4>.

La operación +128 pone el bit de control del último byte a 1

Ejemplo:
si n=824 entonces bytes=<6, 56+128>
si n=5 entonces bytes=<5+128>
si n=214.577 entonces bytes=<13, 12, 49+128>

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

```
VBDECODE(bytestream)
1  numbers ← ⟨⟩
2  n ← 0
3  for i ← 1 to LENGTH(bytestream)
4  do if bytestream[i] < 128
5        then n ← 128 × n + bytestream[i]
6        else n ← 128 × n + (bytestream[i] − 128)
7             APPEND(numbers, n)
8             n ← 0
9  return numbers
```

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

Algoritmo de decodificación.
Cuando el bit de control está puesto a 1, el bytestream(i)>=128, por lo que hay que restar 128 al resultado.

Con este método de compresión, la talla del índice de la colección Reuters-RCV1 es 116MB verificado experimentalmente.
Esto es más de un 50% de reducción respecto del índice no comprimido.

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

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

Ejercicio de decodificación.
Tanto al codificar como al decodificar hay que tener en cuenta que se utilizan los gaps.
Por ello, cuando codificamos una secuencia de postings, en primer lugar la convertiremos en una secuencia de gaps, el primer valor será el mismo posting, pero para el resto se obtendrán los gaps. A partir de esos valores codificaremos.

En el caso de la decodificación, se procederá a decodificar la secuencia de bytes obteniendo la secuencia de gaps.
Una vez decodificada se traducirá a la secuencia de postings.

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

Se pide decodificar la siguiente secuencias de bits **c**odificada con codificación variable (CV) en bytes**:**
000000111000010010001010

Solución:

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

La postings list seria: 388, 398

Hemos espaciado los diferentes bytes para mayor claridad.
Los dos primeros bytes representan un único valor de gap ya que el bit de continuación del primer byte está puesto a 0 y el del segundo a 1.

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 lider eliminado
  - Por ejemplo 13 → 1101 → 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

Otro método de compresión, esta vez ajustando a nivel de bit: códigos gamma.
Asumiendo que los valores del gap G con $1 \leq G \leq 2^n$ son equiprobables, la codificación óptima usa *n* bits para cada *G*.

Un código gamma es decodificado primero interpretando el código unario hasta el primer 0, en el ejemplo los primeros 4 bits, 1110, cuando decodificamos 1110101.
De esa forma sabemos cuántos bits hay que consumir tras el 0, es decir, la longitud del offset: 3 bits en el ejemplo.
El offset 101 es, al que se le concatena el bit líder 1 eliminado en la codificación: 101→1101 = 13.

3. Compresión de los ficheros de postings: códigos gamma

number	length	offset	γ-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

Ejemplos de codificación gamma; en la columna de la derecha se ha conservado una coma separadora de la longitud y el offset que realmente no aparece en el código, se ha conservado por claridad.

El valor 0 no se codifica, no vamos a encontrar ningún gap que valga 0.

El valor 1 se codifica como un 0, el separador.

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]

Ejercicio

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

El primer paso para codificar una secuencia de postings es obtener la secuencia de gaps.
El primer posting se conserva y el resto se convierte en gaps.
El resultado sería la secuencia de códigos gamma que se obtiene concatenando los códigos de la tabla en orden de arriba a abajo.

3. Compresión de los ficheros de postings: códigos gamma

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

1011001110001110011110010 0

Ejercicio

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

Separamos la secuencia de bits en segmentos para mayor claridad, interpretando de forma no ambigua las longitudes.
Decodificamos y posteriormente convertimos la secuencia de gaps en la secuencia de postings.

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)

La longitud del *offset* es $\lfloor \log_2 G \rfloor$ bits y la de *length* es $\lfloor \log_2 G \rfloor + 1$ bits, de forma que la longitud del código completo es $2 \times \lfloor \log_2 G \rfloor + 1$ bits. Estos códigos siempre tienen longitud impar.
Presentan la característica de ser *prefix free*, es decir, ningún código es prefijo de otro. Ello permite una decodificación NO ambigua.
La codificación gamma alcanza buenos ratios de compresión, cerca de 15% mejor que la codificación variable en bytes para la colección Reuters-RCV1. Sin embargo, su decodificación es más costosa, ya que requiere operaciones a nivel de bit.

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, γ–encoded	101.0

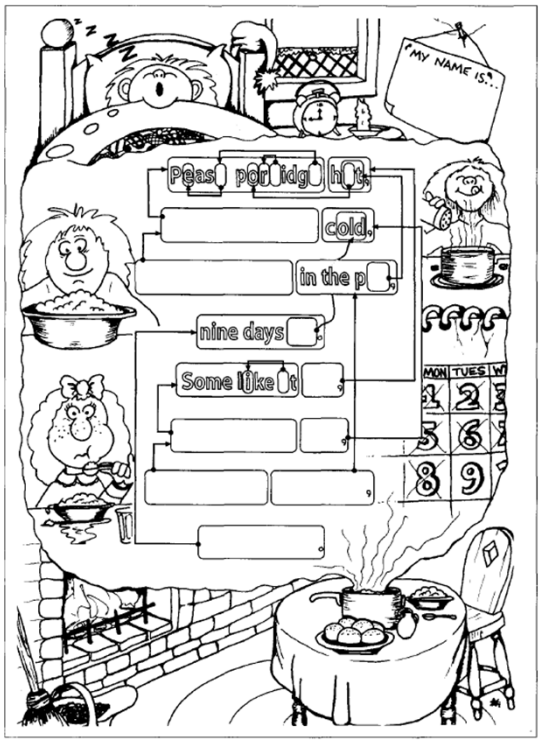
La tabla resume los resultados de aplicar diferentes técnicas de compresión a la colección Reuters-RCV1, tanto sobre el diccionario de términos, como sobre las postings lists.
La matriz de incidencia para esta colección tiene una talla $400{,}000 \times 800{,}000 = 40 \times 8 \times 10^9$bits, 40GB.

Usando los mejores métodos de compresión, códigos gamma para los postings y front coding para el diccionario,
a ratio de compresión es para Reuters-RCV1: $(101 + 5.9)/3600 \approx 0.03$.

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.

Por ultimo, vamos a abordar diferentes técnicas clásicas de compresión de texto.
Para ficheros de audio o imágenes puede plantearse aplicar métodos de compresión con pérdidas, de hecho, una representación digital de una forma de onda analógica siempre comporta pérdidas. Sin embargo, en el caso de la codificación de texto debemos exigir que el método de compresión permita reconstruir fielmente el fichero original.



La figura muestra un ejemplo de un sistema de compresión que representa una técnica muy común en informática.
En la figura, el texto omitido puede ser reconstruido siguiendo las flechas hacia atrás donde el texto ha ocurrido anteriormente. Por ejemplo, la primera caja vacía debe contener la letra ‘e’.
La primera caja vacía de la segunda línea se refiere a la secuencia *‘pease porridge’* de la primera línea. Como las flechas van hacia atrás, el texto original puede ser reconstruido procesando de izquierda a derecha el texto codificado.
Las flechas, punteros, se representan con dos números que indican dónde ha ocurrido el texto y cuántos caracteres deben ser copiados. Estos punteros usualmente ocupan menos espacio de almacenamiento que los trozos que sustituyen, por lo que se consigue la compresión.
Para la transmisión de datos también es interesante el uso de este tipo de técnicas ya que se requiere menos tiempo de transmisión.

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.

El ejemplo de la diapositiva anterior pertenece a los métodos basados en diccionarios. En el ejemplo el diccionario
está constituido por el mismo texto.
Vamos a estudiar un método de los de tipo simbólico y otro de los basados en diccionarios.

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.

Códigos de Huffman es un ejemplo de método simbólico

4. Compresión de texto: código de Huffman

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

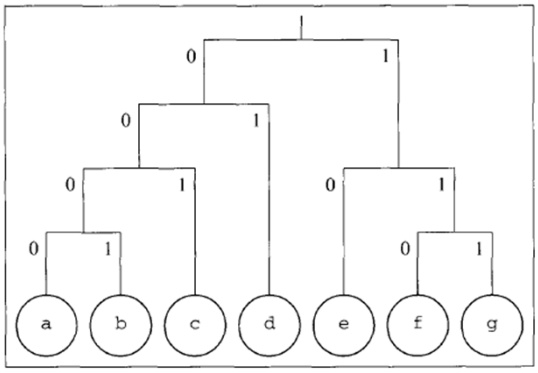
Symbol	Codeword	Probability
a	0000	0.05
b	0001	0.05
c	001	0.1
d	01	0.2
e	10	0.3
f	110	0.2
g	111	0.1

La secuencia *eefggfed* es codificada como 10101101111111101001

Notación:
Codeword: cada secuencia que se utiliza para codificar un símbolo.
Codebook: el diccionario de codewords

La tabla muestra el código Huffman para un alfabeto de siete símbolos: {a, *b,* c, *d, e,* f , *g}.*
Una secuencia se codifica reemplazando cada uno de sus símbolos por el codeword correspondiente según la tabla.
Por ejemplo, la secuencia *eefggfed* es codificada como 10101101111111101001.
La decodificación se lleva a cabo de izquierda a derecha.
La entrada al decodificador empieza por 10 . . . , y el único codeword que empieza por 10 es el del símbolo ‘e’,
que por tanto será decodificado, y la decodificación continuará con el resto.
Los códigos Huffman son también *prefix free*, por lo tanto su decodificación es NO ambigua.

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*)

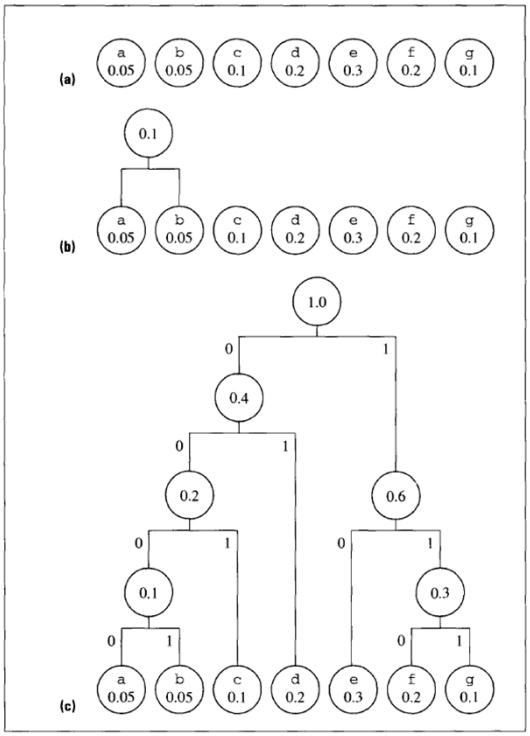
En la figura se muestra un árbol que puede ser usado para la decodificación.
El árbol se recorre desde la raíz y siguiendo la rama según el bit siguiente a decodificar. El camino de la raíz a una hoja, un símbolo, corresponde al codeword del símbolo según la tabla anterior.
Aquí se puede observar claramente cómo ningún codeword es prefijo de ningún otro.
Cuando se ha recorrido el camino desde la raíz a una hoja y se ha decodificado un codeword, el proceso empieza otra vez en la raíz para decodificar el siguiente.

Construcción del *árbol de Huffman*:

(a) Nodos hoja

(b) Combinación de nodos

(c) El árbol completo



El algoritmo de Huffman construye el árbol de decodificación. En la figura se ilustra su construcción para el ejemplo anterior.
Empieza creando una hoja para cada símbolo en la que se almacena el símbolo y su probabilidad (Figura a).
Los dos nodos con menor probabilidad, ‘a’ y ‘b’, pasan a ser hermanos y se inserta un padre común cuya probabilidad es la suma de las probabilidades de sus hijos (Figura b).
La operación de combinación de nodos de dos en dos continúa con los nodos de menor probabilidad activos, excluyendo los que ya son hijos. Por ejemplo, en el siguiente paso se combina el nuevo nodo creado en el paso anterior con el nodo de la ‘c’ creando un nuevo nodo padre con probabilidad 0,2.
El proceso continúa hasta que queda un único nodo que será el nodo raíz del árbol de decodificación (Figura c).
Las dos ramas de cada nodo no-hoja se etiquetan con un 0 i un 1.

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.

El algoritmo de Huffman.
Se define un conjunto *T* que contendrá recursivamente otros conjuntos.
Cada subconjunto representará un nodo en el árbol.
Cuando el algoritmo acaba, *T* contiene un conjunto, que a su vez contiene dos conjuntos, etc.

La codificación de Huffman es en general rápida tanto en la codificación como en la decodificación ya que parte de que la probabilidad asignada a los símbolos es estática.
Existen versiones de este algoritmo que ajustan el árbol para adaptarlo a cambios en la distribución de probabilidad.

Código de Huffman: ejemplo

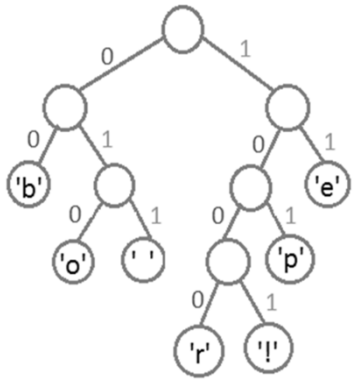
Character	Frequency
‘b’	3
‘e’	4
‘p’	2
‘ ‘	2
‘o’	2
‘r’	1
‘!’	1

Ejemplo.

Sea el siguiente conjunto de símbolos y sus frecuencias. Podemos trabajar directamente con las frecuencias o pasarlas a probabilidades haciendo un proceso de normalización. El resultado del algoritmo será el mismo ya que la ordenación relativa resultante es la misma.

Código de Huffman: ejemplo

Char	Freq
‘b’	3
‘e’	4
‘p’	2
‘ ‘	2
‘o’	2
‘r’	1
‘!’	1



Ejemplo.
Este sería un posible árbol de decodificación de Huffman construido siguiendo el algoritmo anterior.

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

Ejemplo de codificación que compara una representación binaria de los caracteres usando 8 bits con la obtenida con la codificación de Huffman de la diapositiva anterior.
En la representación en binario se usan 8 bits por símbolo: 8x15 símbolos = 120 bits
Con los códigos de Huffman se necesitan para la misma cadena 40 bits

Código de Huffman: ejercicio

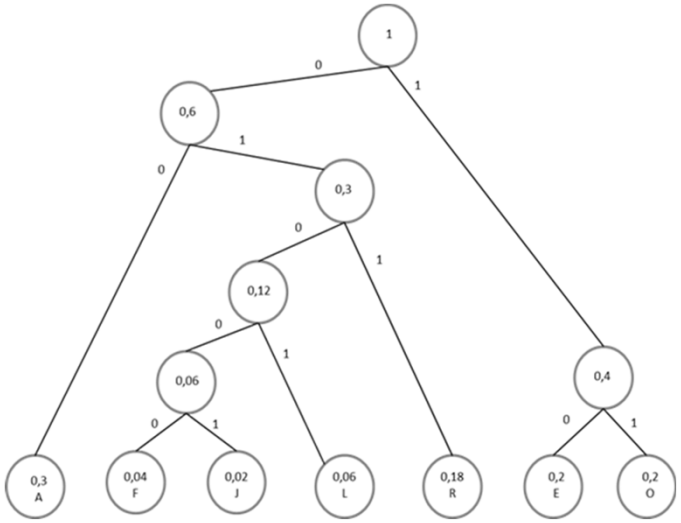
1. Dada la siguiente lista de símbolos y sus probabilidades se pide construir el **árbol de Huffman**

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”

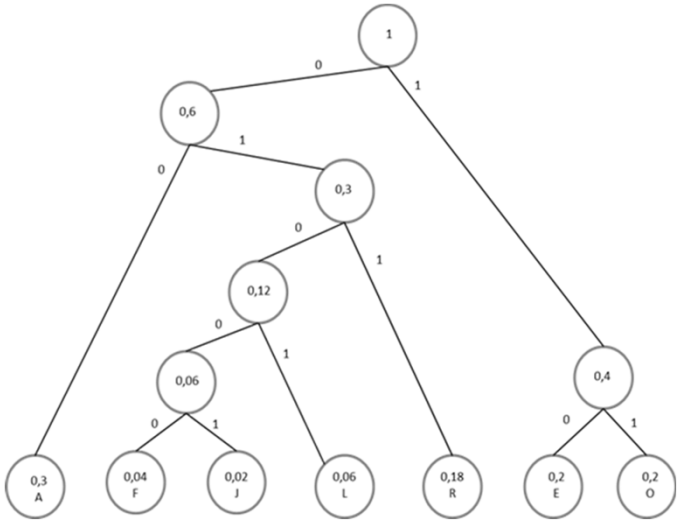
Código de Huffman: ejercicio

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	00000000000000000
101	17	00000000000000001
102	17	00000000000000010
103	17	00000000000000011
...	...	...
yopur	17	00001101010100100
youmg	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

Hay una representación diferente de un código Huffman que decodifica de manera muy eficiente modelos basados en palabras. Esta representación se denomina código canónico de Huffman.
Utiliza las mismas longitudes de codewords que un código Huffman, pero impone una elección particular en los bits de los codewords.
Los codewords se muestran en orden decreciente de longitud y, por lo tanto, en orden creciente de frecuencia de las palabras, excepto que dentro de cada bloque de códigos de la misma longitud, las palabras se ordenan alfabéticamente en lugar de por frecuencia.
La lista comienza con las miles de palabras (y números) que aparecen solo una vez. Muchas de las palabras, como ‘yopur’ y ‘youmg’, aparecen solo una vez porque son errores tipográficos. (Comienzan en 100 en lugar de un número menor porque las palabras de la misma longitud de codeword se ordenan en orden alfabético y no numérico, de modo que 90, 9 1,. . . aparecen más adelante). La tabla muestra los codewords ordenados de mayor a menor.

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.

Con los códigos canónicos no es necesario almacenar un árbol para la decodificación. Todo lo que se requiere es una lista de los símbolos (palabras) ordenados de acuerdo con el orden alfabético de los codewords más un vector que almacena el codeword del primer símbolo (palabra) de cada longitud distinta.
Por ejemplo, si con el ejemplo de la diapositiva anterior llega la secuencia 1100000101... , entonces el decodificador determinará que el siguiente codeword debe venir después de la secuencia de 7 bits (1010100), y antes de la primera palabra de seis bits, (110001).
Se leen los siguientes siete bits (1100000) y se calcula la diferencia de este valor binario al primer valor de siete bits del vector.
En este ejemplo, la diferencia es 12, lo que significa que la palabra tiene 12 posiciones después de la primera de la lista de símbolos (palabras) con codeword de longitud 7, por lo que debe ser ‘with’.

4. Compresión de texto: códigos LZ77 y LZ78

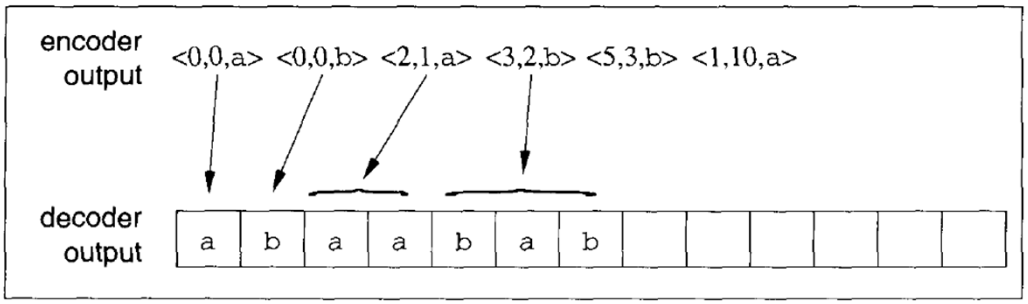
Una subcadena es remplazada por un puntero a una ocurrencia anterior.

- 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

Veamos ahora un método de codificación basada en diccionarios. Prácticamente todos los métodos de compresión basados en diccionarios se han desarrollado a partir de los propuestos por Jacob Ziv y Abraham Lempel en los años 70s.
Se les conoce como los métodos LZ77 y LZ78.

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.

La figura muestra algunos resultados de un codificador LZ77, suponiendo, que el alfabeto de entrada es {a,b}. La salida consta de una serie de tripletes. El primer componente de cada triplete indica qué tan atrás debe desplazarse en el texto anterior para encontrar el siguiente segmento.
El segundo componente indica la longitud del segmento, y el tercero da el siguiente carácter de la entrada. Los primeros dos elementos constituyen un puntero al texto. El último componente realmente es necesario si el carácter a codificar no se encuentra en el texto anterior, pero lo vamos a incluir en el ejemplo por simplicidad.
En el ejemplo de la figura la secuencia de caracteres '*abaabab'* ya ha sido decodificada, y es la hora de decodificar el triplete (5, 3, *b).* El decodificador va hacia atrás 5 caracteres y copia en la secuencia decodificada 3 caracteres a partir del puntero, es decir 'aab'. Como el triplete acaba en 'b' añade una 'b' adicional a la derecha. Tras ese paso el resultado de decodificación será la secuencia: 'abaababaabb'.

Algoritmos de codificación y decodificación LZ77

To encode the text $S[1 \ldots N]$ 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 $S[p \ldots]$ in $S[p - W \ldots p - 1]$. 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 $S[1 \ldots N]$ 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 $S[p \ldots p + l - 1] \leftarrow S[p - f \ldots p - f + l - 1]$.
   (b) Set $S[p + l] \leftarrow c$.
   (c) Set $p \leftarrow p + l + 1$.

En la diapositiva se muestran el algoritmo de codificación y de decodificación para LZ77.
En la codificación, la búsqueda de una coincidencia se restringe a una ventana de longitud W en el texto anterior.
Hay diferentes aproximaciones para llevar a cabo la búsqueda de la coincidencia más larga en esa ventana, que realmente es el proceso más complejo.

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.
- Un 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).

Ejercicio de decodificación con LZ77

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$hombre$aunanariz$pegado$éraseunanariz$superlativa$érase unanariz$sayón$y$escriba

Solución del ejercicio.

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.