Implementación de Máquinas de Búsqueda I:

Indices y Compresión

Gonzalo Navarro

Centro de Investigación de la Web

Universidad de Chile

Mapa de la Charla

- Modelo booleano de Recuperación de Información (RI)
- Modelo vectorial de RI
- Precisión y recuperación
- Consultas más complejas
- Indices invertidos
- Espacio extra y compresión de índices
- Consulta en el modelo booleano
- Consulta en el modelo vectorial
- Consulta de patrones complejos
- Construcción de índices invertidos
- Compresión de textos en sistemas de RI
- La Web como repositorio de información
- Arquitectura de las máquinas de búsqueda Web
- Ranking en la Web
- Indices para la Web

Modelo Booleano de RI

- Un documento será relevante o no para una consulta, sin matices.
- Consultas de una palabra: un documento es relevante sii contiene la palabra.
- Consultas `AND`: los documentos deben contener todas las palabras.
- Consultas `OR`: los documentos deben contener alguna palabra.
- Consultas A `BUTNOT` B: los documentos deben ser relevantes para A pero no para B.
- Ejemplo: los últimos ataques terroristas

```
(11 AND (Septiembre OR Marzo)) BUTNOT 1973
```

- Es el modelo más primitivo, y bastante malo para RI.
- Sin embargo, es bastante popular.

- ¿Por qué es malo?
  - No discrimina entre documentos más y menos relevantes.
  - Da lo mismo que un documento contenga una o cien veces las palabras de la consulta.
  - Da lo mismo que cumpla una o todas las cláusulas de un `OR`.
  - No considera un calce parcial de un documento (ej. que cumpla con *casi* todas las cláusulas de un `AND`).
  - No permite siquiera ordenar los resultados.
  - El usuario promedio no lo entiende:

    `"Investigar los atentados del 11/S y del 11/M"`
    $\longrightarrow$ `11/S AND 11/M`

    (grave error, se perderán excelentes documentos que traten de uno solo de los dos eventos en profundidad, debió ser `11/S OR 11/M`)
- ¿Por qué es popular?
  - Es de las primeras ideas que a uno se le ocurren.
  - Muchos de los primeros sistemas de RI se basaron en él.
  - Es la opción favorita para insertar texto en un RDBMS.
  - Es simple de formalizar y eficiente de implementar.
  - En algunos casos (usuarios expertos) puede ser adecuado.
  - Puede ser útil en combinación con otro modelo, ej. para excluír documentos.
  - Puede ser útil con mejores interfaces.

Modelo Vectorial de RI

- Se selecciona un conjunto de palabras útiles para discriminar (*términos* o *keywords*).
- En los sistemas modernos, toda palabra del texto es un término, excepto posiblemente las *stopwords* o *palabras vacías*.
- Sea $\{t_1, \ldots, t_k\}$ el conjunto de términos y $\{d_1, \ldots, d_N\}$ el de documentos.
- Un documento d_i se modeliza como un vector

$$d_i \quad \longrightarrow \quad \vec{d_i} \;=\; (w(t_1, d_i), \ldots, w(t_k, d_i))$$

  donde $w(t_r, d_i)$ es el *peso* del término t_r en el documento d_i.
- Hay varias fórmulas para los pesos, una de las más populares es

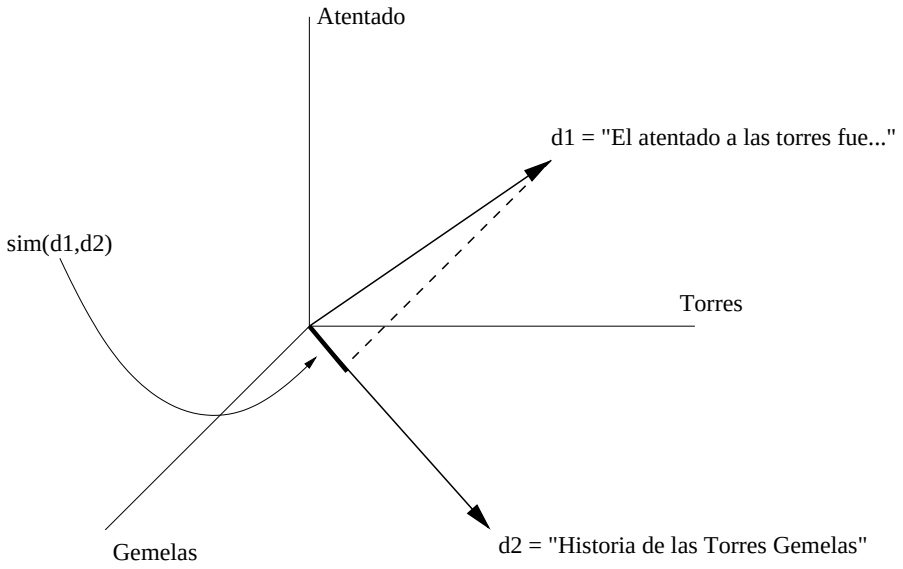
$$w(t_r, d_i) \;=\; w_{r,i} \;=\; \frac{tf_{r,i} \;\times\; idf_r}{|\vec{d_i}|} \;=\; \frac{tf_{r,i} \;\times \log \frac{N}{n_r}}{\sqrt{\Sigma_{s=1}^{k} \left(tf_{s,i} \;\times \log \frac{N}{n_s}\right)^2}}$$

  donde $tf_{r,i}$ (term frequency) es la cantidad de veces que t_r aparece en d_i, y n_r es la cantidad de documentos donde aparece t_r.

- Si un término aparece mucho en un documento, se supone que es importante en ese documento (tf crece).
- Pero si aparece en muchos documentos, entonces no es útil para distinguir ningún documento de los otros (idf decrece).
- Además normalizamos los módulos de los vectores para no favorecer documentos más largos.
- Lo que se intenta medir es cuánto ayuda ese término a distinguir ese documento de los demás.
- Se calcula la *similaridad* entre dos documentos mediante la *distancia coseno*:

$$sim(d_i, d_j) \quad = \quad \vec{d_i} \cdot \vec{d_j} \quad = \quad \sum_{r=1}^{k} w_{r,i} \times w_{r,j}$$

(que geométricamente corresponde al coseno del ángulo entre los dos vectores).



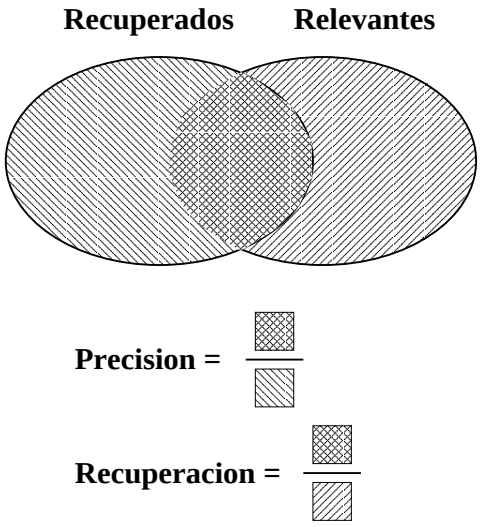
- La similaridad es un valor entre cero y uno.
- Notar que dos documentos iguales tienen similaridad 1, y ortogonales (si no comparten términos) tienen similaridad cero.
- En particular, una consulta se puede ver como un documento (formado por esas palabras) y por lo tanto como un vector.
- Dado que en general cada palabra aparece una sola vez en la consulta y esta es muy corta, se puede en general calcular la *relevancia* de d_i para q con la fórmula

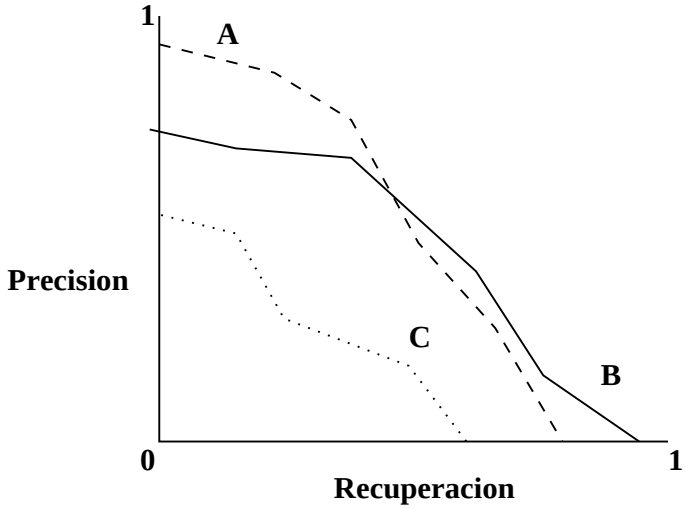
$$sim(d_i, q) \quad = \quad \sum_{r=1}^{k} w_{r,i} \ \times \ w_{r,q} \quad \sim \quad \sum_{t_r \in q} w_{r,i} \ \times \ idf_r$$

- La última fórmula no es idéntica a sim pero ordena los documentos de la misma forma (falta dividir por $|\vec{q}|$).
- Pero el modelo es más general, y permite cosas como:
  - Que la consulta sea un documento.
  - Hacer *clustering* de documentos similares.
  - Relevance feedback ("more like this").
- Este modelo es, de lejos, el más popular en RI hoy en día.

Precisión y Recuperación

- Finalmente, ¿cómo mediremos la bondad de un modelo para RI?
- Hay muchas medidas, pero la más popular es un diagrama de *precisión – recuperación* (precision – recall).
- *Precisión:* cuántos documentos recuperados son relevantes.
- *Recuperación:* cuántos documentos relevantes se recuperaron.





- Según la cantidad que se elige como relevantes, se puede hacer aumentar una a costa de reducir la otra.
- Para evaluar un modelo de RI se debe tener en cuenta todo el gráfico. En la figura, A o B pueden ser preferibles según la aplicación.
- En ciertos casos, como la Web, puede ser imposible calcular la recuperación. Se puede hacer mediante muestreo.

Consultas más Complejas

- Usualmente no es suficiente poder buscar por palabras aisladas.
- Distintos RIs ofrecen distintas posibilidades de búsquedas más complejas.
- Frases: secuencias de palabras que aparecen seguidas

  `"Torres Gemelas"`
- Para hallarlas no basta saber que las palabras están en el documento, sino que se necesita conocer su posición.
- Lo mismo pasa con cercanías de palabras

  `auto* CERCA DE usad*`

  para calzar

  `... vendo auto usado ...`

  `... gran venta de autos de todos los modelos, nuevos y usados ...`

  `... permuto por auto o camioneta usados ...`

  Observar que si no se exige que las palabras estén cerca la consulta puede retornar muchos documentos irrelevantes.
- A veces el poder buscar por prefijos, sufijos, substrings, etc. es útil.

- En determinados casos, el usuario puede necesitar encontrar un conjunto de strings y precisa herramientas más potentes para especificarlo. Por ejemplo patrones extendidos (clases de caracteres, wild cards, caracteres repetibles u opcionales), ej.

  `doc[0-9]+.html?`

  o expresiones regulares como

  `(Dr.|Mr.|PhD.|Prof.)* Knuth`

- Otro caso muy importante es la búsqueda aproximada: permitir un número limitado de diferencias (inserciones, borrados, reemplazos de caracteres) entre el patrón de búsqueda y sus ocurrencias en el texto.

- Esto será necesario cuando el usuario no esté seguro de cómo se escribe una palabra o nombre extranjero, o cuando el texto tenga errores de ortografía, tipeo, OCR, etc.

- Esto último es especialmente relevante en bases de datos de mala calidad como la Web. Ejemplo: un 30 % de las páginas Web acerca de `"Levenshtein"` están mal escritas y se perderían en una consulta normal.

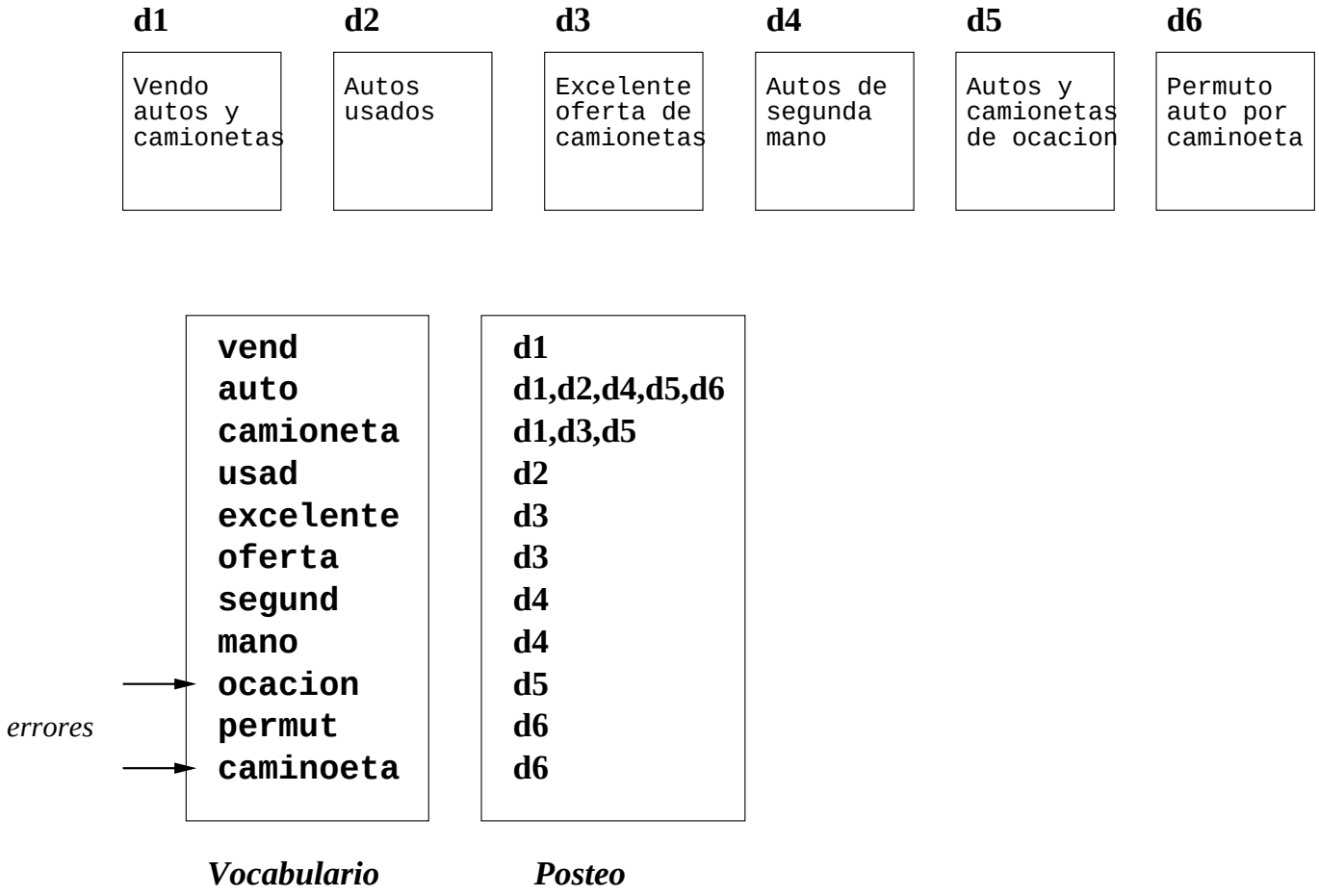
- No está claro cómo combinar adecuadamente una consulta de búsqueda de patrones con RI. Por ejemplo, ¿cómo calcular la relevancia de una búsqueda por proximidad o aproximada?

Indices Invertidos

- Es la estructura más elemental para recuperación de palabras.
- En su versión más básica, consta de dos partes:

  Vocabulario: conjunto de términos distintos del texto.

  Posteo: para cada término, la lista de documentos donde aparece.



- Variante para el modelo booleano:
  - Es conveniente almacenar la lista de posteo de cada término en orden creciente de documento.
- Variante para el modelo vectorial:
  - Debe almacenar el tf correspondiente en cada entrada de posteo (observar que realmente debería ser $tf/|d_i|$).
  - Debe almacenar el idf correspondiente en cada entrada del vocabulario.
  - Es conveniente almacenar el máximo tf de cada término en el vocabulario.
  - Es conveniente almacenar la lista de posteo de cada término en orden decreciente de tf.
- Variante para frases y cercanías:
  - Se debe almacenar además la **inversión**: para cada término y documento, la lista de sus posiciones exactas en el documento.
  - Para reducir espacio, se puede tener un *índice de bloques*: se corta el texto en grandes bloques y se almacena sólo los bloques donde ocurre cada término.

Espacio Extra y Compresión de Indices

- Dos leyes empíricas importantes, ampliamente aceptadas en RI.
- Ley de Heaps: el vocabulario de un texto de largo n crece como

$$k \quad = \quad C \times n^{\beta}$$

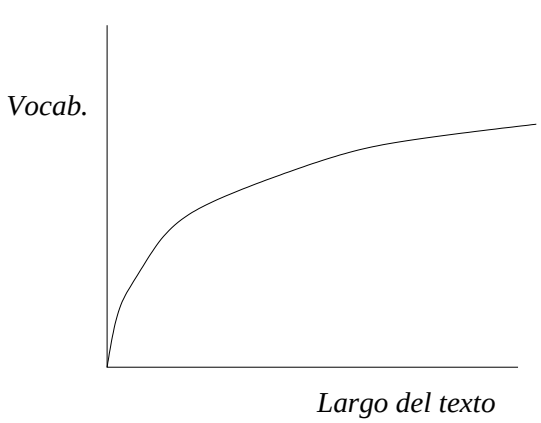
  donde C y $0 < \beta < 1$ dependen del tipo de texto. En la práctica, $5 < C < 50$ y $0{,}4 \leq \beta \leq 0{,}6$, y k es menos del 1 % de n.
- En la práctica el vocabulario entra en memoria RAM (ej. 5 Mb para 1 Gb de texto).
- Ley de Zipf: si se ordenan los términos de un texto de más a menos frecuente, entonces

$$n_r \quad = \quad \frac{N}{r^{\theta} H_N(\theta)}$$

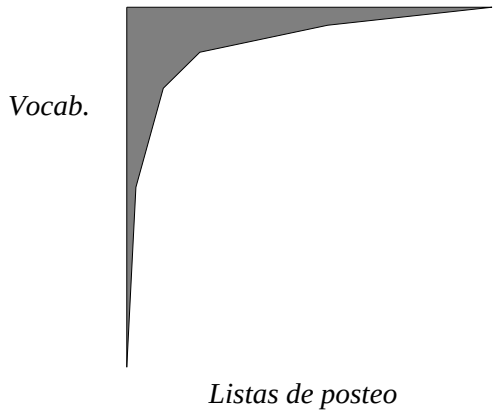
  donde

$$H_N(\theta) \quad = \quad \sum_{r=1}^{k} \frac{1}{r^{\theta}}$$

  lo que significa que la distribución es muy sesgada: unas pocas palabras (las stopwords) aparecen muchas veces y muchas aparecen pocas veces.
- En particular, las stopwords se llevan el 40 %-50 % de las ocurrencias y aproximadamente la mitad de las palabras aparecen sólo una vez.



Ley de Heaps

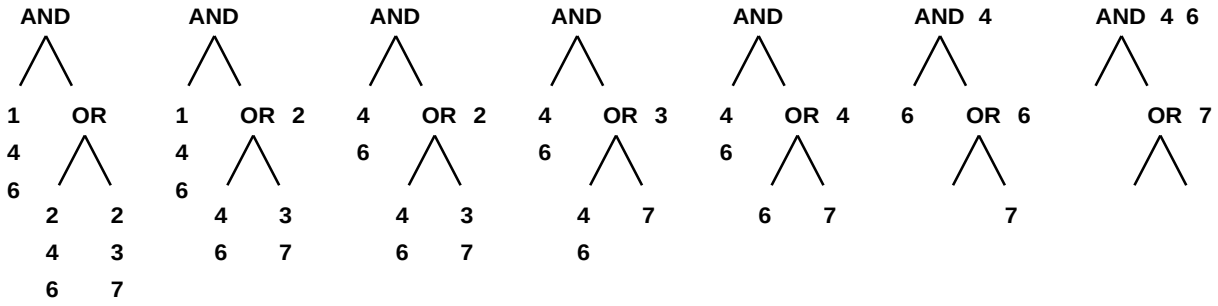


Ley de Zipf

- Posteo/Inversión:
  - Caso booleano: los números de documentos son crecientes, se pueden almacenar las diferencias (Elías). Así comprimido, requiere un 10 %-25 % extra sobre el texto.
  - Caso vectorial: eso ya no ocurre, pero todos los documentos con el mismo tf (muchos, por Zipf) pueden aún ordenarse. El índice requiere 15 %-30 % extra.
  - Frases y cercanías: la inversión lleva el espacio total a 25 %-45 % extra (posiciones crecientes).
  - Con direccionamiento a bloques esto puede bajar hasta a 4 % para colecciones no muy grandes, al costo de mayor búsqueda secuencial.

Consulta en el Modelo Booleano

- Las palabras de la consulta se buscan en el vocabulario (que está en memoria), por ejemplo usando hashing.
- Se recuperan de disco las listas de posteo de cada palabra involucrada.
- Se realizan las operaciones de conjuntos correspondientes sobre ellas (unión, intersección, diferencia).
- Las listas están ordenadas crecientemente, de modo que se puede operar recorriéndolas secuencialmente. Los documentos se recuperan ordenados por número de documento.
- Si una lista es muy corta y la otra muy larga, es más rápido buscar binariamente la corta en la larga ($O(1)$ por Zipf).
- Esto se complica si las listas están representadas por diferencias, pero se puede almacenar cada tanto un valor absoluto.
- Se puede usar evaluación full o lazy

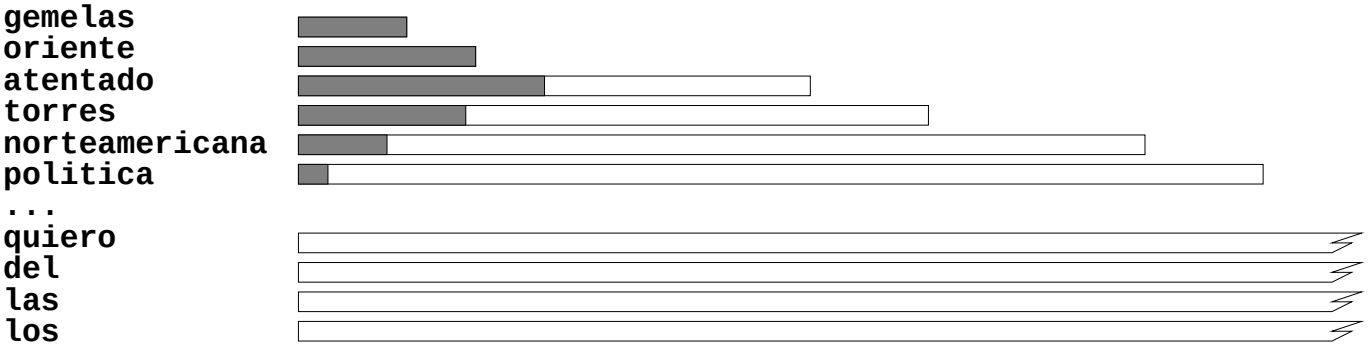


Consulta en el Modelo Vectorial

- La consulta puede tener muchas palabras y sólo nos interesa recuperar los R documentos más relevantes.
- Por otro lado, los documentos de cada término están almacenados en orden decreciente de tf.
- La idea es mantener un *ranking* de los R documentos d_i con mayor $sim(d_i, q)$.
- Partimos con el término de la consulta de mayor idf (lista de posteo más corta), y traemos los R primeros documentos de su lista (almacenados en ese orden).
- Si no llegamos a juntar R, seguimos con el segundo término de mayor idf y así.
- Una vez que tenemos ya R candidatos, seguimos recorriendo las listas de los términos, de mayor a menor idf.
- Sin embargo, como el tf en cada lista decrece, podemos en cierto momento determinar que no es necesario seguir recorriendo la lista pues los candidatos no pueden entrar al ranking de los R mejores.
- Si se almacena el máximo tf en el vocabulario, es posible eliminar términos completos sin siquiera ir al disco una vez.

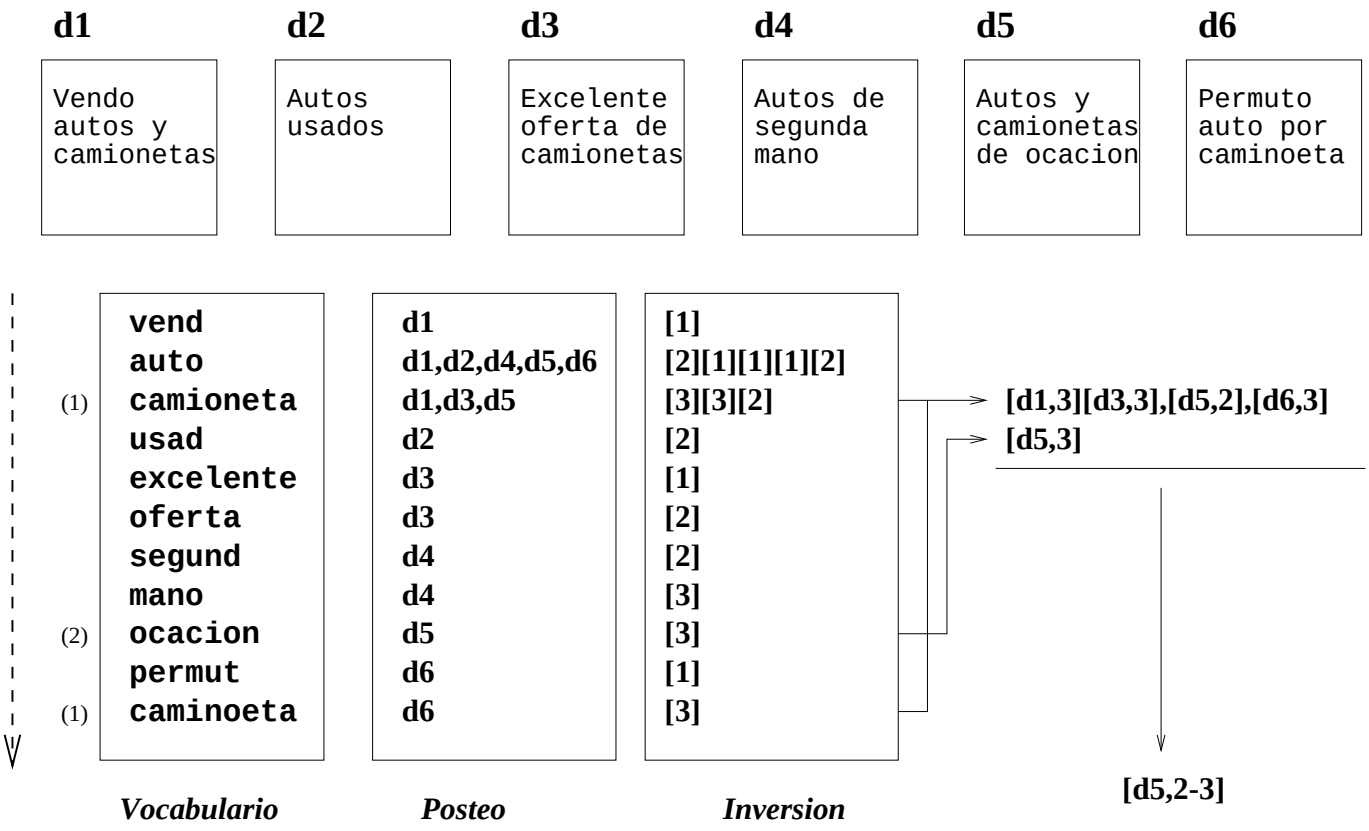
- Puede hacerse incluso más eficiente cortando las listas donde se considere *improbable* que modifiquen el ranking.
- Este tipo de "relajamiento" está permitido en RI y es muy utilizado en las máquinas de búsqueda para la Web.
- En particular, en la Web la *recuperación* no sólo es difícil de medir sino que ya viene condicionada por el crawling, que raramente alcanza a cubrir el 30 % del total. De modo que las máquinas se concentran más en la precisión. Es decir, que lo que se recupera sea bueno más que recuperar todo lo bueno.

quiero informacion sobre las consecuencias del atentado a las torres gemelas en la politica norteamericana en el medio oriente



Búsquedas Complejas

- Se comienza buscando la consulta en el vocabulario. Esto no es tan trivial como antes porque la consulta puede ser una expresión regular, búsqueda aproximada, etc.
- En estos casos se realiza una búsqueda *secuencial* en el vocabulario. Esto es tolerable por Heaps.
- Se realiza la unión de las listas de posteo de todas las palabras del vocabulario que calzaron con el patrón de la consulta. Observar que esto puede ser costoso si la consulta tiene poca precisión.
- Una vez realizado este paso la consulta se puede combinar con otros modelos.
- Para búsqueda por frases o proximidad, se obtienen las listas de las ocurrencias exactas de las palabras involucradas y se realiza una pseudo intersección (inversión, secuencial o combinado).
- Si se tiene sólo un índice de bloques, éste permitirá determinar que el patrón no aparece en algunos bloques del texto, pero en los demás se deberá recurrir a búsqueda secuencial.
- Eligiendo correctamente el tamaño del bloque se puede obtener un índice que sea sublineal en espacio extra y tiempo de búsqueda simultáneamente.
- Esta es una alternativa aceptable para colecciones medianas (200 Mb) y necesita tener acceso al texto (ej. no sirve para indexar la Web pero sí para un buscador interno del Web site).

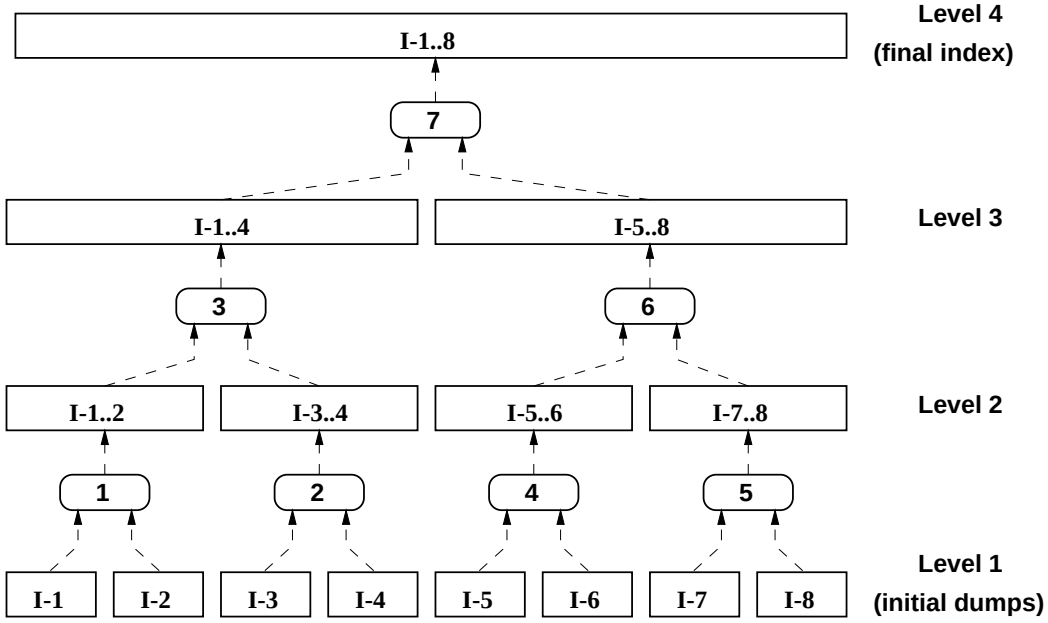


"Camioneta de ocasion" [aproximado]

Construcción de Indices Invertidos

- Se recorre la colección de texto secuencialmente.
- Para cada término leído, se busca en el vocabulario (que se mantiene en memoria).
- Si el término no existe aún, se lo agrega al vocabulario con una lista de posteo vacía.
- Se agrega el documento que se está leyendo al final de la lista de posteo del término.
- Una vez leída toda la colección, el índice se graba en disco.
- En el caso de índices de bloques, se construye sólo el posteo considerando a los bloques como documentos.
- En el caso de necesitar inversión, se almacena además de la lista de posteo de cada término una de inversión, donde se debe almacenar la posición de cada ocurrencia de ese término.
- En el caso de los índices para el modelo vectorial, la lista de posteo está ordenada por tf y dentro del tf por número de documento. A medida que vamos encontrando más y más veces el término en el mismo documento su entrada va siendo promovida más adelante en la lista del término.
- Otro método: generar las tuplas $(t_r, tf_{r,i}, i)$ y luego ordenar, pero no se puede comprimir hasta el final.

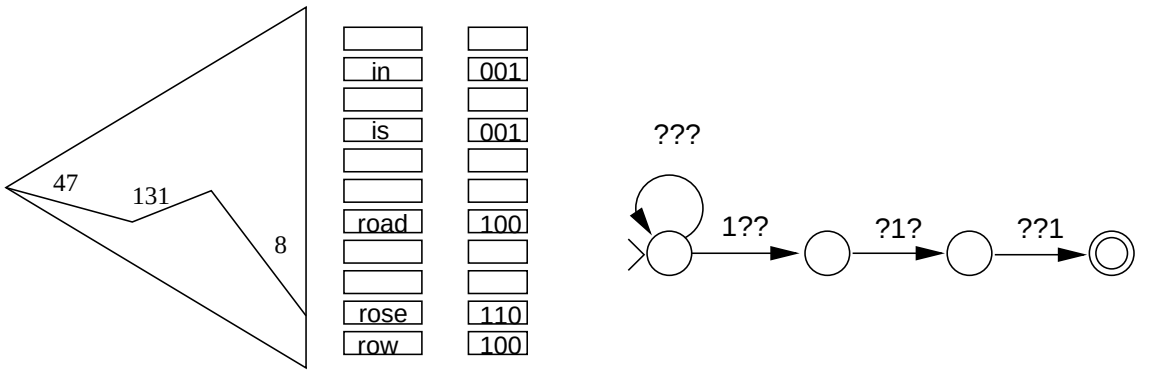
- El mayor problema que se presenta en la práctica es que, obviamente, la memoria RAM se terminará antes de poder procesar todo el texto.
- Cada vez que la memoria RAM se agota, se graba en disco un *índice parcial*, se libera la memoria y se comienza de cero.
- Al final, se realiza un *merge* de los índices parciales. Este merge no requiere demasiada memoria porque es un proceso secuencial, y resulta relativamente rápido en I/O porque el tamaño a procesar es bastante menos que el texto original.
- Aridad del merge, orden del mergeo, forma de hacer el merge.
- El método se adapta fácilmente para actualizar el índice.



Compresión de Texto en Sistemas de RI

- Una forma obvia de reducir el espacio es comprimir el texto mismo.
- Ejemplos: Ziv-Lempel (40 %), Huffman (65 %), PPM/BWT (20 %).
- Sin embargo, no cualquier tipo de compresión convive bien con un sistema de RI.
- La compresión de texto que se aplique debería permitir:
  - Obtener una buena tasa de compresión.
  - Descomprimir eficientemente.
  - Descomprimir el texto a partir de cualquier posición sin tener que descomprimir todo lo anterior.
  - Buscar en el texto comprimido sin descomprimirlo.
- Los modelos más exitosos para texto son los *basados en palabras*: los símbolos de la secuencia son las palabras y no los caracteres.
- La distribución de palabras, por Zipf, es mucho más sesgada que la de caracteres.
- La entropía de orden cero es inferior al 25 %, y de órdenes superiores llega a menos de 20 %.
- Un simple *Huffman sobre palabras* obtiene 25 % de compresión.
- El alfabeto fuente coincide con el vocabulario, lo que facilita integrarlo en un sistema de RI.

- Como el alfabeto es grande, es posible que el árbol de Huffman tenga aridad 256 y todavía la compresión es razonable (30 %).
- En este caso la salida es una secuencia de bytes, que se descomprime muy rápidamente.
- Es posible buscar desde palabras hasta expresiones complejas en el texto comprimido, *más rápido que en el texto original*.
- El resultado es una situación en que se gana por todos lados:
  - Texto más indice ocupan *menos espacio que el texto original*: 40 %-75 % según el tipo de índice.
  - El texto se busca secuencialmente *más rápido que el texto original*: 3 a 8 veces más rápido (sin contar con que tenemos un índice).
  - Se puede combinar con direccionamiento a bloques.
  - El texto siempre está comprimido y se descomprime sólo para mostrarlo.



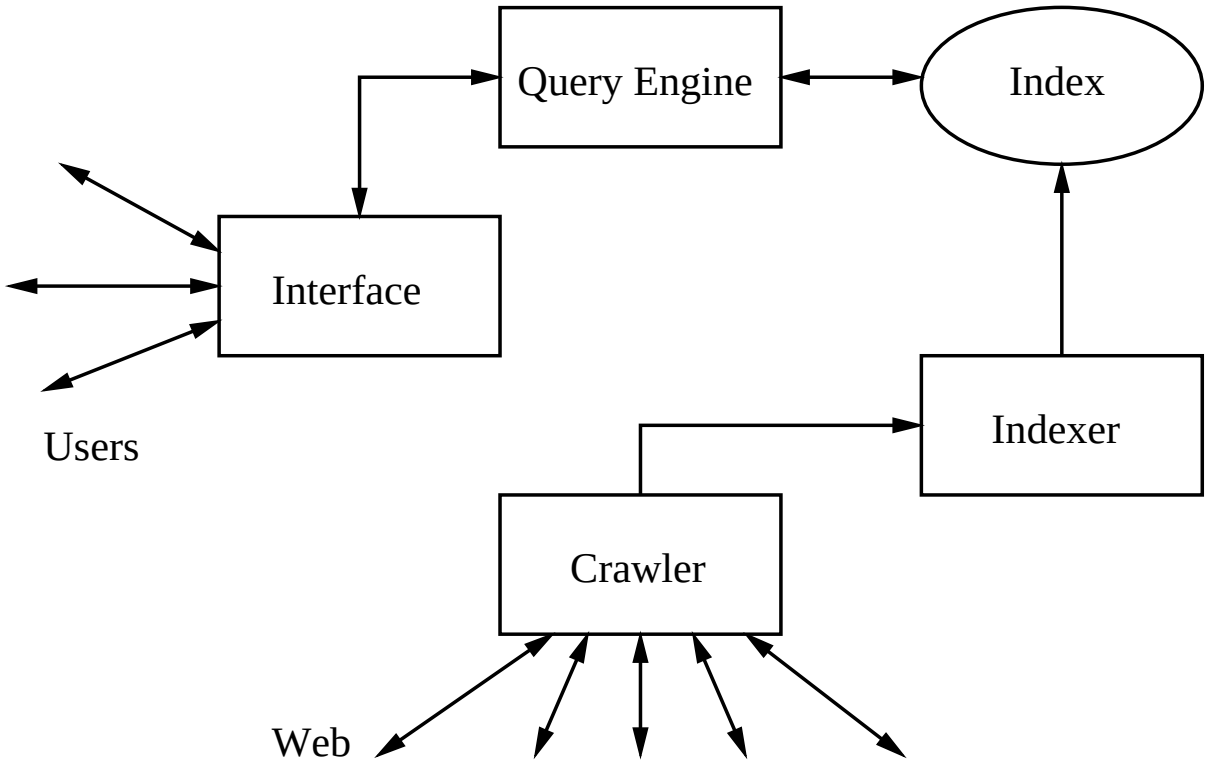
Huffman tree Vocabulary Marks Nondeterministic Searching Automaton

La Web como Repositorio de Información

- Se puede considerar como un gigantesco texto distribuído en una red de baja calidad, con contenido pobremente escrito, no focalizado, sin una buena organización y consultado por los usuarios más inexpertos.
- Principales desafíos:
  - Gigantesco volumen de texto.
  - Información distribuída y conectada por una red de calidad variable.
  - Texto altamente volátil (el 40 % cambia cada mes, y duplica su tamaño en meses).
  - Información mal estructurada y redundante (30 % de las páginas son prácticamente duplicados).
  - Información de mala calidad, sin revisión editorial de forma ni contenido (1 error de tipeo cada 200 palabras comunes y cada 3 apellidos extranjeros).
  - Información heterogénea: tipos de datos, formatos, idiomas, alfabetos.
  - Usuarios generalmente inexpertos, con consultas de 1 a 3 palabras y pobremente formuladas.
- ¡Es la peor pesadilla para un sistema de RI!

Arquitectura de las Máquinas de Búsqueda Web

- Componentes básicos
  - *Crawler:* recorre la Web buscando las páginas a indexar.
  - *Indexador:* manteniene un índice con esa información.
  - *Máquina de consultas:* realiza las búsquedas en el índice.
  - *Interfaz:* interactúa con el usuario.



- Interacciones
  - El crawler (robot, spider...) corre *localmente* en la máquina de búsqueda, recorriendo la Web mediante pedidos a los servers y trayendo el texto de las páginas Web que va encontrando.
  - El indexador corre localmente y mantiene un índice sobre las páginas que le trae el crawler.
  - La máquina de consultas también corre localmente y realiza las búsquedas en el índice, retornando básicamente las URLs rankeadas.
  - La interfaz corre en el cliente (en cualquier parte de la Web) y se encarga de recibir la consulta, mostrar los resultados, retroalimentación, etc.

Ranking en la Web

- La mayoría de las máquinas Web usa variantes del modelo booleano o vectorial. El texto no suele estar accesible al momento de la consulta, de modo que por ejemplo direccionamiento a bloques es impensable. Incluso si el texto se almacena localmente, el proceso sería demasiado lento.
- No hay mucha información publicada sobre ésto, y en todo caso medir la recuperación es casi imposible.
- Una diferencia importante entre indexar la Web y la RI normal está dada por los links. Una página que recibe muchos links se puede considerar muy popular. Páginas muy conectadas entre sí o referenciadas desde la misma página podrían tener contenido similar.
- PageRank (de Google): el valor de una página es la probabilidad estacionaria de que un proceso de recorrido automático se encuentre en ella.
- El recorrido salta a una página aleatoria con probabilidad q y se mueve a por un link aleatorio con probabilidad $(1 - q)$. Sea a una página apuntada por páginas $p_1 \ldots p_n$ y sea $C()$ la cantidad de links que salen de una página. Entonces

$$PR(a) \quad = \quad q \; + \; (1 - q) \sum_{i=1}^{n} PR(p_i)/C(p_i)$$

  donde q se suele fijar en 0,15.

Indices para la Web

- La mayoría usa variantes del índice invertido.
- No se puede almacenar el texto completo, por el volumen que representa.
- Se almacenan algunas cosas, como el título y los primeros caracteres (unos 400), fecha de creación, tamaño, etc.
- Si suponemos 500 bytes por página, sólo las descripciones requieren 50 Gb para 100 millones de páginas.
- Dados los tamaños de las páginas Web, los índices requieren aproximadamente 30 % del tamaño del texto. Para las 100 millones de páginas a 5 Kb de texto por página esto representa unos 150 Gb de índice.
- Algunos buscadores permiten búsquedas de frase y proximidad, pero no se sabe cómo las implementan.
- Normalmente no permiten búsquedas aproximadas o patrones extendidos arbitrarios, pues la búsqueda en el vocabulario suele ser impensable dada la carga de la máquina de consultas. 1 Tb de texto generaría un vocabulario de 300 Mb según Heaps, lo que requiere unos 10-300 segundos para ser recorrido secuencialmente.
- Aún así, algunos buscadores sugieren variantes a términos no encontrados o muy poco frecuentes.

- Al usuario se le presentan sólo los primeros 10–20 documentos. El resto de la respuesta se mantiene en memoria para cuando los pida, para evitar recalcularlos.
- Un problema con ésto es que el protocolo HTTP no tiene el concepto de sesión. ¿Cómo saber que el usuario no está más ahí y descartar la respuesta?
- Una idea interesante para reducir tiempos de consulta es *caching*: mantener las respuestas a las consultas más populares en memoria (¿qué es una consulta, palabras o toda la consulta?), o las páginas más populares, o los pedazos de listas invertidas más populares, etc.
- Otras capacidades interesantes que algunas máquinas tienen es la restricción por dominios, fechas, tamaños, idioma, directorios, etc.