

# Recuperación de la Información: Algoritmos, Estructuras de datos, y Búsqueda en la Web

Gonzalo Navarro  
Ricardo Baeza-Yates  
Universidad de Chile

## Objetivos del Curso

- Comprender los principales desafíos de RI y sus diferencias con otras disciplinas relacionadas.
- Poder entender la descripción de un Sistema de RI (SRI).
- Poder evaluar y elegir un SRI, poder hacer las preguntas adecuadas y entender las respuestas.
- Conocer los conceptos necesarios para implementar un SRI.
- Comprender los principales aspectos de un SRI sobre la Web.

## Bibliografía

- *Modern Information Retrieval*.  
R. Baeza-Yates y B. Ribeiro-Neto.  
Addison-Wesley, 1999.
- *Managing Gigabytes*.  
I. Witten, A. Moffat y T. Bell.  
Van Nostrand Reinhold, 2da edición, 1999.
- <http://www.searchenginewatch.com>

## Contenidos del Curso

### Introducción

#### Parte I: Modelos

1. Modelo booleano
2. Modelo vectorial
3. Modelo probabilístico
4. Evaluación de los modelos clásicos
5. Otros modelos alternativos
6. Expansión de consultas y retroalimentación
7. Precisión y recuperación
8. Búsqueda de patrones

#### Parte II: Índices

1. Índices invertidos
2. Arreglos de sufijos
3. Índices distribuidos

#### Parte III: La Web

1. Caracterizando la Web
2. Arquitectura de las máquinas de búsqueda
3. Algoritmos de ranking
4. Algoritmos de crawling
5. Índices para la Web
6. Alternativas para el usuario

## ¿Qué es Recuperación de Información?

- Recuperar datos (RD) versus recuperar información (RI):

```
SELECT name,position,salary
FROM Employees
WHERE company = "Coca-Cola" and salary >= 3,000
```

versus

Quiero información sobre las consecuencias  
de la crisis de los misiles cubanos  
en el desarrollo de la guerra fría

- Los datos se pueden estructurar en tablas, árboles, etc. para recuperar exactamente lo que se quiere. El texto no tiene estructura clara y no es fácil crearla.
- En RD se sabe exactamente lo que se quiere, en RI no existe la respuesta correcta.
- En RI, cada documento puede ser más o menos relevante, y esto puede cambiar según el usuario y la situación.
- En RD sólo importa la eficiencia (básicamente velocidad y espacio), mientras que en RI importa también (o más) la *calidad* de la respuesta.
- Dado que comprender cabalmente el significado de un texto en forma automática es imposible en la práctica, RI busca una *aproximación* a responder lo que el usuario busca.

- El problema de RI se puede definir como:  
*Dada una necesidad de información (consulta + perfil del usuario + ...) y un conjunto de documentos, ordenar los documentos de más a menos relevantes para esa necesidad y presentar un subconjunto de los más relevantes.*
- Dos grandes etapas para abordar el problema:
  - Elegir un *modelo* que permita calcular la relevancia de un documento frente a una consulta.
  - Diseñar algoritmos y estructuras de datos que lo implementen eficientemente (índices).
- Cuán bien hecha está la primera etapa se mide comparando las respuestas del sistema contra las que un conjunto de expertos consideran relevantes.
- Cuán bien hecha está la segunda etapa se mide considerando el tiempo de respuesta del sistema, espacio extra de los índices, tiempo de construcción y actualización del índice, etc.
- En general, RI no congenia bien con el modelo relacional. Las extensiones hechas a los RDBMS para incorporar texto son limitadas y no todo lo eficientes que se necesita.
- RI requiere modelos y algoritmos especializados.

## Parte I

### Modelos

*O cómo adivinar lo que el usuario realmente quiso preguntar*

#### Modelo Booleano

- La relevancia es binaria: un documento es relevante o no lo es.
- Consultas de una palabra: un documento es relevante si 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: lo mejor de Maradona

**Maradona AND Mundial**  
**AND ((Mexico'86 OR Italia'90) BUTNOT U.S.A.'94)**

- 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:

**"Necesito investigar sobre los Aztecas y sobre los Incas" → Aztecas AND Incas**

(grave error, se perderán excelentes documentos que traten una sola de las culturas en profundidad, debió ser **Aztecas OR Incas**)

- ¿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 excluir documentos.
  - Puede ser útil con mejores interfaces.

## Modelo Vectorial

- 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*.
- Se puede enriquecer esto con procesos de *lematización* (o *stemming*), *etiquetado*, e identificación de frases.
- Sea  $\{t_1, \dots, t_k\}$  el conjunto de términos y  $\{d_1, \dots, d_N\}$  el de documentos.
- Un documento  $d_i$  se modeliza como un vector

$$d_i \rightarrow \vec{d}_i = (w(t_1, d_i), \dots, w(t_k, d_i))$$

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

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

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  $t_r$ .

- La similitud es un valor entre cero y uno.
- Notar que dos documentos iguales tienen similitud 1, y ortogonales (si no comparten términos) tienen similitud 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

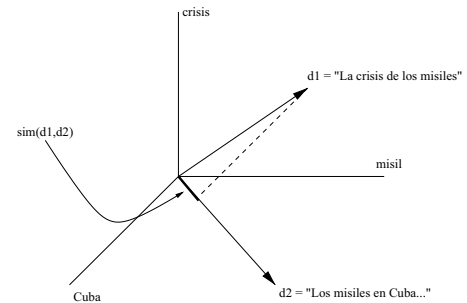
$$sim(d_i, q) = \sum_{r=1}^k w_{r,i} \times w_{r,q} \sim \sum_{t_r \in q} w_{r,i} \times idf_r$$

- La última fórmula no es idéntica a *sim* pero ordena los documentos de la misma forma (falta dividir por  $|q|$ ).
- 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.

- 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 *similitud* entre dos documentos mediante la *distancia coseno*:

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

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



## Modelo Probabilístico

- Se presupone que existe exactamente un subconjunto de documentos que son relevantes para una consulta dada.
- Para cada documento, se intenta evaluar la *probabilidad* de que el usuario lo considere relevante.
- La relevancia de un documento  $d$  se calcula como

$$\frac{P(d \text{ relevante para } q)}{P(d \text{ no relevante para } q)}$$

- Esto se traduce en la siguiente fórmula

$$sim(d_i, q) = \frac{P(R|d_i)}{P(\bar{R}|d_i)} = \frac{P(d_i|R) \times P(R)}{P(d_i|\bar{R}) \times P(\bar{R})} \sim \frac{P(d_i|R)}{P(d_i|\bar{R})}$$

donde  $P(R|d_i)$  es la probabilidad de que  $d_i$  sea relevante,  $P(R)$  la probabilidad de que un documento cualquiera sea relevante, y  $P(d_i|R)$  la probabilidad de elegir aleatoriamente  $d_i$  entre los documentos relevantes.

- Luego de bastante manejo algebraico, suponiendo independencia de los términos y tomando logaritmos, se llega a

$$sim(d_i, q) \sim \sum_{r=1}^k w_{r,i} \times w_{r,q} \times \left( \log \frac{P(t_r|R)}{1 - P(t_r|R)} + \log \frac{1 - P(t_r|\bar{R})}{P(t_r|\bar{R})} \right)$$

$$= \sum_{t_r \in q \cap d_i} \left( \log \frac{P(t_r|R)}{1 - P(t_r|R)} + \log \frac{1 - P(t_r|\bar{R})}{P(t_r|\bar{R})} \right)$$

donde  $P(t_r|R)$  es la probabilidad de que  $t_r$  aparezca en un documento relevante, y  $w_{r,i}$  es 1 si  $t_r$  aparece en  $d_i$  y cero sino.

- Inicialmente se supone  $P(t_r|R) = 0.5$  y  $P(t_r|\overline{R}) = n_r/N$ .
- Luego de una iteración se recuperan  $V$  documentos; sea  $v_r$  el número de documentos recuperados que contienen el término  $t_r$ . Se recalcula  $P(t_r|R) = v_r/V$  y  $P(t_r|\overline{R}) = (n_r - v_r)/(N - V)$ .
- El modelo probabilístico, en teoría, recupera los documentos que con mayor probabilidad son relevantes. Sin embargo, es bastante poco popular.
- ¿Por qué?
  - Se debe comenzar adivinando y luego refinar esa apuesta iterativamente.
  - El modelo no toma en cuenta el factor  $tf$ ; sino que ve cada documento como un conjunto de términos.
  - Necesita presuponer que los términos son independientes.
  - Existen estudios que muestran que es inferior al modelo vectorial.
  - Toda la comunidad científica lo considera inferior al modelo vectorial.
- Sin embargo, tiene una base teórica distinta a la del modelo vectorial y permite extensiones que sí son populares.

- Otro elemento es el uso de tesauros y sinónimos para expandir la consulta, de modo de poder recuperar  
**vendo camioneta usada**  
 frente a la consulta  
**autos de segunda mano**  
 Sin embargo, mantener un buen tesoro es costoso en términos de trabajo manual y no suele ser posible mantenerlo al día con las expresiones que van apareciendo.
- Además es sabido que un tesoro global no funciona siempre bien, por ejemplo "**estrella**" puede ser una generalización de "**supernova**" pero no en un contexto que habla de estrellas de televisión.
- Se puede utilizar información del texto como la estructura para refinar mejor la consulta (ej. un término que aparece en el título debe ser más relevante que en un pie de página).
- Se pueden utilizar distintas técnicas para descubrir que ciertos términos están correlacionados. A esto apuntan los modelos alternativos y las técnicas de expansión de consultas que veremos a continuación.

## Evaluación de los Modelos Clásicos

- El modelo clásico más popular es el vectorial, por ser simple, fácil y eficiente de implementar, y entregar buenos resultados. En muchos casos las aplicaciones llegan hasta aquí.
- Sin embargo, todos los modelos clásicos tienen ciertas falencias comunes, la más notoria de las cuales es la incapacidad para capturar las relaciones entre términos.
- Por ejemplo, si busco  
**guerra fría**  
 quisiera recuperar un documento que habla sobre  
**la crisis de los misiles cubanos**  
 Sin embargo, el sistema no tiene idea de que ambas cosas están relacionadas y la intersección de vocabulario puede ser nula.
- Parte de la solución pasa por el análisis lingüístico:
  - lematizar (para no perderse variantes de la misma palabra),
  - etiquetar (para distinguir verbos de sustantivos),
  - detectar frases comunes (para no recuperar información sobre heladeras cuando se pregunte por "**guerra fría**").
  - Es posible hacer un análisis lingüístico más fino, pero la tecnología existente tiene sus limitaciones.

## Modelos Alternativos

Estos modelos son en general bastante costosos de implementar, y no siempre dan grandes resultados. Por ello son en general poco populares, con la excepción de LSI y las redes neuronales y Bayesianas.

- Extensiones al modelo Booleano
  - Booleano Extendido
  - Conjuntos Difusos
- Extensiones al modelo Vectorial
  - Vectorial generalizado
  - LSI: Latent Semantic Indexing
  - Redes neuronales
- Extensiones al modelo Probabilístico
  - Redes Bayesianas
  - Redes de Inferencia Bayesiana

Conjuntos Difusos

- Un elemento tiene un “grado de pertenencia” (entre 0 y 1) a un conjunto (en particular, de los documentos a la respuesta).
- El **AND**, **OR** y **BUTNOT** operan sobre estos grados de pertenencia como *min*, *max* y “*min(A, 1 – B)*”.
- El grado de pertenencia de *d<sub>i</sub>* a *t<sub>r</sub>*, se calcula con alguna fórmula de relevancia como *w<sub>i,r</sub>* u otras más sofisticadas como

$$\mu(t_r, d_i) = 1 - \prod_{t_i \in d_i} (1 - c_{s,r})$$

(recordar  $\forall c_{s,r} = \sim (\wedge \sim c_{s,r})$ ) donde

$$c_{s,r} = \frac{n_{s,r}}{n_s + n_r - n_{s,r}}$$

es un indicador de cuán correlacionados están los términos *s* y *r* (*n<sub>s,r</sub>* es el número de documentos donde aparecen ambos términos).

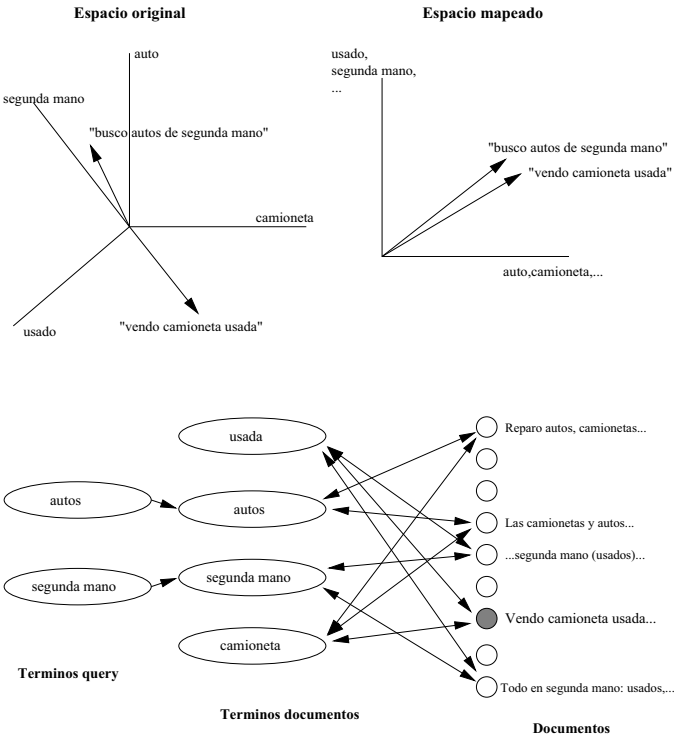
- Con esto el modelo captura documentos donde ocurren otros términos relacionados con los de la consulta.
- Por ejemplo, si muchos documentos que hablan de “**guerra fría**” hablan de “**misiles cubanos**” entonces el *c<sub>j,r</sub>* de los términos será alto, y una consulta sobre “**guerra fría**” podrá recuperar documentos que hablan solamente de “**misiles cubanos**” .

LSI: Latent Semantic Indexing

- La idea es mapear el espacio de vectores a uno de menor dimensión conservando lo mejor posible las distancias entre los vectores. Existen métodos matemáticos bien conocidos (SVD) y heurísticas para hacer esto.
- Lo que ocurre es que se selecciona un conjunto de vectores relativamente independientes, y los dependientes se colapsan en una sola coordenada.
- La idea es que los vectores del espacio reducido representan conceptos más que términos, y que ésto reducirá el “ruido” de las palabras individuales.
- Por ejemplo, si “**auto**” y “**vehículo**” tienden a aparecer en los mismos documentos, los dos vectores serán relativamente dependientes y colapsarán. Luego, al preguntar por uno de ellos se recuperará el otro también.

Redes Neuronales y Bayesianas

- La idea general es tratar de expandir los términos de la consulta por iteración.
- Es un grafo dirigido cuyos nodos son los términos de la consulta, los términos de los documentos y los documentos en sí. Las flechas representan “activación”.
- Los términos de la consulta activan los mismos términos de los documentos.
- Los términos de documentos activan los documentos donde aparecen.
- Estos documentos activan todos los términos que contienen.
- Esto itera con intensidades decrecientes hasta converger.
- El nivel alcanzado por cada documento es su relevancia.
- Esto permite que se activen documentos en forma indirecta, cuando contienen un término que en los documentos de la colección aparece frecuentemente junto a uno de la consulta.
- Las redes Bayesianas se basan en la misma idea pero usan un modelo distinto para los pesos y otras flechas, y no siempre usan iteración.



## Expansión de Consultas

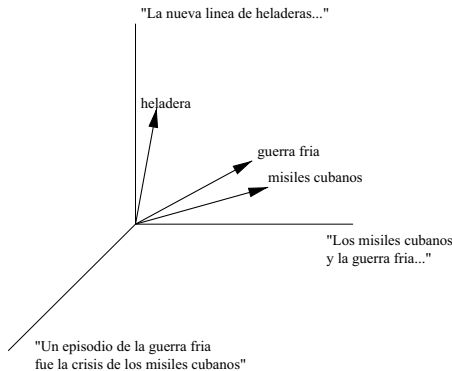
- Es posible asimismo expandir una consulta para mejorar la relevancia del resultado sin alterar el modelo original.
- Un modelo posible es considerar un espacio vectorial donde los *términos son los puntos y los documentos las coordenadas*:

$$t_r \longrightarrow \vec{t}_r = (w_{1,r}, \dots, w_{N,r})$$

- La similitud entre términos se define exactamente como la de documentos:

$$\text{sim}(t_r, t_s) = \vec{t}_r \cdot \vec{t}_s = \sum_{i=1}^N w_{i,r} \times w_{i,s}$$

- Si dos términos tienden a aparecer en los mismos documentos sus vectores serán cercanos.



- Dada una consulta formada por términos, éstos se consideran vectores y la consulta se expande con todos los vectores (términos) cercanos a los de la consulta.
- Otra forma de usar estas distancias es formar *clusters* de términos considerados similares, mediante cualquier método jerárquico (la idea es construir un tesoro automáticamente), y luego recuperar el cluster (a cierto nivel) de la consulta.
- El clustering puede hacerse en forma “local”, considerando solamente los documentos recuperados como relevantes, en un estilo parecido al de redes neuronales. Puede definirse la intensidad de la asociación por la distancia en el texto entre los términos (clusters métricos), la fórmula  $c_{s,r}$  de conjuntos difusos (clusters de asociación), o la previa donde los términos son puntos (clusters escalares).

## Retroalimentación (Relevance Feedback)

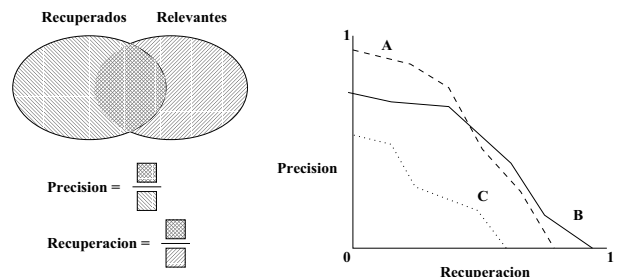
- Finalmente, puede preguntársele al usuario cuáles documentos de los recuperados son relevantes y cuáles no, y volver a pesar todo en función de la respuesta.
- Sea  $V$  el conjunto de documentos recuperados, que el usuario divide en  $V_r$  (relevantes) y  $V_n$  (no relevantes). La fórmula clásica de Rochio modifica el vector  $\vec{q}$  de la siguiente forma

$$\vec{q}_m = \alpha \vec{q} + \frac{\beta}{|V_r|} \sum_{d_i \in V_r} \vec{d}_i - \frac{\gamma}{|V_n|} \sum_{d_i \in V_n} \vec{d}_i$$

- La idea es “acercar” el vector  $\vec{q}$  al conjunto de documentos relevantes y “alejarse” del conjunto de documentos no relevantes.
- Se puede particularizar según el caso. Por ejemplo, puede ser mucho pedir que el usuario marque los documentos no relevantes. En ese caso se puede usar  $\gamma = 0$  (retroalimentación positiva).
- El “more like this” de varios buscadores Web utiliza una versión sencilla de esta fórmula.

## 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.

- Es fácil tener alta precisión: basta con retornar el documento más relevante, pero entonces la recuperación es cercana a 0. Es fácil tener alta recuperación: basta retornar todos los documentos, pero ahora la precisión es cercana a 0. El objetivo es tratar de aumentar ambos al mismo tiempo.

## Búsqueda de Patrones

- Es el problema de encontrar las porciones del texto que calzan con un patrón dado.
- Normalmente surge cuando el texto no contiene lenguaje natural occidental (o sea palabras). Ejemplos: bases de datos de ADN o proteínas, lenguajes orientales, sonido o MIDI.
- Incluso en RI normal sobre lenguaje natural, el problema de búsqueda de patrones se presenta de diversas formas.
- Un caso esencial es la búsqueda de frases y palabras próximas en el texto. Para hallarlas no basta saber que las palabras están en el documento, sino que se necesita conocer su posición. Por ejemplo, se puede querer decir

**auto\* CERCA DE usado\***

para calzar

... vendo auto usado ...

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

... permuta por auto o camioneta usados ...

Observar que si no se exige que las palabras estén cerca la consulta puede retornar muchos documentos irrelevantes.

- 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.
- Todos los sistemas de RI modernos permiten algún grado de búsqueda de patrones, combinándolo con algún otro modelo.
- 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 y aproximada?

## Parte II

### Indices

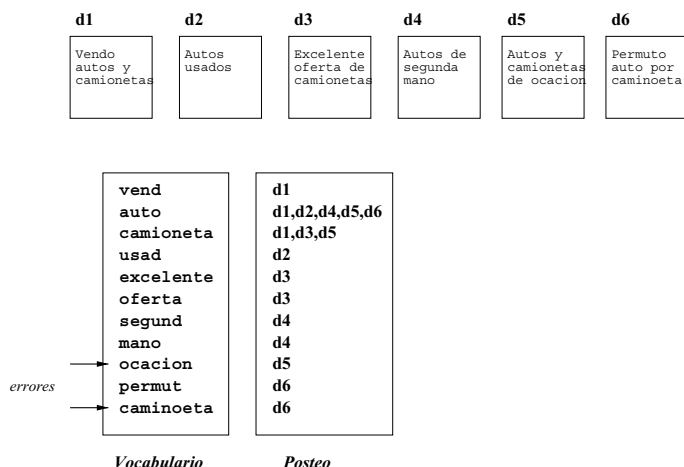
*O cómo, además de adivinar,  
no tardarse demasiado*

## 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 los modelos booleano y probabilístico:
  - 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 búsqueda de patrones:
  - 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.

## Indices Invertidos: Espacio Extra

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

$$k = C \times n^\beta$$

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

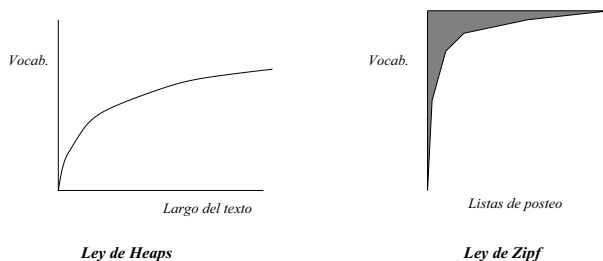
$$n_r = \frac{N}{r^\theta H_N(\theta)}$$

donde

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

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.

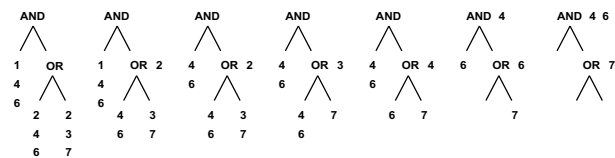


- Posteo/Inversión:
  - Caso booleano/probabilístico: los números de documentos son crecientes, se pueden almacenar las diferencias (Elías). Así comprimido, requiere un 10%-25% extra sobre el texto.
  - Caso vectorial: eso ya no ocurre, pero todos los documentos con el mismo  $tf$  (muchos, por Zipf) pueden aún ordenarse. El índice requiere 15%-30% extra.
  - Búsqueda de patrones: 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.
  - Se puede comprimir el texto a 25%-30% del espacio original (Huffman sobre palabras).

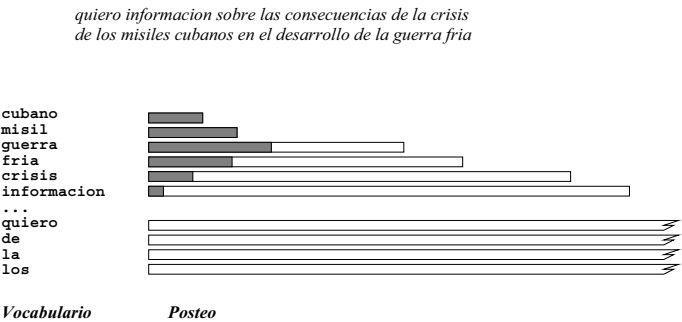


Indices Invertidos: Consulta en el Modelo Booleano

- 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 (restricciones de metadatos).
- 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 (restricciones AND, BUT, NOT).
- En el modelo probabilístico es similar, pues interesa el OR de los términos de la consulta.



- 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.

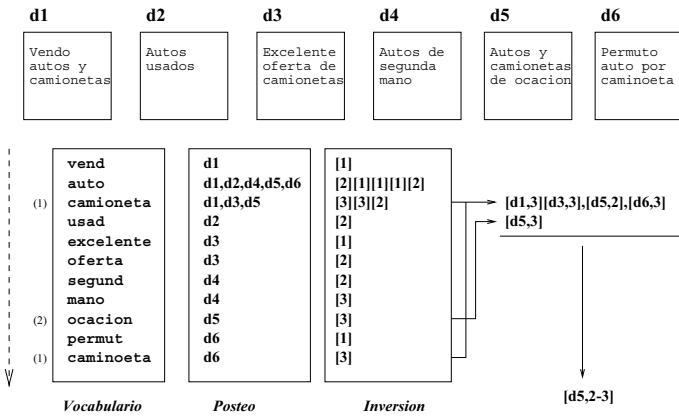


Indices Invertidos: Consulta en el Modelo Vectorial

- 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.

Indices Invertidos: Búsqueda de Patrones

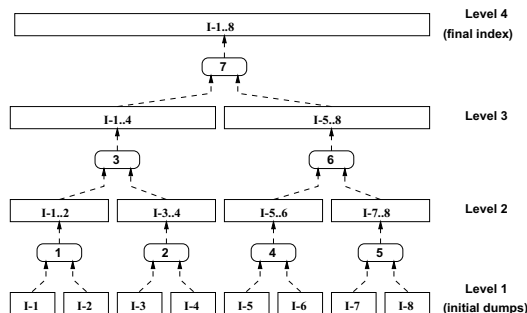
- Se comienza buscando la consulta en el vocabulario. Esto no es tan trivial como antes porque la consulta puede ser una expresión regular, búsqueda aproximada, etc.
- En estos casos se realiza una búsqueda *secuencial* en el vocabulario. Esto es tolerable por Heaps.
- Se realiza la unión de las listas 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 (compresión!).
- Se ha probado que 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).



## Indices Invertidos: Construcción

- 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.
- Con texto comprimido: una pasada para generar vocabulario y frecuencias, otra para indexar y comprimir a la vez.

- 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.



## Arreglos de Sufijos

- Es un índice que no necesita que el texto esté formado por palabras.
- Es capaz de recuperar cualquier substring del texto, buscar expresiones regulares y realizar búsqueda aproximada en tiempo sublineal.
- Resulta ideal para biología computacional, lenguajes orientales, etc.
- En RI tradicional, puede resultar útil para
  - Análisis lingüístico especializado (ej. encontrar la autorrepetición más larga de una colección, permitiendo errores).
  - Búsqueda de frases de varias palabras (es más rápido que un índice invertido).
  - Búsqueda de patrones que trascienda palabras (ej. el índice invertido no se recupera de errores que involucren separadores).
- El espacio que requiere es cercano al 40% del texto.
- Es más costoso de construir y mantener que un índice invertido.
- Es superior al índice invertido para búsqueda de patrones, pero no para las otras operaciones típicas en RI.

## Arreglos de Sufijos: Estructura

- El arreglo de sufijos es simplemente un array con los punteros a todas las posiciones de interés en el texto.
- Cada posición define un *sufijo* del texto.

to be at the beach or to be at work, that is the real question \$ **Text**

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ **Index points**

1 4 7 10 14 20 23 26 29 32 38 43 46 50 55

1: to be at the beach or to be at work, that is the real question  
4: be at the beach or to be at work, that is the real question  
7: at the beach or to be at work, that is the real question  
10: the beach or to be at work, that is the real question  
.....  
55: question

- En el arreglo, las posiciones están ordenadas *lexicográficamente* por los sufijos.

The diagram shows the construction of a suffix array for the text "to be at the beach or to be at work, that is the real question \$".

**Suffix Array:** A row of 15 boxes containing the indices 7, 29, 4, 26, 14, 43, 20, 55, 50, 38, 10, 46, 1, 23, 32. Above each box is its position in the array (1 to 15).

**Text:** "to be at the beach or to be at work, that is the real question \$".

**Index points:** A row of 15 boxes containing the indices 1, 4, 7, 10, 14, 20, 23, 26, 29, 32, 38, 43, 46, 50, 55. Arrows point from each index in the suffix array to its corresponding character in the text.

## Arreglos de Sufijos: Búsqueda

- Todo substring del texto es el prefijo de un sufijo.
- La relación *prefijo de* puede traducirse a relaciones de orden lexicográfico:

$$x : a \text{ prefijo de } y \iff (x : a < y) \wedge (y < x : (a + 1))$$

- Por lo tanto, basta un par de búsquedas binarias en el arreglo de sufijos para obtener el rango del arreglo que contiene todas las ocurrencias de  $x$  (en tiempo logarítmico).

- Una forma menos efectiva de realizar lo mismo es ir considerando las letras del patrón una a una y restringiendo el rango del resultado.
- Esta idea se puede extender para buscar expresiones regulares y permitir diferencias en el calce.
- Por cada nuevo carácter del patrón se consideran todas las letras que pueden seguir y el estado resultante de la búsqueda. Mediante *backtracking* se obtienen todos los rangos de interés.
- El costo de búsqueda en este caso es  $O(n^\lambda \log n)$ , para algún  $0 < \lambda < 1$  dependiente del problema.
- Esto puede recuperar errores de separadores, por ejemplo "**flores**" en textos como

flo res

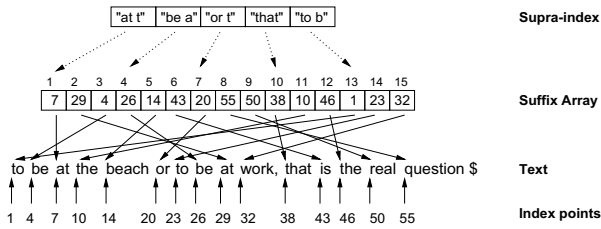
O

lasflores

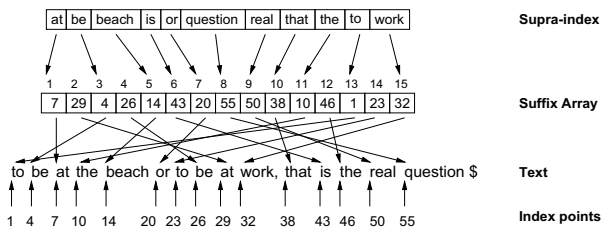
cosa que un índice invertido no resuelve.

- Trie y Arbol de sufijos
  - Trie con todos los sufijos del texto.
  - Búsqueda simple en tiempo  $O(m)$ , compleja  $O(n^\lambda)$ .
  - Cada nodo es un intervalo en el arreglo de sufijos.
  - El árbol comprime caminos unarios,  $O(n)$  espacio y tiempo de construcción.
  - Ocupan mucho espacio, 12 a 60 veces el tamaño del texto.
  - Malos en memoria secundaria.
  - Puede utilizarse la idea para búsqueda secuencial en el vocabulario.

- En memoria secundaria la búsqueda binaria puede resultar costosa. El problema no es tanto el acceso al arreglo de sufijos sino a las posiciones aleatorias del texto.
- Se pueden usar *supraíndices* en RAM para reducir este costo.



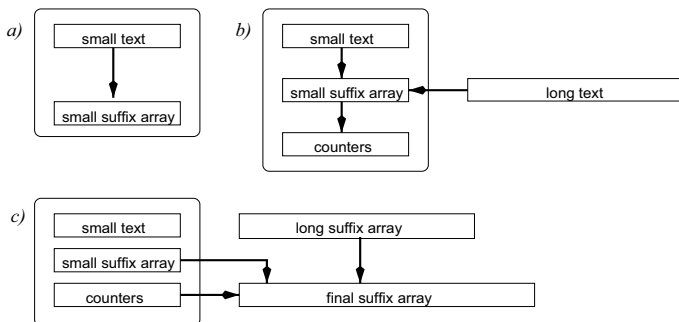
- Un supraíndice interesante puede ser el mismo vocabulario.



- En este caso la estructura es muy similar a un índice invertido para búsqueda de patrones (con inversión).

## Arreglo de Sufijos: Construcción

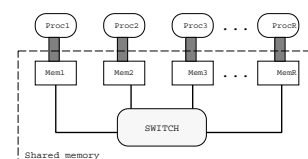
- En principio, no es más que ordenar un arreglo.
- Sin embargo, se puede aprovechar la estructura del problema: los sufijos son sufijos de otros sufijos.
- Existen algoritmos (Manber & Myers) que en promedio toman tiempo  $O(n \log \log n)$ .
- Sin embargo, el verdadero problema aparece en memoria secundaria, por los accesos aleatorios al texto.
- La mejor solución ( $O((n^2/M) \log M)$ ) es la siguiente:
  - Cortar el texto en bloques que se puedan indexar en memoria.
  - Traerse el primer bloque, construir su arreglo y mandarlo a disco.
  - Al procesar el bloque  $i$ , el invariante es que el arreglo de sufijos para los  $i - 1$  bloques anteriores está construido.
  - Traer el bloque  $i$ -ésimo y construir su arreglo de sufijos.
  - Leer el texto de los  $i - 1$  bloques anteriores y buscar cada sufijo en el arreglo del bloque  $i$ -ésimo.
  - Determinar así cuántos sufijos del arreglo grande van entre cada par de posiciones del arreglo chico.
  - Realizar el merge secuencial de los dos arreglos con esa información.



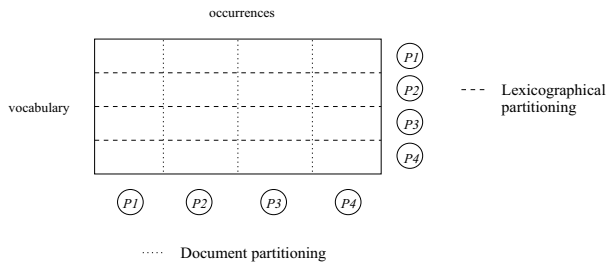
- El método se adapta fácilmente para actualizar el índice.
- En texto comprimido se reduce el espacio, tiempo de construcción y de búsqueda. Conviene usar Hu-Tucker en vez de Huffman.

## Indices Distribuidos

- En muchos casos, por bueno que sea el algoritmo de indexación o de búsqueda, no es suficiente para cubrir la demanda:
  - El texto es demasiado grande
  - La frecuencia de actualización es demasiado alta
  - Llegan demasiadas consultas por segundo
  - La velocidad de los discos no está creciendo al ritmo necesario.
- Una alternativa es utilizar paralelismo. Bien diseñado, puede expandir la capacidad de procesamiento tanto como se quiera.
- Las redes muy rápidas formadas por unas pocas máquinas muy potentes se han convertido en una alternativa de bajo costo.
- En estas redes el acceso remoto cuesta aproximadamente lo mismo que el acceso al disco local.
- Normalmente todos los procesadores pueden comunicarse de a pares sin causar congestión.
- Se puede considerar el total de RAMs como una gran memoria distribuida.



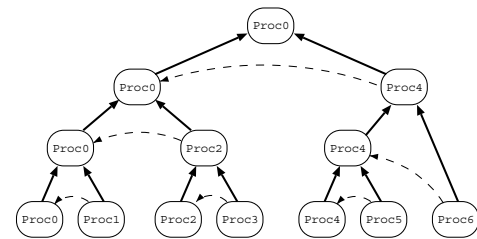
- Podemos utilizar un conjunto de máquinas de diversas formas:
  - Para construir (o actualizar) el índice en paralelo.
  - Para replicar el índice (si no es muy grande).
  - Para particionar el texto.
  - Para particionar los términos.



- Dos medidas de interés para las consultas
  - *Throughput*: cantidad de consultas respondidas por segundo.
  - *Tiempo de respuesta*: tiempo que demora una consulta particular.

## Generación Distribuida de Índices Invertidos

1. Se distribuye el texto entre las máquinas equitativamente.
2. Cada máquina construye su índice invertido local.
3. Se aparean de a dos, jerárquicamente, hasta que una sola contiene todo el vocabulario.
4. Esa máquina calcula qué parte del vocabulario será responsabilidad de cada procesador, y distribuye esa información.
5. Los procesadores se aparean todos con todos intercambiando las listas de posteo.
6. Secuencialmente transmiten su parte del índice a un disco central, donde se concatenan para formar un índice centralizado.



## Tipos de Índices Invertidos Distribuidos

- Si nos detenemos en el punto (2), tenemos un índice particionado por documentos.
  - Para consultar se debe distribuir la consulta a todos los procesadores y luego integrar los resultados.
  - El trabajo se reparte bien, pero como el tiempo es sublineal, el throughput no escala idealmente.
  - El tiempo de respuesta se reduce un poco.
  - Para el modelo booleano es ideal porque no transmite información redundante por la red.
  - Para el modelo vectorial debe seleccionar lo mejor de todos los rankings, pero presenta un problema grave: los rankings locales no coinciden con el ranking global.
  - Otro problema posible son los documentos muy populares que pueden generar una carga mal repartida.
  - En sistemas heterogéneos suele ser la única opción.
- Si nos detenemos en el punto (4), tenemos una versión de lo anterior que resuelve el problema del ranking inconsistente.

- Si nos detenemos en el punto (5), tenemos un índice particionado por léxico.
  - Para consultar se pide a cada procesador que resuelva las palabras de las que es responsable y luego se coopera para integrar los resultados.
  - En consultas cortas maximiza el throughput, pero el tiempo de respuesta puede no variar porque toda la consulta la sigue resolviendo un procesador (salvo si hay congestión).
  - En consultas complejas tiene el problema de enviar demasiada información por la red (para el caso booleano). Mismo algoritmo que con las listas.
  - Para el modelo vectorial es problemático responder con exactitud por la misma razón que cuando se cortaba el recorrido de listas invertidas. Mismo algoritmo que con las listas.
  - Para el Web es bueno porque la mayoría de las consultas son muy cortas y suele haber congestión, de modo que escala casi idealmente.
  - Un problema posible son los términos muy populares que pueden generar una carga mal repartida.
- Si llegamos hasta el final tenemos un índice centralizado (construido en forma distribuida), que puede ser replicado en todos los procesadores
  - Sólo es factible si una sola máquina puede manejar toda la colección.
  - Permite repartir idealmente el trabajo y es tolerante a fallas.

## Parte III

### La Web

*O cómo adivinar, y rápido,  
lo que la mitad del planeta quiere decir  
y lo que la otra mitad quiere saber.*

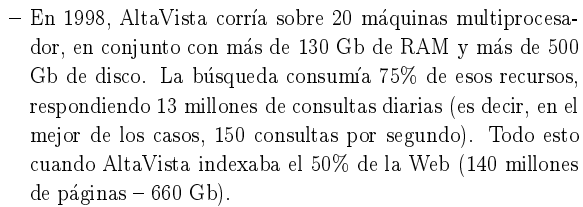
- Otras estadísticas de interés
  - Web sites no uniformes: el 1% de los sites tiene el 50% de las páginas.
  - Tamaños de documentos no uniformes: en promedio una página tiene como 8 Kb de texto (es casi tan malo como indexar posiciones exactas); el 93% de las páginas mide menos de 10 Kb, pero hay una cantidad no despreciable de documentos grandes.
  - En promedio de una página salen 8 links, la mayoría locales. La “red” no está muy densamente conectada: el 80% de las home pages recibe menos de 10 links desde fuera.
  - Yahoo! es quien más links hacia fuera tiene. Microsoft, Netscape, Yahoo! y las mejores universidades de U.S.A. son quienes reciben más links. Los directorios como Yahoo! son quienes verdaderamente “conectan” la Web.
  - 70% de las páginas están en inglés, 3% en español.

### Caracterizando la Web

- Se puede considerar como un gigantesco texto distribuido 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 (datos de 2000):
  - Gigantesco volumen de texto (más de 10 Tb sólo de texto)
  - Información distribuida y conectada por una red de calidad variable (27 millones de Web sites, más de 2000 millones de páginas).
  - Texto altamente volátil (el 40% cambia cada mes, y duplica su tamaño en 9 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.
- ¡Es la peor pesadilla para un sistema de RI!

### Arquitectura de las máquinas de búsqueda

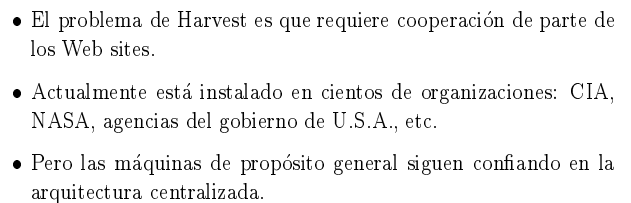
- Componentes básicos
  - *Crawler*: recorre la Web buscando las páginas a indexar.
  - *Indexador*: mantiene un índice con esa información.
  - *Máquina de consultas*: realiza las búsquedas en el índice.
  - *Interfaz*: interactúa con el usuario.
- Arquitectura centralizada (ej. AltaVista)
  - El crawler (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.



- Problemas de una arquitectura centralizada

- El crawling es la actividad más lenta, y posiblemente en el futuro el tamaño de la Web y su dinamismo hagan imposible que se realice en forma centralizada. Actualmente las mejores máquinas indexan sólo un 25% de la Web. Notar que esto no se resuelve haciendo el crawling con varias máquinas!
- Los servidores Web se saturan con pedidos de diferentes crawlers.
- Las redes se saturan innecesariamente porque los crawlers traen toda la página, para luego descartar casi todo su contenido.
- Cada crawler recolecta independientemente, sin cooperar.
- Se deben cumplir las normas de cortesía (robots.txt, meta-tags)

- Se basa en dos tipos de componentes, *gatherers* y *brokers*.
- *Gatherer*: recolecta páginas de ciertos servidores Web y extrae la información a indexar, periódicamente.
- *Broker*: indexa la información de determinados *gatherers* y otros *brokers* y responde consultas.
- Se pueden organizar de distintas formas.
- Por ejemplo, un *gatherer* puede ser local y no generar tráfico, y enviar la información a varios *brokers*, evitando repetir la recolección.



## Algoritmos de Ranking

- 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 clustering métrico es impensable. Incluso si el texto se almacena localmente, el proceso es 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. Algunos métodos modernos explotan esta información.
- Por ejemplo, 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.
- Ej. WebQuery utiliza métodos “locales”: a partir de la respuesta normal, re-rankeo usando información de cuán conectadas están las páginas y trae otras páginas que reciban muchas conexiones del conjunto ya recuperado.
- PageRank, el esquema utilizado por Google, considera la probabilidad de visitar una página realizando una navegación aleatoria en la que nos aburrimos con probabilidad  $q$  y escogemos al azar una página cualquiera de la Web. Sino, seguimos con la misma probabilidad cualquier link en la página. Es decir, tenemos que

la probabilidad de visitar la página  $p$  está dada por:

$$PR(p) = \frac{q}{T} + \frac{(1-q)}{|V|} \sum_V PR(v_i)$$

donde  $T$  es el número total de páginas y  $V = \{v_i\}$  es el conjunto de páginas apuntadas por  $p$ . Cuando realizamos una búsqueda, seleccionamos las páginas que calzan con la consulta y proyectamos estas páginas al orden ascendente dado por  $PR$ . La respuesta queda entonces ordenada por popularidad.

- Otro esquema (HITS) toma el conjunto  $S$  de páginas de la respuesta más las que están a un link de distancia (directo o inverso). Llama *autoridades* a las páginas que reciben muchos links desde  $S$  (que deberían tener contenido relevante) y *hubs* a las que apuntan mucho a  $S$  (deberían apuntar a contenido similar).
- HITS considera que los buenos hubs y autoridades están conectadas, y los evalúa usando

$$H(p) = \sum_{u \in S, p \rightarrow u} A(u), \quad A(p) = \sum_{v \in S, v \rightarrow p} H(v),$$

- Un problema con esto es que el contenido se puede difuminar, pero se puede resolver utilizando ranking clásico.
- Las pruebas muestran que la precisión y recuperación mejoran significativamente.
- ¿Es bueno rankear por popularidad? ¿No terminaremos viendo novelas y programas de animación como en la TV? ¿El efecto se retroalimenta!

## Algoritmos de Crawling

- Se comienza desde un conjunto de URLs muy populares o enviadas explícitamente por administradores de Web sites, y se siguen los links desde allí recursivamente, evitando repeticiones.
- El recorrido puede ser breadth-first (cobertura amplia pero no profunda) o depth-first (cobertura vertical).
- Es complicado coordinar varios crawlers para que no repitan trabajo. Una alternativa utilizada es que se repartan los dominios.
- Lo que vemos en un índice es como las estrellas del cielo: jamás existió. Cada página se indexó en un momento distinto del tiempo (pero al ir a ella obtenemos el contenido actual).
- Las páginas suelen tener entre 1 día y 2 meses de antigüedad, 2%-9% de los links almacenados son inválidos. Las páginas enviadas a indexar explícitamente demoran pocos días o semanas en ser indexadas, las no enviadas entre semanas y dos meses.



- Los mejores crawlers recorren unas 10 millones de páginas por día. Requerirían un mes para rotar todo el Web, en el mejor caso. En la práctica ni siquiera alcanzan a indexar gran parte.
- Para complicar las cosas, existen protocolos de buen comportamiento para crawlers, de modo que no saturen a los servidores Web.
- Algunos crawlers aprenden la frecuencia de actualización de las páginas e indexan las más volátiles más frecuentemente (ej. la página de la CNN versus mi página personal).
- Google utiliza el algoritmo de ranking de páginas para determinar el orden de crawling, y obtiene muy buenos resultados.
- Las páginas generadas dinámicamente son actualmente uno de los mayores problemas para los crawlers. Para peor, esta tendencia se está acentuando.

- 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 esto 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.

## 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, máquinas como Altavista sugieren variantes a términos no encontrados o muy poco frecuentes.

## Alternativas para el Usuario

- Puede utilizar directorios como Yahoo! (bueno cuando se sabe exactamente lo que se busca) o buscadores generales (bueno en el otro caso). Y siempre puede hacer browsing usando los links.
- Una propuesta interesante es WebGlimpse, que intercepta la navegación para agregar una caja de búsqueda al final de cada página entregada. Esta caja permite buscar en la vecindad de la página actual.
- Otra alternativa útil son los *metabuscadors*, que dan una única interfaz y consultan a varios buscadores. Su principal argumento a favor es la poca cobertura e intersección entre buscadores específicos (la intersección entre los 4 mayores es sólo 1%). Sus principales problemas son
  - Ranking inconsistente entre los resultados obtenidos.
  - Distintos lenguajes de consulta en cada buscador.
  - No son queridos por los buscadores (pues viven de la publicidad), quienes tratan de hacerles la vida difícil.

- Un problema para el usuario es que la misma query tiene distinta semántica según el buscador, y además los usuarios no suelen entender esta semántica (problemas con el modelo booleano, mala elección de palabras para la consulta, etc.).
- A la hora de seleccionar un buscador, los usuarios consideran facilidad de uso, velocidad, cobertura, calidad de la respuesta y hábito.
- Algunas estadísticas sobre cómo consultan los usuarios Web
  - El 80% de las consultas no usa ningún tipo de operador.
  - El 25% de los usuarios usa normalmente *una sola palabra* en su consulta, y en promedio usan 2-3. Esto afecta la calidad el ranking pero simplifica el algoritmo de consulta.
  - Sólo el 15% restringe la búsqueda a un tema determinado.
  - Un 80% no modifica la consulta (retroalimentación cero).
  - Un 85% sólo mira la lista de los primeros calces.
  - Un 65% de las consultas son únicas (recordar caching).
  - Muchas palabras de la consulta suelen aparecer en la misma frase, sería bueno buscar por proximidad.
- Diseñar buenas interfaces es un desafío interesante.
- Está creciendo la idea de que vale la pena enseñarle al usuario a buscar lo que quiere más que tratar de adivinarlo.