UNIVERSIDAD COMPLUTENSE MADRID

Índices

Sistemas de Gestión de Datos y de la Información

Enrique Martín - emartinm@ucm.es

Máster en Ingeniería Informática

Fac. Informática

Índices

- Un **índice** es la **estructura de datos** principal que se utiliza para la recuperación de información.
- El objetivo de un índice es **almacenar** la **información importante** de los documentos para acceder de manera **rápida** a la hora de resolver consultas.

Índices

- ¿Cuál es la información importante de los documentos? Principalmente la aparición de **términos clave** en ellos. La información almacenada puede ser más o menos detallada:
  - Si el término clave aparece o no
  - El número de apariciones del término clave
  - Las posiciones en las que aparece

Índices

- El principal tipo de índices utilizado en recuperación de información se conoce como **índices invertidos**.
- La razón es que en la colección, la relación directa es que “los documentos contienen los términos clave”. En el índice invertido la relación es al revés, relacionamos “términos clave con los documentos que los contienen”

Índices: ejemplo

Vocabulario	#doc	d1	d2	d3
el	3	5	1	2
amigo	1	-	-	2
hola	3	4	3	1
verde	3	5	1	4
pez	1	-	2	-
coral	1	6	-	-
agua	3	3	3	2
mar	2	3	-	1
abuela	1	-	4	-

- Consideramos 3 documentos: d1, d2 y d3
- #doc almacena el número de documentos donde aparece
- Para cada documento, almacena el número de apariciones

Índices: ejemplo

Vocabulario			
el	(1,5)	(2,1)	(3,2)
amigo	(3,2)		
hola	(1,4)	(2,3)	(3,1)
verde	(1,5)	(2,1)	(3,4)
pez	(2,2)		
coral	(1,6)		
agua	(1,3)	(2,3)	(3,2)
mar	(1,3)	(3,1)	
abuela	(2,4)		

- Es un sistema real, hay miles o millones de documentos.
- Para la mayoría de los términos, la fila estará prácticamente vacía.
- Por ello lo más normal es almacenar las ocurrencias como **listas ordenadas**.

Índices invertidos completos

- Los índices invertidos vistos hasta ahora únicamente almacenan los **documentos** en los que aparece un término clave, y el **número de ocurrencias**.
- Si queremos poder responder a consultas sobre **frases** o que tengan en cuenta la **proximidad** de los términos clave, deberíamos extender la información que almacenan.

Índices invertidos completos

- Para cada término clave un índice invertido completo almacenará:
  - El número de documentos que lo contienen.
  - Una lista completa de apariciones, es decir, una lista de tuplas
    (#doc, n, [pos1, …, posn])
    donde:
    - *#doc* es un número de documento
    - *pos1,…,posn* son posiciones dentro del documento.

Índices invertidos completos

- La información almacenada como *posición* puede ser de más o menos detallada:
  - Carácter en el que empieza
    Obtención de extractos del documento, búsqueda de frase, búsqueda de cercanía
  - Número de palabra en el documento
    Búsqueda de frase, búsqueda de cercanía
  - Número de bloque/párrafo en el que aparece
    Búsqueda de cercanía

Índices invertidos completos: ejemplo

Vocabulario	***ni***	**Lista de apariciones**
el	3	[(1,5,[1,3,5,7,9]), (2,1,[4]), (3,2,[17,9])]
amigo	1	[(3,2,[4,86])]
hola	3	[(1,4,[22,24,56,71]), (2,3,[1,10,20]),(3,1,[1])
verde	3	[(1,5,[19,25,32,40,59]),(2,1,[103]),(3,4,[76,80,91,111])]
pez	1	[(2,2,[45,123])]
coral	1	[(1,6,[101,134,145,204,250,278])]
agua	3	[(1,3,[67,78,106]), (2,3,[56,78,99,301]), (3,2,[1,34])]
mar	2	[(1,3,[500,509,700]), (3,1,[72])]
abuela	1	[(2,4,[303,581,600,708])]

Índices invertidos vectoriales

- Un índice invertido completo contiene toda la información posible.
- A partir de esa información se pueden calcular los **pesos** (w_{ij}) necesarios para calcular la similitud.
- Sin embargo, para consultas vectoriales (o booleanas) es más eficiente calcular todos los pesos y almacenar esa información:
  - En lugar de listas de posiciones → un número decimal representando el peso.

Índices invertidos vectoriales

Vocabulario	***ni***	**Pesos**
el	3	[(1, 2.345), (2, 1.987), (3, 2.3452)]
amigo	1	[(3, 2.990475)]
hola	3	[(1, 1.34234), (2, 2.324),(3,1.000234)
...	...	...
abuela	1	[(2,4.390248729384)]

Búsquedas con índices

Búsqueda con índices

- Hasta ahora hemos visto dos modelos de recuperación de información, que nos definen de manera **formal** qué significa que un documento encaje con una consulta.
- Sin embargo, no hemos abordado cómo calcular este encaje de manera **eficiente** en un sistema real.

Búsqueda con índices

- Una aproximación muy simple sería recorrer todos los documentos *di*, y para cada uno de ellos calcular su valor de encaje con la consulta *q*:

```
Buscar( c:Colección, q:Consulta):Colección
 res := Ø;
 para_cada d en c:
     si R(d,q) >= UMBRAL entonces
      res := res U {d};
 devolver res;
```

Búsqueda con índices

- Esta solución es **sencilla**, pero **muy poco eficiente** porque tiene que recorrer todos los documentos en cada consulta.
- En general, tenemos un número muy alto de documentos pero solo queremos recuperar un número bajo de resultados relevantes.
- Para conseguir eficiencia necesitaremos sacar provecho de los **índices**.

Modelo booleano

- Para realizar una consulta t1 AND t2 tendremos que:
  - Recuperar la lista de apariciones de t1 y t2 del índice.
  - Cruzar las listas y quedarse con los documentos comunes → INTERSECT(p1,p2).
- Todo esto hay que realizarlo de manera **eficiente**.

Modelo booleano

- Obtener la lista de apariciones de un término es rápido: solo debe consultarse el índice.
- Esta lista estará ordenada de manera **ascendente** por identificador de documento.
- Si **p** es una lista de apariciones, **docID(p)** nos devuelve el identificador de documento de su primera entrada.

Modelo booleano

```
INTERSECT(p1,p2):
  answer = {}
  while p1 != NIL and p2 != NIL:
    if docID(p1) == docID(p2):
      add(answer, docID(p1))
      p1 = next(p1)
      p2 = next(p2)
    elif docID(p1) < docID(p2):
      p1 = next(p1)
    else:
      p2 = next(p2)
  return answer
```

Modelo booleano

- La función INTERSECT(p1,p2) saca provecho de que las listas están **ordenadas**: cada lista se recorre como mucho una vez.
- Si p1 tiene longitud **x** y p2 longitud **y**, INTERSECT tiene un coste en **O(x+y)**.
- Para consultas t1 AND t2 AND ... AND tn se puede extender INTERSECT usando la heurística de *“intersecar primero las listas más cortas”*.

Modelo booleano

```
INTERSECT(<p1,...,pn>):
  terms = sortByLength(<p1,...,pn>)
  answer = head(terms)
  terms = tail(terms)
  while terms != NIL and answer != NIL:
    e = head(terms)
    answer = INTERSECT(answer,e)
    terms = tail(terms)
  return answer
```

Modelo booleano

- Para consultas OR se necesitaría una función **UNION(p1,p2)** para unir listas de apariciones de manera eficiente. También se hace uso de que están **ordenadas**.
- El operador NOT puede ser poco eficiente si se considera aislado, pero se pueden crear funciones específicas para su uso más común:
  - x AND (NOT y) → **INTERSECT_NOT(p1,p2)**

Consultas de frase

- Las **consultas de frase** requieren encontrar términos contiguos en documentos, p. ej.
  - *“Stanford University”*
  - *“European Union”,*
  - *“President of the United States of America”*
  - *“Universidad Complutense de Madrid”*
- Para este tipo de consultas se pueden aplicar:
  - Índices de **frase**
  - Índices completos **posicionales**

Índices de frase

- En lugar de términos clave aislados, almacenan grupos de **términos consecutivos**.
- Para simplificar se puede usar **tamaño 2**:
  - Consulta *“Hospital Infanta Leonor”* → *“Hospital Infanta”* **AND** *“Infanta Leonor”*
- Esta traducción puede dar **falsos positivos**.
- Considerar distintos tamaños genera índices extremadamente grandes.

Índices completos

- Almacenan las **posiciones** (*offset* o número de palabra) en las que aparece cada término en cada documento.
- Para resolver una consulta de frase, utilizan las listas de apariciones de todas las palabras involucradas.
- Se usa una variante de INTERSECT que en caso de coincidencia revisa si las posiciones relativas son las esperadas.

Índices completos

```
INTERSECT_PHRASE(<p1,...,pn>):
 1 cont = True
 2 answer = {}
 3 while cont:
 4   if sameDocID(<p1,...,pn>):
 5     if consecutive(<p1,...,pn>):
 6       add(answer, docID(p1))
 7     <p1,...,pn> = advanceMin(<p1,...,pn>)
 8   else:
 9     <p1,...,pn> = advanceMin(<p1,...,pn>)
10   cont = allNoNIL(<p1,...,pn>)
11 return answer
```

Índices completos

- `sameDocID` → bool: las cabezas de p1,...,pn almacenan el mismo docID.
- `consecutive` → bool: las cabezas de p1,...,pn son consecutivas.
- `advanceMin` : avanza las listas con menor docID en su cabeza (cada lista está ordenadas por docID). Puede avanzar todas las listas
- `allNoNIL` → bool: ninguna lista p1,...,pn se ha terminado

Modelo vectorial

- Buscar los documentos que encajan con una consulta utilizando el **modelo vectorial** y el índice invertido es un poco más elaborado.
- Recordemos la función de **relevancia**:

$$R(dj,q) = \frac{dj \cdot q}{|dj||q|} = \frac{\sum_{i=1}^{t} w_{ij} w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2} \sqrt{\sum_{i=1}^{t} w_{iq}^2}}$$

Modelo vectorial

- Según parece, vamos a necesitar todos los pesos w_{ij} del documento y w_{iq} de la consulta para poder calcular la similitud.
- Si tengo que acceder a todos los datos de cada documento, ¿para qué me sirve el índice invertido?

$$R(dj,q) = \frac{dj \cdot q}{|dj||q|} = \frac{\sum_{i=1}^{t} w_{ij} w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2} \sqrt{\sum_{i=1}^{t} w_{iq}^2}}$$

Modelo vectorial

- Miremos primero el producto escalar:

$$dj \cdot q = \sum_{i=1}^{t} w_{ij} w_{iq}$$

- **PREGUNTA:** ¿Para una consulta dada, de verdad necesitamos conocer todos los w_{ij} ?

Modelo vectorial

- Miremos primero el producto escalar:

$$dj \cdot q = \sum_{i=1}^{t} w_{ij} w_{iq}$$

- **PREGUNTA:** ¿Para una consulta dada, de verdad necesitamos conocer todos los w_{ij} ?
- **RESPUESTA:** NO, únicamente para aquellos términos clave *ki* en los que w_{iq} es mayor que 0. Para los términos clave *kl* tales $w_{lq} = 0$ entonces $\boldsymbol{w_{lj} \cdot w_{lq} = w_{lj} \cdot 0 = 0}$.

Modelo vectorial

- Como *q* es un vector con 1 en los términos buscados y 0 en los no buscados, el producto escalar *dj* • *q* se simplifica:

$$d_j \cdot q = \sum_{i=1}^{t} w_{ij} w_{iq} \rightarrow \sum_{i \in \text{terms}(q)} w_{ij}$$

para los términos *i* que aparecen en *q*.

- *(Los términos i que no aparecen en q no deben considerarse, pues no contribuyen al sumatorio debido a que* $w_{iq} = 0$*)*

Modelo vectorial

- Por último, la norma |q| es un factor positivo constante en todas las similitudes, por lo que podemos omitirlo y el **orden relativo** de los documentos no cambia:

$$R(dj,q) > R(dk,q) \iff$$

$$\frac{dj \cdot q}{|dj||q|} > \frac{dk \cdot q}{|dj||q|} \iff$$

$$\frac{1}{|q|}\frac{dj \cdot q}{|dj|} > \frac{1}{|q|}\frac{dk \cdot q}{|dj|} \iff$$

$$\frac{dj \cdot q}{|dj|} > \frac{dk \cdot q}{|dj|}$$

Modelo vectorial

- Para el cálculo de la norma de |*dj*| necesitamos **todos sus pesos**, incluso para los términos clave que no aparecen en la consulta.

$$|dj| = \sqrt{\sum_{i=1}^{t} w_{ij}^2}$$

- **Pero no depende de la consulta → precalcular y almacenar**
- Supondremos que esta información está almacenada en una estructura `Length[N]` donde `Length[j]` = |*dj*|

Modelo vectorial: pseudocódigo

```
FASTCOSINESCORE(q, index, k):
1 Scores[N] = 0 # Diccionario de resultados
2 for t in q:
3   p = index(t) # Lista de pesos 'w' del término t en cada
                 # documento 'd'
4   for (d,w) in p:
5     Scores[d] += w # Evitamos el  producto
6 for d in Scores:     # Dividimos por |d|
7   Scores[d] = Scores[d]/Length[d]
8 return best(k,Scores)
```

Modelo vectorial: pseudocódigo

- 1) Inicializamos el producto escalar parcial asociado a cada documento d (como máximo habrá N)
- 3) Obtenemos la lista [(doc,peso),...(doc,peso)] asociada al término t
- 4-5) Sumamos el peso en cada producto escalar parcial
- 6-7) Dividimos entre la norma |d|
- 8) Obtenemos los k documentos con mayor relevancia

Tamaño de los índices

Índices

- Los índices almacenan una lista de entradas por cada palabra del vocabulario.
- Cada entrada puede ser simple (docID) o muy detallada (docID y lista de apariciones).
- Por tanto, los **índices** ocupan una **cantidad elevada de memoria.** Pueden incluso no caber en memoria principal.

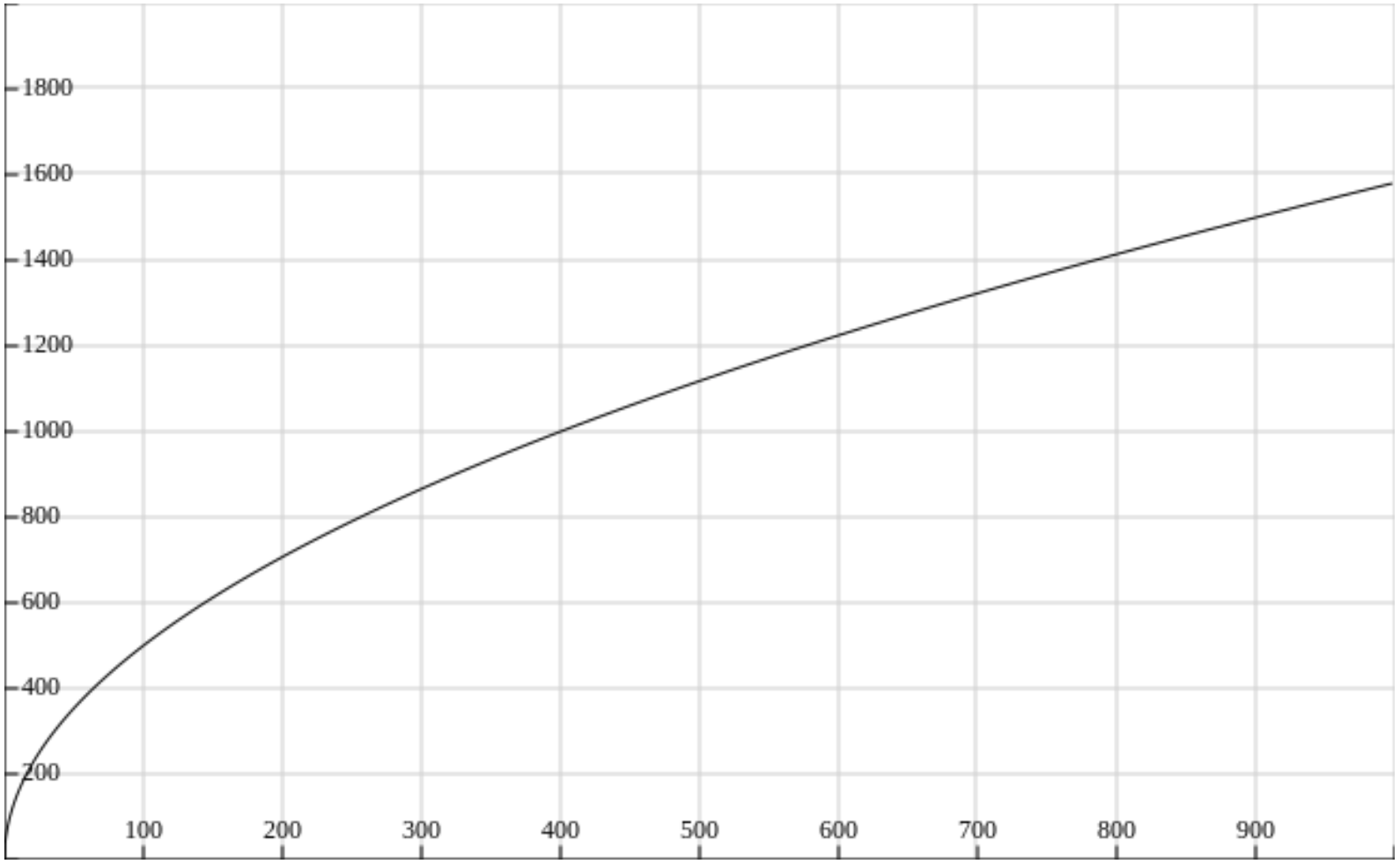
Ley de Heaps

- La **Ley de Heaps** estima el tamaño del vocabulario V en base al número de palabras T en una colección:

$$V = kT^b, \ con \ 30 \leq k \leq 100 \ y \ b \approx .5$$

- Es una ley empírica, con valores concretos para cada colección.
- Muestra que el número de términos crece **proporcional a la raíz del número de palabras**.
- Por tanto, el vocabulario crece de manera controlada según aumenta la colección.

Ley de Heaps



Para k=50 y b=0,5

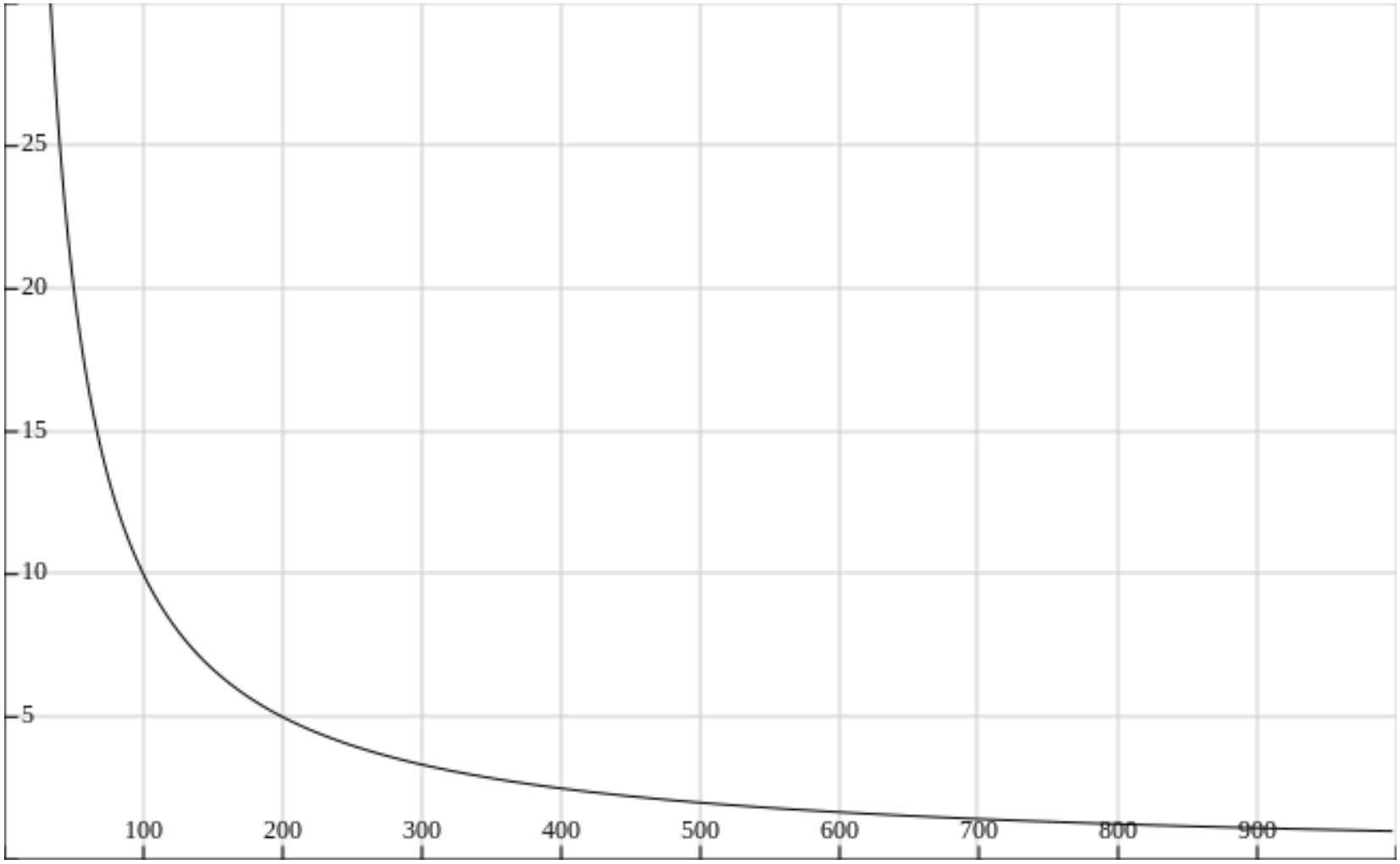
Ley de Zipf

- Por otro lado la **Ley de Zipf** relaciona la repeticiones de cada término.
- Sea cf_i las repeticiones del i-ésimo término más frecuente. Estas repeticiones cf_i son proporcionales a 1/i:

$$cf_i \approx \frac{1}{i}$$

  - El segundo elemento más frecuente aparece 1/2 de veces que el más frecuente.
  - El tercer elemento más frecuente aparece 1/3 de veces que el más frecuente.

Ley de Zipf



Ley de Zipf

- Esta ley empírica nos indica que unos pocos términos se repetirán mucho, por lo que contribuirán mucho al tamaño del índice.
- Estos términos suelen ser **separadores/palabras vacías** *(stopping words)* como artículos, pronombres, conjunciones, etc.
- Descartar separadores reducirá el tamaño del índice. Sin embargo no siempre es posible → búsquedas de frases.

Compresión

- Para reducir el tamaño de un índice se utilizan distintas técnicas de **compresión**.
- Las técnicas que más reducen el tamaño son las que comprimen **listas de apariciones**.
- Representan las apariciones a bajo nivel usando **bits**, por lo que **reducen** el tamaño pero **incrementan** el coste al necesitar procesarlas
  → compensa si evitas accesos a disco.

Compresión

- Las listas de repeticiones son listas de posiciones donde aparece un término en un documento: [4,16,65,89,134].
- Un primer paso para comprimir es no almacenar posiciones sino sus **diferencias**:
  [4,16-4,65-16,89-65,134-89]
  = [4,12 ,49 ,24 ,45]
- Las diferencias son números más pequeños y por tanto necesitarán **menos bits** para representarse.

Compresión

- Si todos los números necesitan el mismo número de bits no se gana nada al usar diferencias, pues todos ocuparán 32/64 bits.
- Debemos aplicar codificación de **longitud variable**.
- Veremos 4 técnicas:
  - Código unario
  - Elias-gamma
  - Elias-delta
  - Bytes variables

Compresión

- Sin embargo existen más técnicas:
  - Elias-omega
  - Golomb/Rice
  - LLRUN
  - Combinaciones de las anteriores

Código unario

- Muy sencilla: un número **n** se representa como (n-1) **1's** seguido de un **0**.
  - 1: **0**
  - 3: **110**
  - 7: **1111110**
- Una lista de diferencias se puede representar fácilmente como una lista de bits:
  [1,3,7] → **01101111110** (11 bits)
- Es óptimo desde el punto de vista de espacio para distribuciones de probabilidad:

$$P(n) = 2^{-n}$$

Elias-gamma

- La representación Elias-gamma $(\text{Elias-}\gamma)$ de un número **n** se forma concatenando 2 partes:
  - *offset:* **n** en binario, quitando el primer 1.
  - longitud: representación en **unario** de la longitud de **n** en **binario** $(\lfloor log_2\ n \rfloor + 1)$
- La representación es ***longitud + offset***

Número	Binario	Offset	Longitud (unario)	Representación
1	1		0 (1)	**0**
2	10	0	10 (2)	**100**
3	11	1	10 (2)	**101**
4	100	00	110 (3)	**11000**
5	101	01	110 (3)	**11001**

Elias-delta

- La representación Elias-delta $(\text{Elias-}\delta)$ es similar a Elias-gamma, se concatenan 2 partes.
  - *offset:* **n** en binario, quitando el primer 1.
  - longitud: **representación en** $\text{Elias-}\gamma$ de la longitud de **n** en **binario** $(\lfloor log_2\ n \rfloor + 1)$
- La representación es ***longitud + offset***

Número	Binario	Offset	Longitud Elias-gamma		Representación
1	1		0	(1)	**0**
2	10	0	100	(2)	**1000**
3	11	1	100	(2)	**1001**
4	100	00	101	(3)	**10100**
5	101	01	101	(3)	**10101**

Elias-gamma y Elias-delta

- En general la codificación Elias-gamma es más corta que Elias-delta para número pequeños (hasta 32).
- Según los números van creciendo, la codificación Elias-delta genera codificaciones más cortas.
- En ambos códigos, una lista de repeticiones se codifica concatenando la representación de sus elementos. P. ej. la lista [3,4,5]:
  - Elías-gamma → 101110001100 1 (13 bits)
  - Elías-delta → 100110100101 01 (14 bits)

Bytes variables

- Cada números se representan como secuencias de 1 o varios bytes. Este número de bytes es variable.
- El primer bit de cada byte es el llamado *flag* de continuación:
  - **1**: el byte actual es el último de la secuencia.
  - **0**: el byte actual será continuado con otro byte.
- Los restantes 7 bits almacenan (una parte de) la representación en binario del número.

Bytes variables

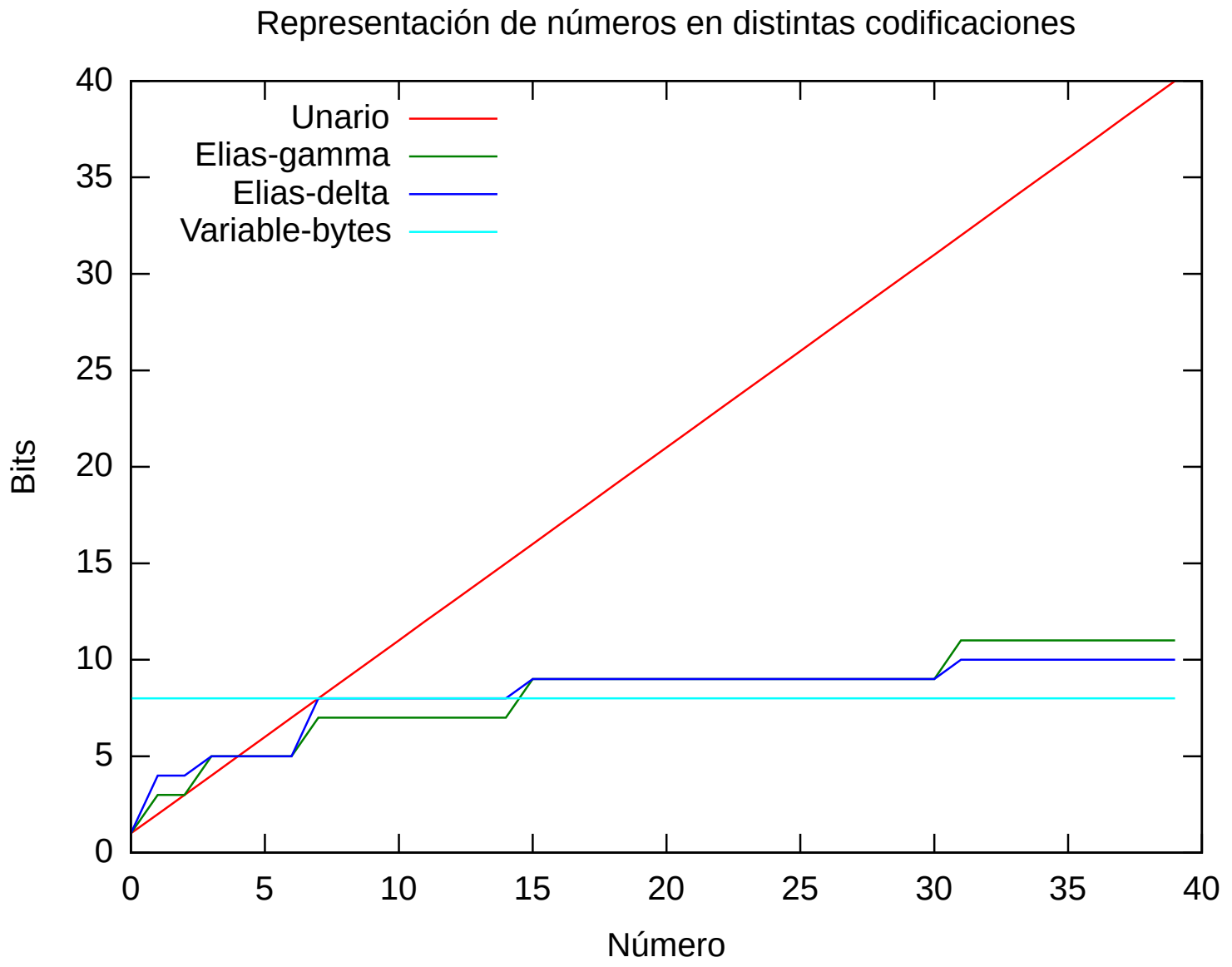
Número	Binario	Variable bytes
1	1	**1**0000001
2	10	**1**0000010
4	100	**1**0000100
5	101	**1**0000101
127	1111111	**1**1111111
315	100111011	**0**0000010 **1**0111011
824	1100111000	**0**0000110 **1**0111000

- Una lista de número se representa como una cadena de bytes:

10000100 **1**0000101 **1**1111111 **0**0000010 **1**0111011

4 5 127 315

Comparación codificaciones



Comparación codificaciones

- Tamaño de la representación de una lista aleatoria de 50 números entre el 1 y el 1000:
  - Enteros en binario: 1600 bits (32.0 bits/número)
  - Unario: 23107 bits (462.14 bits/número)
  - Elias-gamma: 838 bits (16.76 bits/número)
  - Elias-delta: 730 bits (14.6 bits/número)
  - Bytes variables: 744 bits (14.88 bits/número)

Bibliografía

Bibliografía

- **Information Retrieval: Implementing and Evaluating Search Engines.** Stefan Büttcher, Charles L. A. Clarke, Gordon V. Cormack. MIT Press, 2010.
- **Introduction to Information Retrieval**. *Christopher D. Manning, Prabhakar Raghavan y Hinrich Schütze*. Cambridge University Press. 2008. http://www-nlp.stanford.edu/IR-book/

Bibliografía

- **Modern Information Retrieval, the concepts and technology behind search**, second edition. *Ricardo Baeza-Yates y Berthier Ribeiro-Neto*. Pearson Education Limited (2011).
- **Search Engines: Information Retrieval in Practice** (International Edition). *W. Bruce Croft, Donald Metzler y Trevor Strohman*. Person Education. 2010.