# Sistemas de recuperación total de textos

Full text retrieval systems

INDICE:

**INTRODUCCIÓN** .......... **3**

**ÍNDICES INVERTIDOS** .......... **4**

COMPRESIÓN DE NÚMEROS DE DOCUMENTO .......... 6
*Modelos Globales* .......... *6*
Códigos Unarios .......... 6
Código Gamma .......... 7
Códigos Delta .......... 8
Modelo global tipo Bernoulli .......... 9
Códigos de Golomb .......... 10
Forma vectorial de los códigos .......... 11
Modelo Global de Frecuencia Observada .......... 12
*Modelos Locales* .......... *12*
Modelo Local tipo Bernoulli .......... 12
Modelo Local de Frecuencia Observada .......... 12
Batching .......... 12
ALMACENAMIENTO DE LOS TÉRMINOS .......... 13
Términos de longitud fija .......... 13
Concatenación de términos .......... 14
Front Coding .......... 15
Hashing Perfecto y Mínimo .......... 16
CONSTRUCCIÓN DE ÍNDICES INVERTIDOS .......... 25
Inversión por transposición de matrices .......... 25
Inversión por sort .......... 26

**SIGNATURE-FILES** .......... **27**

CONSTRUCCIÓN DE SIGNATURE-FILES – BIT SLICES .......... 28

**OPTIMIZACIONES** .......... **30**

Case-Folding .......... 30
Stop-Words .......... 30
Stemming .......... 30

**RESOLUCIÓN DE CONSULTAS** .......... **31**

CONSULTAS BOOLEANAS .......... 31
WILDCARDS .......... 32
N-Gramas .......... 32
Léxico rotado .......... 33
CONSULTAS RANQUEDAS .......... 33
Coordinate Matching .......... 34
Producto Interno .......... 35
Producto Interno Mejorado .......... 36
Modelos de espacios vectoriales – método del coseno .......... 37
PHRASE QUERIES (CONSULTAS DE FRASE) .......... 39
Los índices nextword .......... 40
CONSULTAS POR PROXIMIDAD .......... 41

**REFERENCIAS** .......... **42**

# Introducción

En este apunte nos enfocaremos en las base de datos de texto, para las que no se cuenta con registros con campos de distinto tipo sino con una colección de documentos formados por texto y en algunos casos elementos multimedia, como imagines o archivos de video. En general se trabaja con un tamaño muy grande de documentos (medido en millones o miles de millones) y se desea poder efectuar consultas que devuelvan aquellos que estén relacionados con un tema en particular. Para que este tipo de consultas no tarden eternidades se tiene que generar un sistema que se preocupe por resolver las consultas en tiempos aceptables pero sin utilizar demasiado espacio físico, que ya escasea debido a lo que ocupan la gran cantidad de documentos con los que se cuenta. Además el tipo de consultas en el que no existe una clave identificatoria de los documentos ni se busca por un documento en particular requiere la generación de distintas estructuras para poder resolverlas

Estos sistemas son conocidos como full-text retrieval systems (sistemas de recuperación total de textos) y como ejemplos tenemos:

- Un sistema con artículos sobre medicina en general, donde se puede consultar todas las investigaciones y descubrimientos sobre un tema en particular
- Un motor de búsqueda de páginas de Internet, donde se puede obtener links de sitios sobre los temas consultados
- Una enciclopedia en CD-ROM donde se permite buscar información general de todo tipo
- Una librería digital con artículos científicos que permiten encontrar papers de distintos autores. Como ejemplo se cita CiteSeer.IST (http://citeseer.ist.psu.edu/cis) página donde se pueden encontrar los papers utilizados como referencia para crear este apunte (los interesados pueden buscar los papers de Alistair Moffat como comienzo)

En general en los sistemas de recuperación total de texto se distinguen dos unidades de información:

- Término: es la unidad mínima con la que se cuenta, en la gran mayoría de los casos son palabras. Las consultas están compuestas de términos y con ellos se delimita sobre qué tema se quiere obtener los resultados
- Documento: Esta formado por una cantidad de términos y en varios casos por otros datos. Es la unidad que se obtiene como respuesta a una consulta.

En el primer ejemplo, cada artículo de medicina sería un documento y cada palabra del artículo sería un término. Ante una consulta como el término "genoma" se devolverán todos los artículos relacionados con este tema.

A continuación veremos distintos métodos para lograr tiempos de respuesta de consulta aceptables y al mismo tiempo hacer un uso eficiente del espacio en disco requerido para índices u otras estructuras utilizadas.

# Índices Invertidos

Un índice invertido es un índice en el que se relaciona cada uno de los términos del sistema con los documentos en donde aparece. Cada entrada de este índice tiene al término, la frecuencia o numero de ocurrencias (en cuantos documentos aparece) y luego una lista ascendente de todos los números de documentos en los cuales se encuentra al menos una vez. El nombre de este tipo de índice surge justamente de la forma invertida en que se tiene los datos, por un lado se tiene una cantidad de documentos en los que cada uno tiene sus términos y en el índice invertido se tiene por cada término sus documentos. Por ejemplo una entrada del tipo:

Linebacker | 3 | 52 | 53 | 54

Indica que el término "Linebacker" aparece en 3 documentos que son el número 52, el 53 y el 54. No trataremos en este apunte la relación entre numero de documento y el documento mismo ya que depende mucho del sistema en sí, podría ser necesaria una tabla que relacionara el numero de documento con el path del archivo, o quizás todos los documentos siguen una nomenclatura del tipo documentoXXXX.pdf donde XXXX es el número de documento.

Supongamos que tenemos el siguiente texto

Sapp salió al mercado de agentes libres
en la primera semana del período,
de esta forma, Sapp se convirtió en una valiosa
adición a la defensiva de Raiders

Si los términos son las palabras del texto y definimos que un documento es una línea, tendremos 4 documentos en total. El índice invertido para el sistema sería de la forma:

| Palabra | Ocurrencias | Documentos. |
|---|---|---|
| Sapp | 2 | 1,3 |
| Salio | 1 | 1 |
| al | 1 | 1 |
| mercado | 1 | 1 |
| de | 3 | 1,3,4 |
| agentes | 1 | 1 |
| libres | 1 | 1 |
| en | 2 | 2,3 |
| la | 2 | 2,4 |
| primera | 1 | 2 |
| semana | 1 | 2 |
| del | 1 | 2 |
| período | 1 | 2 |
| esta | 1 | 3 |
| forma | 1 | 3 |
| se | 1 | 3 |
| convirtió | 1 | 3 |
| una | 1 | 3 |
| valiosa | 1 | 3 |
| adición | 1 | 4 |
| a | 1 | 4 |

| defensiva | 1 | 4 |
|---|---|---|
| Raiders | 1 | 4 |

En el caso de que un término apareciera más de una vez en un mismo documento no se debe repetir el numero de documento. El espacio ocupado por el índice no es nada despreciable. En sí los documentos tienen miles de términos con lo que se tendrá una gran cantidad de ellos en todo el sistema. No solo se debe guardar la cadena de caracteres (lo que da un promedio de aproximadamente 5 o 6 caracteres por término en castellano) sino que se debe guardar la frecuencia (2 bytes por ejemplo) más cada número de documento (dependiendo de la magnitud del sistema podría verse como 3 o 4 bytes). Se ve que el tamaño ocupado será muy grande; en general es del 50% del ocupado por todos los documentos juntos. Es necesario entonces que busquemos formas de reducir el espacio sin que los tiempos de respuesta de consulta aumenten mucho (por más que queramos, cualquier reducción en espacio llevará a un tiempo mayor de consulta). Analizaremos entonces como almacenar de otras formas los números de documento primero y los términos después

# Compresión de números de documento

Comenzaremos viendo como utilizar menos espacio para indicar en que documentos aparece un término. Sin ningún tipo de compresión, para un termino se debería guardar no solo los números de documentos en donde aparece sino también la frecuencia. Conociendo la cantidad de documentos total, se puede utilizar una cantidad n de bits para representar los números de documentos, donde n sea el primer entero mayor o igual al logaritmo en base dos del máximo numero de documento. Por ejemplo, habiendo 100 documentos se podría utilizar 7 bits para cada numero de documento.

Una entrada de un índice invertido para el termino "mariscal" podría ser :

Mariscal – Freq= 7 - 3 | 4 | 6 | 7 | 11 | 12 | 17.

En este caso el primer 7 indica que el termino aparece en 7 documentos y luego se indican los documentos en los que aparece, que son el 3, el 4, etc. Para todo termino se cumple que la lista de documentos en los que está se encuentra ordenada ascendentemente. podría entonces guardarse también en vez de el numero de cada documento, la distancia desde el numero de documento anterior. Es decir, que la posición i nueva de la entrada sería la posición i vieja menos la posición i-1 vieja. Siguiendo el ejemplo, la entrada para el termino "mariscal" seria 7 | 3 | 1 | 2 | 1 | 4 | 1 | 5. Con esta entrada se puede reconstruir fácilmente el numero del 4to documento por ejemplo ya que es 3 + 1 + 2 + 1 = 7. Sin embargo este cambio es bastante importante ya que en primera instancia logramos que los números sean menores y en segunda instancia que hayan repeticiones dentro de una misma entrada, ya que el numero de documento 11 no se podía repetir en el caso anterior pero la distancia 1 se repite 3 veces en el segundo ejemplo. Estas dos propiedades nos permitirán sacar provecho de utilizar la lista de distancias en vez que la lista de números de documentos

Supongamos que los términos son palabras y los documentos son textos. Para las palabras mas frecuentes, al aparecer en muchos textos, tendremos una gran cantidad de distancias, pero estas en general serán números pequeños. En cambio para las palabras poco frecuentes las distancias serán pocas y con valores altos. Lo que necesitamos para comprimir los punteros a documentos, o en realidad las distancias, es un método que asigne tiras de bits cortas para las distancias pequeñas y tiras de bits largas para las distancias largas. Dado que la frecuencia de las primeras es mucho mayor, el archivo final disminuirá notablemente su tamaño

A continuación analizaremos varios modelos que hacen lo anterior, divididos en 2 grupos que denominamos modelos globales y locales. Los modelos globales utilizan el mismo método de compresión para todas las entradas del archivo, en cambio en los modelos locales para cada entrada se utiliza un modelo distinto y en consecuencia cambiará la codificación de una misma distancia de acuerdo a en qué entrada se esté

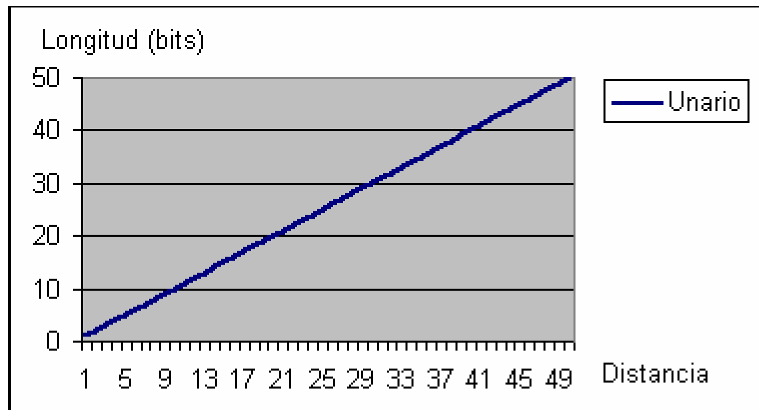
(**Aclaración**: Todos los logaritmos de este apunte están en base 2 a menos que se indique otra base)

## *Modelos Globales*

### Códigos Unarios

El modelo mas simple es el de los códigos unarios, en el que la distancia X se representa con X bits. La distancia 1 se codifica con un bit 0 y el resto de las distancias X se codifican con X-1 bits en 1 y un bit final en 0. Entonces la distancia 2 se codifica como 10, la distancia 5 como 11110, etc. Para leer un código en unario se busca desde la posición actual el primer bit que sea 0 y se obtiene la longitud del numero, que es el valor a leer.

Pese a que las distancias pequeñas se representan con muy pocos bits, para las grandes distancias la cantidad es excesiva, en un sistema con 100 documentos se necesitan 100 bits (12 bytes y medio) para representar la distancia de un termino que solo aparece en el ultimo, por lo que es necesario buscar alguna alternativa que funcione mejor con distancias grandes, que pese a que se dan con menor frecuencia igualmente influyen en el tamaño final del archivo. A continuación se grafica la longitud en bits con respecto a cada distancia; como se ve dicha función es lineal




## Código Gamma

En un código gamma una distancia x se representa por 2 códigos distintos:

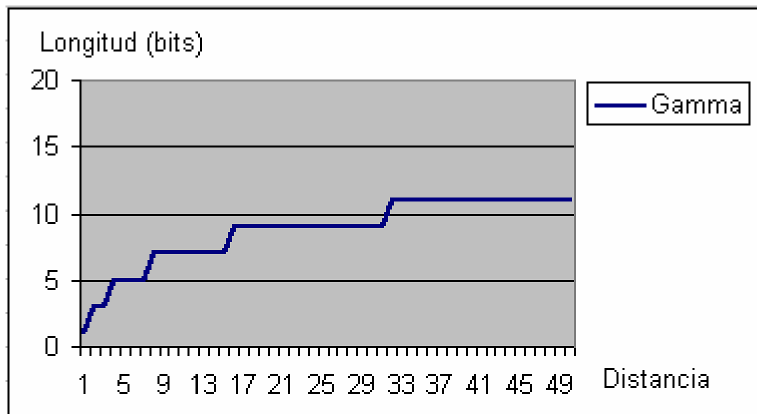
- Un código unario correspondiente a 1+ $\lfloor \log(x) \rfloor$
- Una representación binaria de X- $2^{\lfloor \log(x) \rfloor}$ que ocupa exactamente $\lfloor \log(x) \rfloor$ bits.

Por ejemplo, para la distancia 10:
$\lfloor \log(x) \rfloor$ = 3 => hay que representar 4 en unario (1110)
Seguido de 10-2^3 = 10-8 = 2 que es 010 en binario usando 3 bits
=> 10 en gamma es 1110 010, o sea un total de 7 bits contra los 10 que ocuparía en unario.

Para leer un código en gamma, primero se lee un código en unario, obteniendo su valor (llamémoslo v) para luego leer v-1 bits en binario, obteniendo otro valor (llamémoslo w). La distancia X leída es X = X=2^(v-1)+w

Ejemplos del código gamma: 1=0 2=100 3=101 4=11000 5=11001 6=11010 7=11011 8=1110000, etc. Como se ve el código gamma es peor que el unario para X=2 y X=4 pero en el resto de los valores la longitud del código gamma es menor y a medida que aumenta el valor la diferencia crece aun mas.

La distancia x se representa en gamma usando 1+2 $\lfloor \log(x) \rfloor$ bits. El grafico de longitud en bits para cada distancia es:




## Códigos Delta

Una tercera forma de codificación son los códigos delta, en los cuales la distancia se representa por dos códigos:

- Un código correspondiente a $1+\lfloor \log(x) \rfloor$ en código gamma
- Una representación binaria de X- $2^{\lfloor \log(x) \rfloor}$ que ocupa exactamente $\lfloor \log(x) \rfloor$ bits.

Por ejemplo, para la distancia 10:

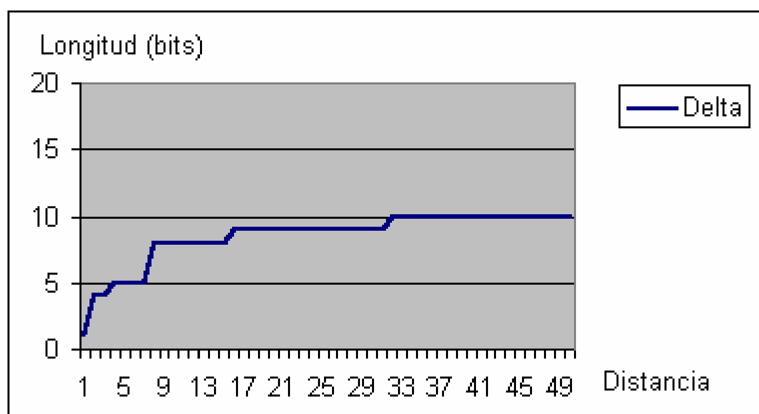
$\lfloor \log(x) \rfloor = 3$ => hay que representar 4 en gamma (11000)

Seguido de 10-2^3 = 10-8 = 10 en binario usando 3 bits (010)

=> 10 en delta es 11000 010, o sea un total de 8 bits

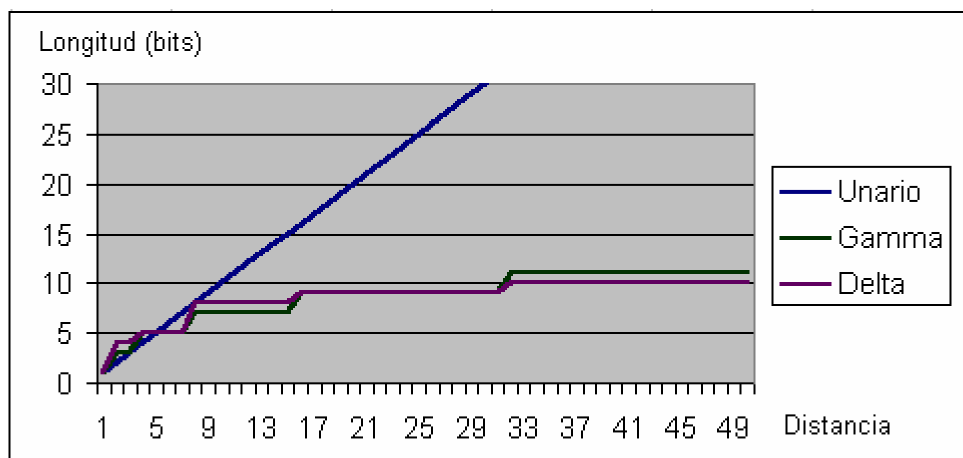
Como se ve son muy similares a los gamma salvo que se usa esta codificación en vez que el unario. Al hacer esto, en algunas distancias bajas el delta será peor que el gamma y el unario, pero en distancias grandes será mucho mejor que los dos. Para leer un código delta, primero se lee un código gamma (llamémoslo v) y luego se lee un código en binario de v – 1 bits que llamaremos w. La distancia X leída es X = X=2^(v-1)+w

La longitud de una distancia X es $1+2\lfloor \log[1+\lfloor \log(x) \rfloor] \rfloor + \lfloor \log(x) \rfloor$ que es lo mismo que $1+2\lfloor \log^2(2x) \rfloor + \lfloor \log(x) \rfloor$. Su grafico de longitud en bits entonces es:




Podemos comparar estos 3 métodos:

| Distancia | Unario | Gamma | Delta |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 10 | 100 | 1000 |
| 3 | 110 | 101 | 1001 |
| 4 | 1110 | 11000 | 10100 |
| 5 | 11110 | 11001 | 10101 |
| 6 | 111110 | 11010 | 10110 |
| 7 | 1111110 | 11011 | 10111 |
| 8 | 11111110 | 1110000 | 11000000 |
| 9 | 111111110 | 1110001 | 11000001 |
| 10 | 1111111110 | 1110010 | 11000010 |




Se ve que para las distancias menores a 15 el código gamma ocupa menos que el delta pero de ahí en adelante esto deja de ocurrir y desde la distancia 32 el código delta será siempre menor. Por ejemplo para X=1000000 el código gamma ocupa 39 bits y el delta solamente 28. También se puede apreciar la gran diferencia que hay entre ambos códigos y el unario para distancias grandes.

## Modelo global tipo Bernoulli

El modelo global de bernoulli incluye una mirada probabilística a la codificación de las distancias. Si conocemos la probabilidad de que un termino este en un documento, que llamaremos p, entonces la probabilidad de que no esté en el documento es 1-p. Basado en estos dos valores, podemos calcular la probabilidad de la distancia x pensada como la probabilidad de que el termino no se encuentre en los próximos x-1 documentos, pero sí en el próximo. De aquí sale la distribución probabilística de bernoulli P(x), que es

$P(x) = (1-p)^{(x-1)} * p$

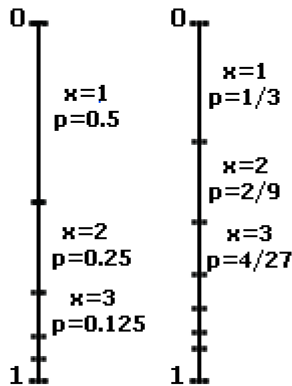
Lo que nos está faltando es la probabilidad p. Podemos calcularla como la cantidad de distancias que hay y la cantidad máxima que podría haber:

$p = k / N * t$

Donde k es la cantidad total de punteros que tiene el sistema, N la cantidad de documentos y t la cantidad de términos distintos. Es necesario notar que este valor p es genérico para toda distancia y todo termino (estamos trabajando con modelos globales). Por ejemplo, si todos los documentos

contienen a todos los términos, entonces la cantidad de distancias k será igual a N * t y entonces p=1.

Conociendo p entonces podemos utilizar la distribución probabilística de bernoulli para emitir un código de longitud optima para esa longitud. Por ejemplo se puede utilizar un compresor aritmético en el que el intervalo se divide para cada distancia. Como ejemplo se muestran los intervalos para p=1/2 y p=2/3. En ellos se puede ver para las distancias x=1, x=2 y x=3 la probabilidad P(x)




Es necesario aclarar que se debe guardar en algún sector del archivo (como en una cabecera con datos administrativos) el valor de p ya que no puede ser calculado al momento de leer el archivo porque las distancias están comprimidas utilizándolo

## Códigos de Golomb

Dada la complejidad de implementar un compresor aritmético para utilizar con el modelo de bernoulli, presentamos un método que emite códigos de longitud entera de bits, teniendo en cuenta la misma probabilidad p.

Los códigos de golomb utilizan un parámetro, que llamaremos g, y se representan mediante dos códigos:

- Un código correspondiente a q + 1 en unario, donde q = $\left\lfloor \frac{X-1}{g} \right\rfloor$
- Una representación binaria en prefijo de r= X – q * g – 1

Analizando lo anterior se verá que r = $g(\frac{X-1}{g} - \left\lfloor \frac{X-1}{g} \right\rfloor)$ . Si se opera con esto se verá que r es el resto de la división de X-1 con g. Por eso, r puede únicamente tomar valores de 0 a g-1. En el caso en que g sea una potencia de 2, se podrá utilizar binario de longitud log(g) bits. En cambio si g no es potencia de 2 se estarían desperdiciando algunos valores que nunca se darían por lo que se utiliza un código prefijo que beneficie a los valores mas bajos de r. Por ejemplo si g = 6 se puede utilizar para codificar r: 0=00 1=01 2=100 3=101 4=110 5=111

Ejemplos: Si tomamos g=3, para la distancia 10, q=3 y r=0. Utilizando la representación en prefijo 0=0 1=10 2 = 11, el código se representaría primero con 1110 (q+1 en unario) y luego 0, con lo que quedaría 11100. En la siguiente tabla se muestran ejemplos para g=3 y g=6

| Número | g=3 | g=6 |
|---|---|---|
| 1 | 00 | 000 |
| 2 | 010 | 001 |
| 3 | 011 | 0100 |
| 4 | 100 | 0101 |
| 5 | 1010 | 0110 |
| 6 | 1011 | 0111 |
| 7 | 1100 | 1000 |
| 8 | 11010 | 1001 |
| 9 | 11011 | 10100 |
| 10 | 11100 | 10101 |

Lo que resta saber de este método es que valor de g utilizar. Recordando el inicio de la explicación, veremos que el g estará relacionado con la probabilidad p utilizada en la distribución de bernoulli. No es la intención de este apunte entrar en demostraciones complicadas, por lo que diremos que se demuestra que el g elegido es optimo cuando se cumple que

(1-p)^g + (1-p)^(g+1) <= 1 < (1-p)^(g-1) + (1-p)^g

Despejando g:

g = log (2-p) / - log (1-p)

Se debe elegir un valor redondeado de g. Por ejemplo cuando p = 0.33, g = 1.28 con lo que se elige g = 1 (que puede demostrarse, es igual al unario). En cambio para p = 0.2 g = 2.63 con lo que se tomaría un g = 3

## Forma vectorial de los códigos

Una forma alternativa de expresar un código es utilizando un vector (x1, x2, x3, … xn), donde xi son números enteros. Lo que representa el vector es que los primeros x1 códigos pertenecen al primer grupo, luego los siguientes x2 códigos a un segundo grupo, etc. Además del vector se deben dar 2 formas de codificación: primero como se codificará el grupo y luego como se codificará el código para distinguirse de los otros códigos de su grupo (una especie de offset dentro del grupo).

Para representar un código X se busca primero a que grupo pertenece. El código pertenece al grupo i si se cumple que $\sum_{j=1}^{i-1} xj < X \leq \sum_{j=1}^{i} xj$. Con este dato, se representará primero i con la codificación del grupo. Luego se representara la posición del código dentro del grupo como $X - 1 - \sum_{j=1}^{i-1} xj$ con la codificación de distinción. Generalmente la segunda codificación es en binario si xi es una potencia de 2 o en código prefijo si no lo es.

Por ejemplo, los códigos gamma usan el vector (1, 2, 4, 8, 16, 32, …..), usan unario para definir a que grupo pertenece y binario para identificar el código dentro del grupo. Para codificar 17 en gamma primero se busca en que grupo está. El i buscado en este caso es 5, ya que x1+x2+x3+x4 < X < x1+x2+x3+x4+x5 o 15 < 17 < 31, por lo que se codifica 5 en unario (11110)

Luego en el grupo 5 hay 16 códigos distintos. Utilizando binario necesito 4 bits para representar a 17-16, o sea 1 (0001), Uniendo ambos códigos queda que 17 en gamma es 111100010

Como ejemplos de otros códigos, delta utiliza (1,2,4,8,16…) codificando con gamma y binario y golomb es (b, b, b, b,…) utilizando unario y códigos prefijó

Ahora supongamos que nos dan el siguiente vector: (2, 2, 4, 8, 8) y nos piden utilizando unario y binario representar la distancia 2 y la 7. Para la 2, se encuentra en el primer grupo por lo que es 1 en unario (0) y luego 1 en binario de 1 bit (1) con lo que el código final es 01. Para la distancia 7 se codifica el grupo 3 en unario (110) y 2 en binario de 2 bits (10) con lo que el código final es 11010

### Modelo Global de Frecuencia Observada

El ultimo modelo global que veremos es el de frecuencia observada. En este modelo se cuenta la frecuencia de cada distancia y estas se representan utilizando algún modelo estadístico como huffman. Este método es muy lento por tener que leer todas las entradas una vez para obtener la frecuencia de cada distancia y otra vez para escribir los códigos comprimidos. Además no es muy representativo ya que la probabilidad está tomada en cuenta para todos los términos, cuando en realidad en términos frecuentes serán muy probables las distancias chicas y en los no frecuentes las grandes.

## *Modelos Locales*

Como dijimos antes, en los modelos locales la codificación de las distancias es distinta en cada entrada. Esto será beneficioso ya que la distribución de distancias no es la misma para términos muy frecuentes que para términos que aparecen pocas veces, pero en general requieren de mas espacio en disco por necesitar guardar mas datos administrativos

### Modelo Local tipo Bernoulli

A diferencia del modelo global de bernoulli en que utilizábamos una probabilidad p para todas las distancias, en este caso calcularemos para cada entrada su verdadera probabilidad de ocurrencia, que se obtiene dividiendo la cantidad de documentos en que aparece dicho termino por la cantidad total de documentos. Luego se utiliza Golomb o un aritmético con esa probabilidad para cada entrada

### Modelo Local de Frecuencia Observada

En este modelo, para cada entrada se calcula la frecuencia de cada distancia y luego se emiten códigos utilizando un Huffman. El gran problema de este modelo es que para cada entrada hay que guardar la tabla de frecuencias de distancias para poder utilizar el huffman, y al final se termina desperdiciando demasiado espacio en disco

### Batching

Este modelo intenta disminuir el agregado de datos que utiliza el modelo de frecuencia observada. Lo que hace es para todos los términos con una misma frecuencia f utiliza un solo modelo. Entonces se deben guardar muchas menos tablas de frecuencias de distancias y se termina teniendo un ahorro mas notable. también se puede seguir ahorrando espacio utilizando un modelo para las distancias con frecuencia 1, otro para las distancias con frecuencia 2 y 3, otro para las distancias con frecuencia 4, 5, 6 y 7, etc. Igualmente mientras menos modelos se utilicen, la compresión por huffman será peor ya que no se estarán analizando las particularidades de los casos en que un termino tiene una frecuencia sino que se analizan varios de ellos de una vez

## Almacenamiento de los términos

Luego de decidir la forma de almacenar las distancias, ya estamos en condiciones de ver la estructura final del índice invertido. Lo mas importante es tener en cuenta que a éste se lo utilizará para hacer consultas y entonces es necesario poder determinar de una forma rápida los documentos de un termino, es decir acceder a una entrada del índice. El método mas efectivo de búsqueda que no tiene necesidad de agregar datos administrativos (recordemos que el espacio a utilizar debe ser lo menor posible) es el de la búsqueda binaria. Sin embargo para efectuar ésta de forma rápida es necesario que el tamaño de las entradas sea fijo (si fuera variable sería imposible determinar cual es el registro del medio). También se necesita que el índice este ordenado ascendentemente por término, en la sección de construcción del índice se verá como lograr esto

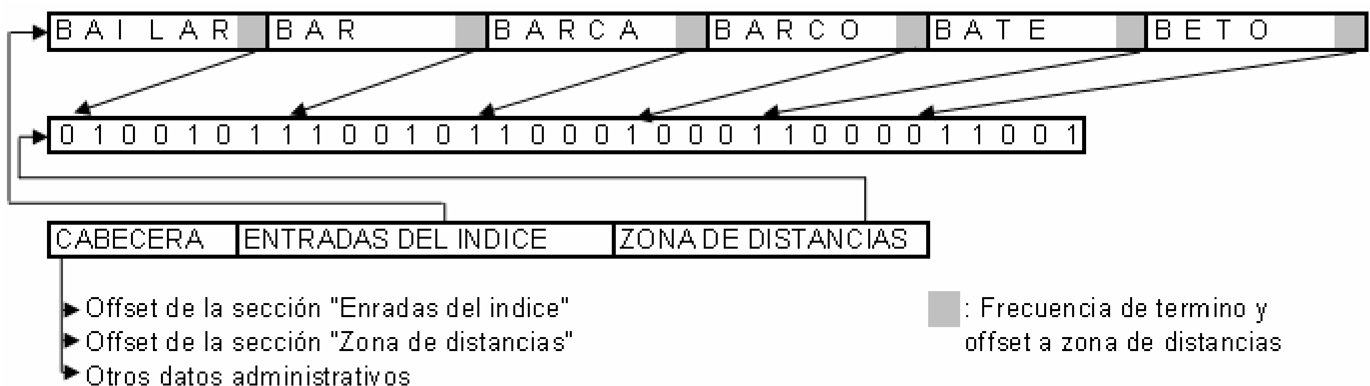
Para todo el capitulo trabajaremos con las siguientes entradas de un índice:

| Palabra | Ocurrencias | Documentos. |
|---|---|---|
| Bailar | 3 | 1,3,6 |
| Bar | 2 | 5,6 |
| Barca | 1 | 4 |
| Barco | 2 | 2,3 |
| Bate | 1 | 4 |
| Beto | 2 | 1,6 |

### Términos de longitud fija

De los 3 campos que tiene cada entrada del índice (término, frecuencia y distancias) solo la frecuencia es de tamaño fijo. Las distancias traen otro problema que es el hecho de que no siempre tienen una cantidad de bits múltiplo de 8. Lo que se suele hacer con ellas es concatenar todos los bits de las distancias de cada termino en un sector del archivo (que llamaremos zona de distancias) y guardar un offset al primer bit de la primer distancia en cada entrada. Pese a que este offset ocupa espacio de más, permite que no haya desperdicio de bits por no llenar una cantidad entera de bytes, y más importante aún, deja que el único campo con longitud variable que quede sea el del término

La primer opción que puede utilizarse es determinar el tamaño máximo de termino y grabarlos a todos utilizando esa cantidad de bytes, llenando con algún caracter especial (el \0 por ejemplo) el espacio que sobre. A continuación se muestra un gráfico del índice invertido, donde la zona gris luego de cada termino incluye la frecuencia y el offset a los bits. La codificación de distancias utilizada es la gamma y se ha pasado de numero de documento a distancias



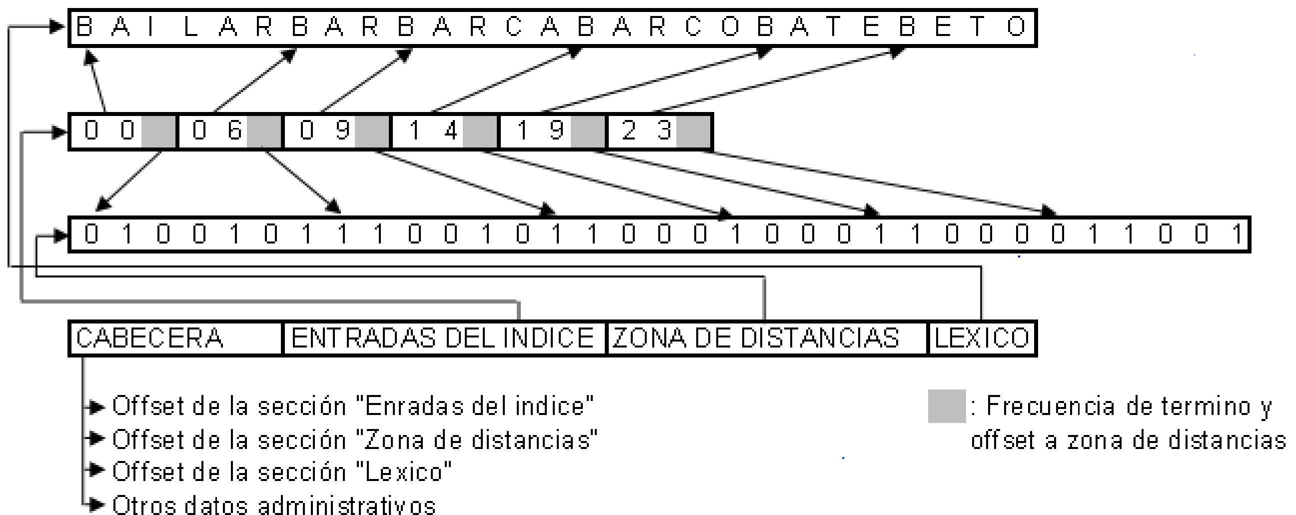
Como aclaración, pese a que aquí siempre trabajaremos con 1 solo archivo no por ello queremos decir que no se puedan usar un archivo para la zona de punteros y otro para las entradas del índice

El problema de esta solución es que desperdicia demasiado espacio en disco. Por ejemplo, en el término bar solo un 50% del espacio está siendo utilizado. De existir una palabra de 10 caracteres, se habrían desperdiciado 7 bytes. En una base con gran cantidad de términos, la suma de estos bytes se notará mucho y es por eso que es necesario utilizar otra técnica

## Concatenación de términos

Así como se hizo con los bits de las distancias, otra solución es concatenar los términos en otra zona del índice, que llamaremos "léxico". La entrada del índice entonces tendrá un offset al léxico para poder obtener el término, la frecuencia del mismo, y un offset a la zona de distancias para poder obtener todas las distancias del término

Al concatenar las distancias, como teníamos la frecuencia del término sabíamos cuantos bits leer, pero en cambio al concatenar los caracteres no hay nada que nos diga hasta donde parar, con lo que a primera vista pareciera que debe agregarse algún un campo indicando la longitud del término o un caracter especial para marcar la separación. Sin embargo esto no es necesario ya que la longitud del término puede calcularse como el offset del termino siguiente menos el offset del término del que se quiere saber la posición. La estructura del archivo queda entonces:

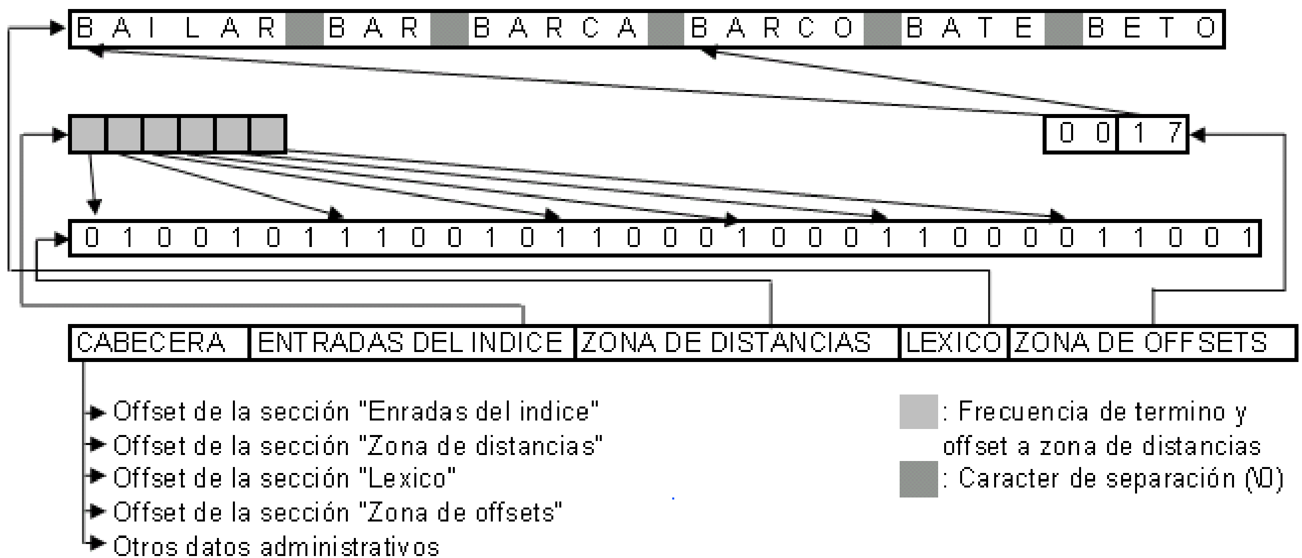



Dado que en general el promedio del tamaño de cada palabra es mucho menor que la longitud máxima, el ahorro por concatenar es mayor que lo que se pierde por agregar el offset, con lo cual este método termina logrando un ahorro considerable de espacio en disco. Sin embargo, las consultas serán un poco mas lentas porque se necesitará un acceso más a disco para leer el término de la entrada

Una modificación a este método que permite ahorrar más espacio es el utilizar una menor cantidad de offsets (que en general ocupan 4 bytes) y utilizar separadores de términos como el \0 que ocupan un solo byte. En esta técnica se forman grupos de N términos cada uno y el offset indica el inicio de cada grupo. Luego dentro del grupo se deberá buscar secuencialmente hasta encontrar el término. Para implementar esta técnica, en las entradas solo quedarán la frecuencia del término y el offset a los bits de las distancias, y se debe agregar una zona de offsets al léxico en el archivo

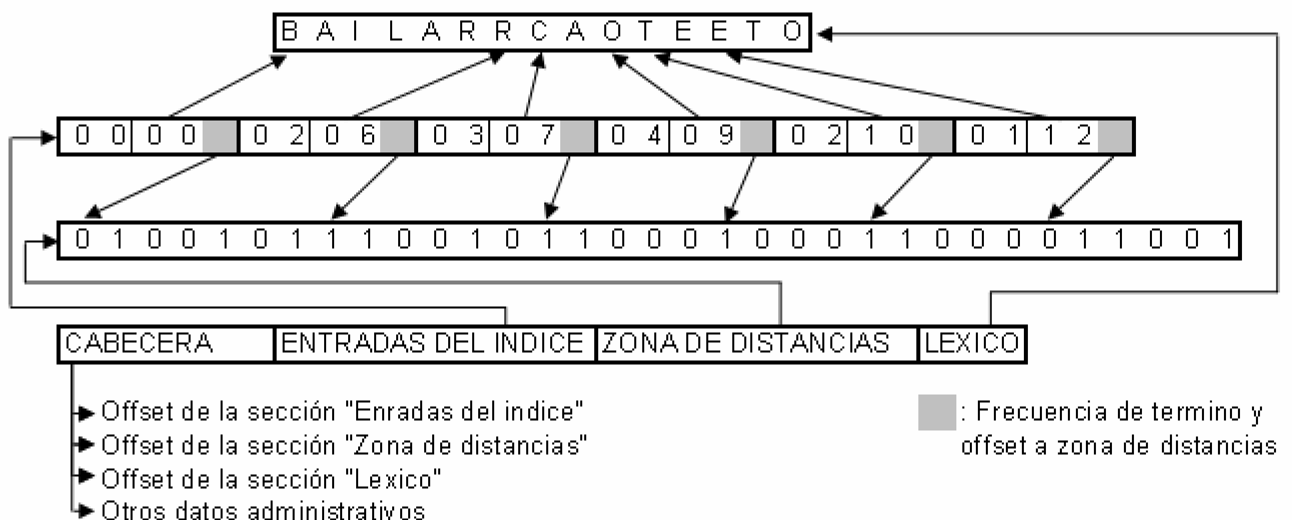
Por ejemplo para buscar el término X se debe primero hacer la división entera de $A = X / N$ y se accede a dicho grupo A. Luego se hace $B = X \bmod N$ y entonces el término numero B dentro del

grupo será el buscado. Como se ve, aquí también el ahorro en espacio implica un mayor tiempo de consulta. La estructura del archivo, tomando N = 3, es la siguiente:




## Front Coding

La técnica de front coding se aprovecha del ordenamiento de los términos para disminuir el tamaño del léxico considerablemente. Por estar ordenados, es muy frecuente que dos términos consecutivos tengan un inicio similar; en el ejemplo barco y barca comienzan con los mismos 4 caracteres y solo difieren en uno. Al hacer front coding, los primeros caracteres que sean iguales con el término anterior no se almacenan en el léxico, y para poder recuperar el término completo se agrega un campo más a la entrada del índice que indica la cantidad de caracteres iniciales que coinciden con el término anterior. El índice entonces queda de la siguiente forma (el primer campo numérico de la entrada es la cantidad de caracteres coincidentes y el segundo es el offset en el léxico):

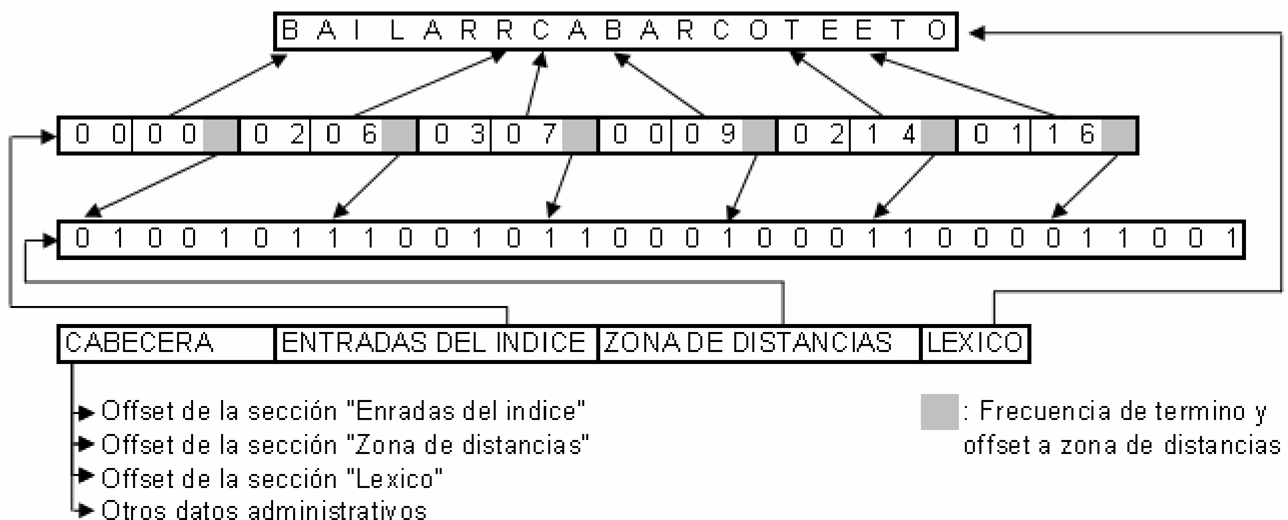


Veamos como ejemplo como se puede obtener el cuarto termino. Según la entrada del índice tiene 4 caracteres coincidentes con el anterior y el resto de los caracteres no coincidentes comienzan en el offset 09, o sea en el caracter O. Si se lee la quinta entrada (la del término que sigue) se ve que sus caracteres no coincidentes comienzan en el offset 10 con lo que la longitud de caracteres no coincidentes del cuarto termino es 10-9 = 1 , con lo que la cadena de caracteres no coincidentes completa es “O”. Para saber los primeros 4 caracteres del término necesitamos conocer al tercer término, que siguiendo la técnica recientemente utilizada sabemos que comienza con los mismos 3 caracteres que el segundo y luego le sigue la cadena “CA”. De esta cadena solo nos interesa la C ya que la A es el quinto caracter y no es compartido con el término que buscamos

Necesitaremos entonces acceder al segundo término del que no sabremos los primeros 2 caracteres pero sí que termina con “R” y finalmente al acceder al primero, como no tiene caracteres compartidos con el anterior, sabremos que es “BAILAR” Ahora con esto podemos componer el segundo término (“BAR”), luego el tercero “BARCA” y con los primeros 4 caracteres de este y la cadena “O” formaremos el cuarto término, que es “BARCO”

Como se ve, pese a que el tamaño del léxico se reduce notablemente, para poder acceder a un termino hay que efectuar una gran cantidad de accesos a los términos anteriores hasta llegar a uno que no tenga coincidencias con el término anterior y recién ahí poder construir el término buscado. Esto es demasiado costoso en tiempo y hace que se utilice una alternativa conocida como Front coding parcial

El front coding parcial se utiliza la misma técnica que en el front coding pero con la salvedad que cada N términos se guarda el término completo independientemente de cuantos caracteres coincidan con su término anterior. Gracias a esto, al intentar construir un término como mucho se deberán leer los N-1 términos anteriores. Jugando con el valor de N se puede llegar a un tamaño bastante reducido y una velocidad de consulta aceptable. A continuación se muestra el índice con front coding parcial cada 3 términos (N=3)




Como se ve, se puede acceder al cuarto término sin acceder al tercero, y para acceder al quinto solo se necesitará leer el cuarto. Como aclaración, en los ejemplos se utilizaron tantos offsets como términos había, pero también se puede utilizar una menor cantidad de offsets y separador de términos como se vio en concatenación de términos

## Hashing Perfecto y Mínimo

Una función de hashing es una función que convierte un string de una cierta longitud (o de longitud variable) en un número comprendido en un cierto rango. Una función de hashing puede

producir que dos o más strings originen como resultado el mismo número, se dice entonces que estos strings son sinónimos.

Función de hashing:
Dado cualquier string ‘s’ y un espacio de direcciones N.
h(s) =r tal que 0<=r<N.

Funciones de hashing perfectas.
Una función de hashing es perfecta si nunca produce sinónimos.
Función de hashing perfecta.
h(s1) = h(s2) <=> s1 = s2

Funciones de hashing mínimas.
Una función de hashing es mínima cuando la cantidad de strings posibles a hashear es igual al espacio de direcciones N.

Funciones de hashing preservadoras del orden.
Una función de hashing preserva el orden si:
h(s1) < h(s2) <=> s1 < s2

Para la construcción de nuestro índice utilizaremos una función de hashing perfecta, mínima y conservadora del orden. Al buscar uno de los términos, se le aplicara dicha función que nos devolverá un número que corresponderá con la entrada en el archivo invertido para dicho término. Entonces:

- Los strings a hashear son los términos del índice.
- El espacio de direcciones es igual a la cantidad de strings.

Y la función debe ser:

- Perfecta porque si no lo fuera dos términos tendrían la misma entrada en el índice.
- Mínima para que no haya entradas en el índice que no corresponden a ningún string desperdiciando espacio.
- Preservadora del orden para poder manejar el índice en forma eficiente.

Sabiendo que el espacio de funciones es infinito, no es de sorprender que exista al menos una función de hashing perfecta, mínima y conservadora del orden para un set de términos dados. El problema esta en como encontrarla. Durante mucho tiempo la construcción de una función de este tipo fue considerada como una tarea extremadamente difícil. A continuación presentamos una técnica para construir funciones de hashing perfectas, preservadoras del orden y mínimas para cualquier conjunto de strings, siendo los strings conocidos previo a la construcción y la cantidad de ellos fija ya que la función debe reconstruirse si se agregan strings

Construcción.

- Dadas una cantidad de strings N (que llamaremos string1 a stringN) y el espacio de direcciones N se deben construir dos funciones de hashing: h1 y h2 comunes (no tienen ningún requisito) cuyo espacio de direcciones sea M con M>N.
- Por un método que describimos luego se genera una función G.

• La función de hashing perfecta para un string s se obtiene de:
h(s) = g(h1(s)) + g(h2(s))  La suma se hace modulo N.

Como el espacio de direcciones de h1 y h2 es M, solamente se necesitan guardar en el archivo los valores de g(0) hasta g(M-1). Además, como la suma se hace en modulo N, cada valor g(i) se guarda en modulo N, ocupando solamente $\lceil \log(N) \rceil$ bits.

Para la construcción de G se parte de N ecuaciones que deben cumplirse para que h sea perfecta, mínima y conservadora del orden, y que son:

0 = g(h1(string1)) + g(h2(string1))
1 = g(h1(string2)) + g(h2(string2))
2 = g(h1(string3)) + g(h2(string3))
……..
N-1 = g(h1(stringn)) + g(h2(stringn))

Fijando h1 y h2, obtenemos los 2 hash de cada string y llegamos a N ecuaciones con M incógnitas que son g(0), g(1) ….. g(M-1). De acuerdo a los valores de las funciones de hashing elegidas, si no hay ninguna contradicción entre las ecuaciones (ejemplo: 1 = g(0) + g(1) y 2 = g(0) + g(1)) se podrán encontrar valores de g(0) a g(M-1) que cumplan las restricciones.

Sin embargo todavía hay un pequeño problema, ya que el hecho de que haya solución no implica que los valores de g encontrados sean enteros, con lo que no se podrá utilizar $\lceil \log(N) \rceil$ bits para representarlos. Por suerte para nosotros, existe un método computacionalmente sencillo para verificar que dadas dos funciones de hashing y N strings distintos se podrá construir con el mecanismo anterior una función de hashing perfecta, mínima y conservadora del orden y además con todos los valores de g enteros en álgebra de modulo N. El método es el siguiente:

1. Se forma un grafo de M vértices, donde cada vértice representa uno de los posibles valores devueltos por h1 o h2. Cada vértice se numera con un número entre 0 y M-1. De aquí en mas a dichos vértices los llamaremos v0, v1, v2, … , v(m-1)
2. Se agregan N aristas al grafo donde cada arista representa una de las ecuaciones que deben cumplirse. Para cada ecuación i-1 = g(h1(stringi)) + g(h2(stringi)) se agrega una arista que vaya del vértice numero h1(stringi) al vértice numero h2(stringi). Cada arista se rotula con el valor de la suma de los dos g, o sea con el valor i-1.
3. Se verifica que el grafico no sea cíclico. Si es cíclico entonces o bien no habrá una solución o bien hay grandes posibilidades de que la solución de valores no enteros de g(x), por lo que se eligen otras funciones h1 y h2 distintas y se vuelve al primer paso.
4. Finalmente una vez que se tiene el grafo acíclico se comienzan a fijar los valores de g(x). A cada vértice del grafo se le asociara un numero de tal forma de que si al vértice i se le asocia el numero K, entonces g(i) = K. Una vez fijados valores para todos los vértices se tendrán todos los valores de g necesarios. Para asociar los valores a los vértices se procede de la siguiente forma:
   4.1. Se elige un vértice vi al que no se le haya fijado un valor todavía. Se le asocia un valor cualquiera entre 0 y N-1. Al asociar este valor al vértice vi, el valor g(i) queda fijado
   4.2. Mientras queden aristas que conecten un vértice vj con valor asociado y un vértice vk sin valor asociado, se fija el valor asociado a vk como Rotulo de la arista – g(j), fijando entonces el valor de g(k). Es importante saber que la resta anterior también debe ser hecha trabajando con álgebra de modulo N.
   4.3. Finalmente, si al llegar a este paso quedo algún vértice al que no se le haya asociado un valor, se vuelve al paso 4.1

Una vez finalizado el método todos los vértices han quedado con un valor asociado, y la función g se puede obtener como g(i) = Valor asociado al vértice i.

Analicemos el método:

- Cada vértice esta asociado con un valor de g. g(i) será el valor asociado al vértice vi.
- Cada arista representa a una de las ecuaciones que deben cumplirse para que h sea perfecta, mínima y conservadora del orden. Relaciona los vértices que equivalen a los dos valores de g de la ecuación y se asocia a un numero que corresponde a la suma de dichos valores
- En el paso 4.1 se fija un valor de g(i). Al hacer esto, algunas de las ecuaciones que relacionan valores de g pueden haber quedado con una sola incógnita, con lo cual se pueden despejar sus valores. Eso es lo que se hace en el punto 4.2. A su vez al mismo tiempo al despejar un nuevo valor de g aún más ecuaciones pueden haber quedado con una sola incógnita, y se debe repetir hasta que únicamente queden ecuaciones con 2 incógnitas. Si se repiten los pasos 4.1 y 4.2 llegará un momento en que no haya vértices sin valor asociado y que todas las aristas relacionen vértices tal que la suma de sus valores asociados en álgebra de modulo N sea el rotulo de la arista. Cuando ocurra esto, tenemos valores de g tal que cumplen con todas las ecuaciones y por ello tenemos una función h perfecta mínima y conservadora del orden
- En el paso 3 hay que notar que podría tenerse un ciclo en el grafo pero igual obtener una solución con valores enteros. Como ejemplo, si se tiene 3 = g(a) + g(b); 4 = g(a) + g(c) y 5 = g(b) + g(c), se tendrá un ciclo en el grafo pero existe una solución que es g(a) = 1, g(b) = 2, g(c) = 3. Sin embargo, para grandes cantidades de términos es mucho más simple volver a empezar todo de nuevo que perder una gran cantidad de tiempo analizando los ciclos a ver si se producirán valores no enteros de g(x). Así que siempre que se encuentre un ciclo se tomaran nuevas funciones h1 y h2 y se comenzará desde el principio hasta encontrar un grafo acíclico
- Finalmente se ve que con este método no solo se deben grabar los M valores g(x) sino que también se deberá identificar en el archivo que funciones de hashing h1 y h2 se terminaron utilizando. Una forma simple es utilizar funciones h1 y h2 que dependan de una cierta cantidad de valores random; si el grafo queda cíclico se eligen otros valores random hasta que quede acíclico. En este caso, lo que se guarda en el archivo final es la tira de valores random que finalmente se utilizo

Veamos ahora como funciona el hashing perfecto con un ejemplo. Se utilizaran los siguientes términos: Arnaldo, Bartolo, Celsa, Edgar, Emilce, Hilda, Humberto, Olga, Rene, Sandro y Victor. Las funciones de hashing h1 y h2 tendrán la forma

$$\sum_{i=0}^{strlen(s)-1} v[(i\% sizeof(v))] * s[i]$$

O sea, la sumatoria de cada carácter multiplicado por una posición distinta de un vector v de valores random (obviamente cada función con un distinto vector random o devolverían lo mismo). También se le debe aplicar a la salida de cada función un MOD 16, numero > 11 (cantidad de términos) que se eligió como espacio de direcciones de h1 y h2. Los vectores generados al azar de 4 caracteres de tamaño fueron:

- Vector Random 1 : 47 - 60 - 187 - 120
- Vector Random 2 : 0 - 218 - 231 - 34

Con lo que se llego a:

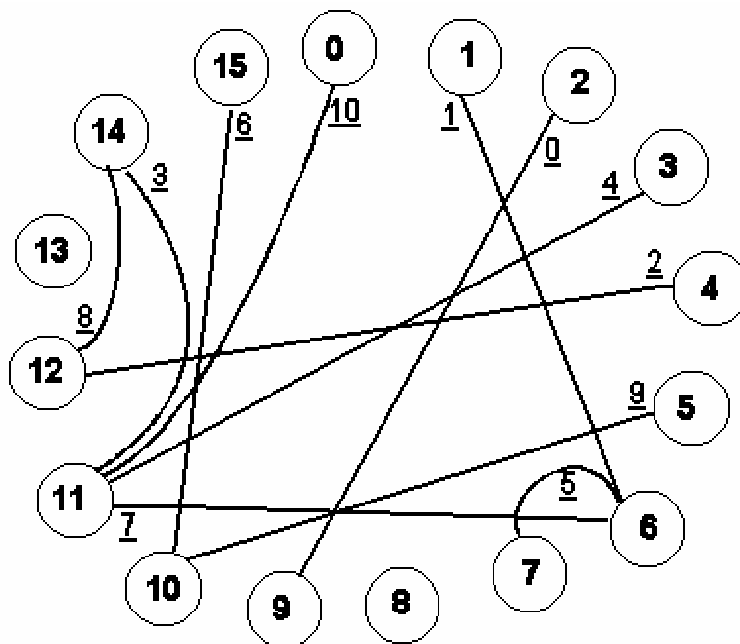
| Términos | h1(s) | h2(s) |
|---|---|---|
| Arnaldo | 2 | 9 |
| Bartolo | 6 | 1 |
| Celsa | 4 | 12 |
| Edgar | 14 | 11 |
| Emilce | 3 | 11 |
| Hilda | 7 | 6 |
| Humberto | 10 | 15 |
| Olga | 6 | 11 |
| Rene | 12 | 14 |
| Sandro | 5 | 10 |
| Victor | 0 | 11 |

Paso 1: Construimos el grafo, con 16 vértices. El numero 16 no fue elegido por alguna razón en particular, cualquier numero mayor a 11 habría servido, pero mientras mayor sea el numero es mas fácil que el grafo quede acíclico.

Paso 2: Para cada string los valores de h1 y h2 determinan una arista del grafo. En nuestro caso las aristas son:

2-9;6-1;4-12;14-11;3-11;7-6;10-15;6-11;12-14;5-10;0-11

Las aristas se rotulan con el número de string que la genero, para el string “Arnaldo” las funciones de hashing daban valores 2 y 9, por eso la arista 2-9 tiene el rotulo 0 (cero). El grafo generado luego de los dos primeros pasos es: (el texto subrayado son los rótulos de las aristas)



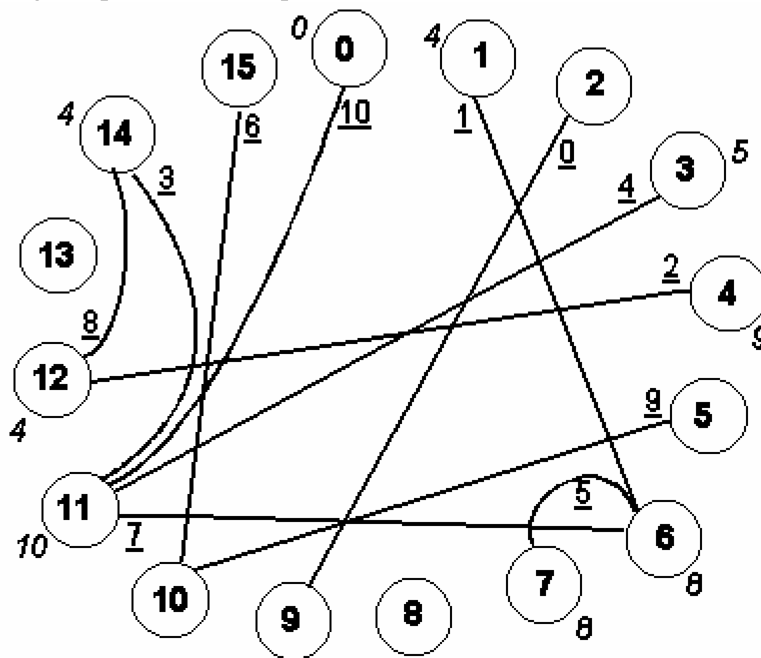

Paso 3: Se comprueba que el grafico es acíclico. Si hubiera quedado cíclico se deberían elegir nuevas funciones h1 y h2, en nuestro caso cambiando los valores de los vectores random

Paso 4: Se elige un vértice, en este caso elegiremos el 0 pero se puede empezar por cualquiera. Se le asocia un valor cualquiera entre 0 y 10, por ejemplo, 0. Entonces g(0) = 0.

Ahora la arista rotulada con 10 conecta un vértice con valor asociado y un vértice sin valor asociado. Entonces se asocia al vértice 11 un valor de 10-0 = 10. Ahora nos quedan varias aristas conectando vértices con valor asociado y vértices sin valor asociado que son las rotuladas con 3, 4 y 7. Asociamos entonces al vértice 14 el valor de 3 – 10= -7 que en álgebra mod 11 es 4; al vértice 3 el valor de 4-10 = -6 = 5 en álgebra mod 11; y al vértice 6, 7-10 = -3 = 8 en álgebra mod 11. Continuando con el método, al vértice 7 se le asigna el valor 5-8=-3=8 en álgebra mod 11, al vértice 1 se le asigna el valor 1-8=-7=4 en álgebra mod 11 y al vértice 12 el valor 8-4 = 4; luego al vértice 4 el valor 2-4 = -2 = 9 en álgebra mod N.

En este momento ya no quedan aristas conecta un vértice con valor asociado y un vértice sin valor asociado, pero si quedan vértices sin valor asociado (por ejemplo el 2) con lo que se deberá volver al paso 4.1. Quedaron definidos entonces g(11) = 10, g(14) = 4, g(3) = 5, g(6) = 8, g(7) = 8, g(1) = 4, g(12) = 4, g(4) = 9

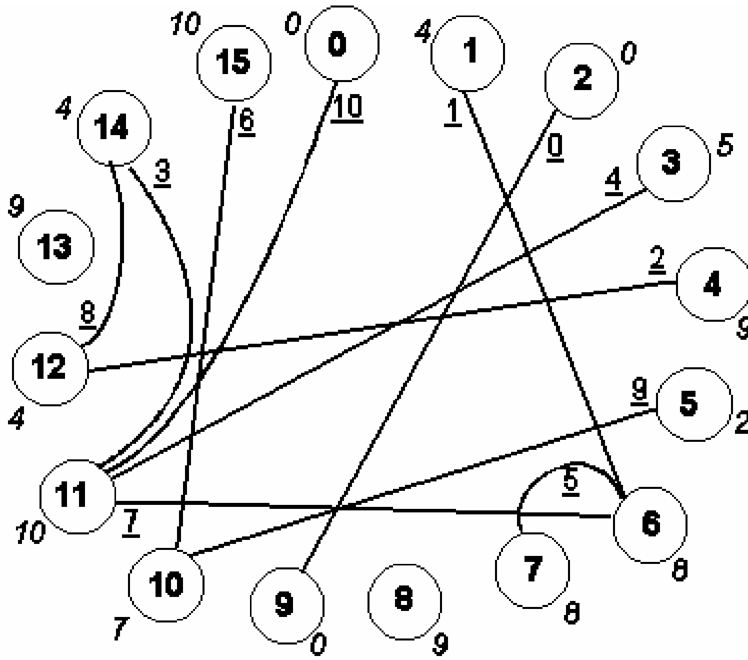
El grafo por ahora esta quedando de la forma:




Elegimos un valor asociado cualquiera para el vértice 2, por ejemplo 0. Entonces por la arista rotulada 0, el valor del vértice 9 será 0-0 = 0. Se definió entonces g(2) = 0 y g(9) = 0. Se vuelve al paso 4.1 porque hay vértices sin valor asociado

Elegimos un valor asociado cualquiera para el vértice 5, para variar un poco elegiremos el valor 2. El valor asociado al vértice 10 será entonces 9-2 = 7. Luego el valor del vértice 15 será de 6-7 = -1 = 10 en álgebra mod 11. Volvemos al paso 4.1 debido a los vértices 8 y 13, que los rotulamos a ambos con un valor cualquiera ya que todas las aristas ya conectan vértices con valor asociado. Para ambos elegimos el valor asociado 9. Quedaron definidos entonces g(5) = 2, g(10) = 7. g(15) = 10, g(8) = 9, g(13) = 9.

Hemos terminado con el método. El grafo final es el siguiente:




La función g ha quedado de la siguiente forma:

| x | g(x) |
|---|---|
| 0 | 0 |
| 1 | 4 |
| 2 | 0 |
| 3 | 5 |
| 4 | 9 |
| 5 | 2 |
| 6 | 8 |
| 7 | 8 |
| 8 | 9 |
| 9 | 0 |
| 10 | 7 |
| 11 | 10 |
| 12 | 4 |
| 13 | 9 |
| 14 | 4 |
| 15 | 10 |

Una vez descripta la función g es posible calcular la función de hashing h.

| string | h1(s) | h2(s) | g(h1(s)) | g(h2(s)) | h(s) |
|---|---|---|---|---|---|
| Arnaldo | 2 | 9 | 0 | 0 | 0 |
| Bartolo | 6 | 1 | 8 | 4 | 1 |
| Celsa | 4 | 12 | 9 | 4 | 2 |
| Edgar | 14 | 11 | 4 | 10 | 3 |
| Emilce | 3 | 11 | 5 | 10 | 4 |
| Hilda | 7 | 6 | 8 | 8 | 5 |
| Humberto | 10 | 15 | 7 | 10 | 6 |

| Olga | 6 | 11 | 8 | 10 | 7 |
|---|---|---|---|---|---|
| Rene | 12 | 14 | 4 | 4 | 8 |
| Sandro | 5 | 10 | 2 | 7 | 9 |
| Victor | 0 | 11 | 0 | 10 | 10 |

Como puede observarse los valores de h son únicos con lo cual la función de hashing h(s) es perfecta , mínima y además mantiene el orden de los strings.

Una vez vista la forma en la cual se construye la función de hashing es conveniente analizar cuanto tiempo insume la construcción de la misma. El problema surge de la necesidad de probar grafos hasta encontrar uno acíclico que asegura la existencia de la función g. La cantidad de grafos a probar disminuye si el número M se hace mas grande.

La cantidad de grafos a testear (promedio) utilizando M = C * N puede calcularse como:

$$e^{\sum_{k=1}^{m} \frac{2^k}{2kc^k}}$$

Para un c pequeño, esta cantidad es excesivamente grande. En cambio para c > 2 se da que

$$\lim_{m->\infty} \sum_{k=1}^{m} \frac{2^k}{2kc^k} = \frac{1}{2} \ln \frac{c}{c-2}$$

Con lo que si tenemos un numero muy grande de términos (o sea un n muy grande) la cantidad de grafos promedios a testear quedaría expresada como

$$\sqrt{\frac{c}{c-2}} = \sqrt{\frac{M}{M-2N}}$$

Con M=3N por ejemplo la cantidad de grafos a testear en promedio es 1,7 lo cual es muy manejable. Sin embargo usar un valor de M muy grande como por ejemplo 3N tiene como desventaja que se ocupa mucho espacio en disco (la función g hay que guardarla en disco) con lo cual deja de tener sentido usar una función de hashing en lugar de los strings.

La solución pasa por aumentar el número de funciones de hashing que se aplican a los strings, usando 3 funciones de hashing en lugar de dos. De esta forma en lugar de un grafo binario tenemos un grafo ternario. Cada arista conecta ahora tres vértices y el requisito para que exista la función g es que no haya subgrafos que contienen únicamente vértices de grado dos o superior. El algoritmo a aplicar si el grafo reúne las condiciones es idéntico al que aplicábamos con el grafo binario.

Con un grafo ternario puede obtenerse una función de hashing perfecta probando muy pocos grafos con valores de M cercanos a 1.23*N. El algoritmo es muy eficiente y para la base más grande de nuestros ejemplos: TREC insume alrededor de un minuto en construir la función de hashing perfecta.

Si se utiliza una función de hashing perfecta hay que guardar por un lado la función G (insume 4 bytes * M) es decir alrededor de 5 bytes por string y en otra parte se guardan las frecuencias y punteros.

Para buscar un término en el índice usando hashing se debe primero saber que funciones de hashing se utilizaron como h1 y h2 (en el ejemplo nuestro consistiría en leer del archivo los valores random de ambos vectores, guardados en algún sector administrativo como la cabecera por ejemplo). Conociendo las funciones se hashea el string y se obtienen h1(s) y h2(s). Luego accediendo a la función g (que esta guardada) se obtiene el valor de la función de hashing h, que indica la entrada en el archivo invertido que corresponde al termino. Luego se accede al índice de frecuencias y punteros en la posición indicada por la función de hashing.

El gran problema esta en si buscamos un termino que no esta en la lista inicial, por ejemplo Pierre. Hasheandolo obtendríamos que h1(Pierre) = 13 y h2(Pierre) = 3. g(13) + g(3) = 9 + 5 = 14 = 3 en álgebra mod 11. Entonces accederíamos a la tercer entrada del archivo invertido y estaríamos listando todos los documentos que contienen al termino "Edgar". Cualquier usuario se vera frustrado al ver que el sistema de consultas le devolvió 20 documentos conteniendo a "Pierre" pero en ninguno de ellos aparece 1 sola vez (y encima, no paran de nombrar a un tal Edgar Agar). La única forma de lograr que esto no ocurra es que antes de devolver los documentos en los que se encuentra el término buscado, se recorra un documento (el más corto) buscando alguna ocurrencia del término buscado. Si no se encuentra ninguna, es que se accedió a una entrada de otro término y se lo informa. En cambio si se encuentra una ocurrencia ya se sabe que el término es el buscado y se deja de recorrer el documento, mostrando todos los documentos que originalmente se habían encontrado en la entrada del archivo invertido. Esto hace bastante más lentas las consultas pero se gana mucho en espacio en disco

Una alternativa sin embargo se da en el caso en que el espacio no es critico, por ejemplo en una enciclopedia en CD-ROM en la que luego de hacer concatenación de strings sobran algunos megas para llegar al limite de la capacidad. Si ahí se aplica hashing perfecto como complemento y no como alternativa a la concatenación de strings lo que se logrará es pasar de una búsqueda binaria a un acceso directo a la posición del término, y como este se encuentra en el archivo, un acceso más para comprobar si es el buscado o no. Esta alternativa ocupa mas espacio pero da una mejor performance

# Construcción de Índices Invertidos

Hasta ahora hemos visto distintas técnicas para almacenar el índice invertido, sin embargo no hemos visto como llegar partiendo desde los documentos a tener todas las entradas termino – frecuencia – documentos para crear el índice. Dado que se está manejando una gran cantidad de información, la construcción no es trivial ya que si es llevada de mala forma tardará un tiempo excesivo.

## Inversión por transposición de matrices

Este es el método mas sencillo de construcción. Suponiendo que se tiene el ejemplo

Donovan Mc Nabb
Steve Mc Nair
Mc Nabb
Donovan Mc Nair
Donovan.

Suponiendo que cada término es una palabra y que cada línea es un documento el proceso de inversión consta de dos fases. En la primera se construye una matriz a medida que se leen los documentos de la siguiente forma

| | Donovan | Mc | Nabb | Steve | Nair |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 1 |
| 3 | 0 | 1 | 1 | 0 | 0 |
| 4 | 1 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 | 0 |

El segundo paso es obtener la transpuesta de la matriz anterior:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Donovan | 1 | 0 | 0 | 1 | 1 |
| Mc | 1 | 1 | 1 | 1 | 0 |
| Nabb | 1 | 0 | 1 | 0 | 0 |
| Steve | 0 | 1 | 0 | 0 | 0 |
| Nair | 0 | 1 | 0 | 1 | 0 |

Luego se debe ordenar las filas de la matriz y recorriéndola por filas se forma el índice invertido. Este método es excelente si se puede albergar la matriz completa en memoria, pero justamente esto no se podrá hacer en una gran cantidad de los casos, con lo que se deberán hacer varios accesos a discos y teniendo en cuenta el excesivo tamaño que ocupa, los tiempos de generación del índice serán muy grandes

La segunda opción a estudiar es utilizar listas enlazadas en vez de matices. El gran problema del tamaño de la matriz es que un termino que aparece pocas veces igual necesita tantos valores como documentos hayan. Utilizando listas enlazadas, solo se ingresarían elementos cuando el termino aparece el documento. Sin embargo estos punteros ocupan bastante espacio en memoria y

si bien el método tarda menos que utilizar una matriz estática, el tiempo de generación sigue siendo inaceptable

## Inversión por sort

El método mas utilizado es el de inversión por sort de dos pasadas. En este método se generaran 2 archivos, uno con el léxico (todos los términos distintos) y otro con los pares numero de termino – numero de documento. Luego ordenando ambos archivos con algún método de sort externo se podrá formar el índice invertido. Veremos a continuación las 3 fases del método en detalle

La primera es la extracción del léxico y construcción del archivo auxiliar. Leyendo documento por documento secuencialmente se construyen 2 archivos. El primero será un archivo con el léxico, identificando a cada término con un numero de termino. Cada vez que se lea un término nuevo se agregará a este archivo y se le dará un numero nuevo. El segundo archivo, que llamaremos archivo auxiliar, tendrá un registro con 2 campos, el primero el número de término y el segundo el número de documento. Por cada término que se lea (sea nuevo o no) se agregará un registro a este archivo. Con el ejemplo anterior, los archivos que tendremos al finalizar la fase serán:

Léxico: 1)Donovan 2)Mc 3)Nabb 4)Steve 5)Nair
Auxiliar: (1,1) (2,1) (3,1) (4,2) (2,2) (5,2) (2,3) (3,3) (1,4) (2,4) (5,4) (1,5)

La segunda fase es el sort de ambos archivos. Primero se debe ordenar el léxico para que el índice quede finalmente ordenado por término. Esto hará que los números de término queden también ordenados:
Léxico ordenado: 1)Donovan 2)Mc 3)Nabb 5)Nair 4)Steve
Luego se debe ordenar el archivo auxiliar por numero de término y numero de documento (en ese orden) pero utilizando el orden dado por el léxico ordenado. El archivo queda:
Auxiliar Ordenado: (1,1) (1,4) (1,5) (2,1) (2,2) (2,3) (2,4) (3,1) (3,3) (5,2) (5,4) (4,2)

En la última fase se procede a la construcción del índice invertido, recorriendo secuencialmente el archivo auxiliar ordenado. Cada vez que cambie el número de término se accede a la siguiente entrada del léxico ordenado para obtener el string del término. En este paso se almacenarán las distancias y los términos de acuerdo al método elegido de los que se han visto anteriormente. Para calcular la frecuencia, se van acumulando la cantidad de registros para el mismo numero de término y al cambiar se graba el valor acumulado y se setea en 0 nuevamente. Recorriendo los archivos auxiliares del ejemplo se llega a la siguiente información:
Donovan: Aparece en los documentos 1, 4 y 5 (frecuencia 3)
Mc: Aparece en los documentos 1, 2, 3 y 4 (frecuencia 4)
Nabb: Aparece en los documentos 1 y 3 (frecuencia 2)
Nair: Aparece en los documentos 2 y 4 (frecuencia 2)
Steve: Aparece en el documento 2 (frecuencia 1)

# Signature-Files

Otra forma de implementar un índice para un archivo de texto es utilizando signature-files. Un signature-file es un archivo que contiene una entrada por documento, en cada una de estas entradas se almacena un signature (firma) correspondiente al documento, que surge de los términos que lo integran. El tamaño del signature es un parámetro a definir, con lo que con este método se puede delimitar el tamaño a utilizar en disco

Para construir el signature de un documento se debe contar con una cantidad M de funciones de hash que devuelvan valores entre 0 y N-1, donde N es el tamaño en bits del signature file. Lo que se hace es hashear cada término del documento con cada una de dichas funciones y setear en 1 el bit que se encuentra en la posición de lo que devuelve cada función. Como ejemplo, utilizando signatures de 16 bits, si se utilizan 3 funciones para hashear el termino 'a' y devuelven los valores 0, 9 y 2 el signature de dicho termino es 1010 0000 0100 0000. Finalmente se hace un OR entre los signature de cada término del documento y con ello se obtiene lo que será el signature del documento. Puede darse que haya colisiones entre los valores que devuelven las funciones de hash para un termino, en cuyo caso en vez de setear en 1 N bits se seteará una cantidad menor.

Por ejemplo, si tenemos el documento "Ser o no ser esa es la cuestión", hasheamos cada término del documento con tres funciones de hashing, hacemos MOD 16 de los tres valores devueltos por las funciones y seteamos los bits correspondientes.

| | |
|---|---|
| ser = | 0100 0000 1000 0100 |
| o = | 0001 0000 0001 1000 |
| no= | 0011 0000 0000 0010 |
| ser = | 0100 0000 1000 0100 |
| esa = | 0100 0000 0000 0000 |
| es = | 0001 0001 0000 1000 |
| la = | 0001 0000 0000 0100 |
| cuestión = | 0101 0000 0000 0000 |
| Documento: | 0111 0001 1001 1110 |

Como se ve en los términos "esa", "la" y "cuestión" hubo colisiones por lo que no hay 3 bits seteados en 1 (o sea, dos o mas de las funciones devolvieron el mismo valor, por lo que la cantidad de bits en 1 es menor a la cantidad de funciones de hashing utilizadas). El valor del signature del documento que surge de hacer el OR entre todos los anteriores es: 0111 0001 1001 1110 (notar que es innecesario hashear mas de una vez un mismo término, como "ser" en el ejemplo, ya que el resultado es el mismo)

La forma en la cual se utiliza un signature file para resolver consultas es distinta de la forma en la cual se resuelven con un índice invertido; al buscar un término se debe hashearlo con todas las M funciones de hash y obtener su signature. Luego se debe recorrer el signature de cada documento buscando aquellos que tienen encendidos los bits que se encuentran encendidos en el signature del término. Para efectuar esto bastaría con hacer un AND binario entre el signature del término y el signature del documento, y se buscaran aquellos en los cuales el resultado de dicho AND sea el mismo signature del término. Estos documentos son candidatos a contener el término ya que no es obligatorio que los tengan pues un mismo bit puede ser encendido por varios términos, por lo que luego se deberá hacer una búsqueda dentro de los documentos candidatos para responder la consulta. Sin embargo, los otros documentos se pueden descartar porque es seguro que no contendrán el término buscado.

La ventaja principal de los signature-files reside en que ocupan muy poco espacio y que dicho espacio puede ser elegido en el momento de la creación por un valor conveniente.

## Construcción de Signature-Files – Bit Slices

En general uno de los problemas relacionados con el uso de signature files es la cantidad de accesos al disco que hay que hacer para comparar un signature contra todos los signatures de los documentos almacenados, para reducir el número de accesos a realizar los signatures se almacenan en forma de bit-slices. Para ello en vez de almacenar cada signature de cada documento por separado y andar fijándose uno por uno si es candidato a contener un término o no, los signatures de los documentos se guardan agrupados por bits, es decir, primero el primer bit de todos los signatures de los documentos, luego el segundo y así con todos los bits. Cada una de estas entradas se conoce como bit slice y en un signature-file de N bits tenemos N bit slices

Como ejemplo, supongamos que tenemos el siguiente signature-file.

| Doc | Signature |
|---|---|
| 1 | 0110000101000010 |
| 2 | 0100001000101010 |
| 3 | 1101000011001011 |
| 4 | 1101000001000100 |
| 5 | 0100111000010011 |

Como se ve se necesitarán 16 bit slices, siendo cada slice una columna de la tabla anterior:

| Slice | Valor |
|---|---|
| 0 | 00110 |
| 1 | 11111 |
| 2 | 10000 |
| 3 | 00110 |
| 4 | 00001 |
| 5 | 00001 |
| 6 | 01001 |
| 7 | 10000 |
| 8 | 00100 |
| 9 | 10110 |
| 10 | 01000 |
| 11 | 00001 |
| 12 | 01100 |
| 13 | 00010 |
| 14 | 11101 |
| 15 | 00101 |

Si queremos buscar un cierto término lo que se hace es hashearlo con las M funciones y luego recuperar únicamente los slices que correspondan con los valores devueltos por ellas. Por ejemplo si para un término las funciones dan como resultado 0,1 y 3 (o sea, el hash es 11010) recuperamos los slices 0, 1 y 3: 00110, 11111 y 00110. Con esto lo que tenemos son los bits 0, 1 y 3 del signature de cada documento, que son los que realmente nos interesan. Los documentos que pueden llegar a tener el termino buscado son aquellos que tienen esos 3 bits encendidos con lo que si se hace un And de los slices se obtendrá una tira de bits donde un 1 en la posición i indica que el documento i puede tener a dicho termino y un 0 indica que el documento i no tiene a dicho termino. Siguiendo el

ejemplo, haciendo un AND entre los 3 slices se obtiene 00110 con lo cual los documentos a chequear son los documentos 3 y 4 (los dos bits en uno). De esta forma solo se hacen solamente tres accesos al disco (uno por cada función de hashing).

Finalmente, en la construcción de un signature-file se debe especificar un tamaño para los signatures, este parámetro puede ser aprovechado para aumentar la eficiencia del signature file. Se buscará que la cantidad promedio de unos y ceros sea similar, ya que una mayor proporción de ceros con respecto a unos indica que el signature podría ser reducido de tamaño sin perder mucha efectividad; y en el caso contrario una mayor proporción de unos con respecto a ceros indica que es probable que se hayan producido muchas colisiones y por ende que una gran parte de los documentos candidatos a contener un termino finalmente no lo contendrán, por lo que se puede necesitar aumentar el tamaño del slice para disminuir las colisiones y evitarse de recorrer dichos documentos

# Optimizaciones

Independientemente de que tipo de índice tengamos podemos efectuar algunas técnicas que nos permitirán ahorrar aún más espacio o tiempo sin perder precisión a la hora de resolver consultas.

## Case-Folding

El Case-folding se utiliza cuando no es necesario distinguir entre mayúsculas y minúsculas para los términos. En este caso todos los términos se guardan en minúsculas (o mayúsculas si se quiere) y al recibir términos de una consulta se hace lo mismo. Este método es muy utilizado ya que en general una consulta por los términos “Barco” o “barco” se quiere que devuelva lo mismo

## Stop-Words

Las stop-words son palabras con una gran frecuencia y que no influyen en las consultas. Ejemplos de stop words son las palabras “el”, “la”, “en”, etc. Dada su gran frecuencia ocupan un espacio grande en el índice y no son muy influyentes en las consultas, con lo que se suelen obviar no incluyéndolas en el índice e ignorándolas al efectuarse consultas

## Stemming

La última optimización que veremos es la de stemming, técnica por la cual las palabras se reducen a una raíz común para todas las palabras de una familia, por ejemplo “brillo”, “brilloso” y “brillante” se reducen todas a brillo y se toman como si únicamente fueran ese término. Esta técnica reduce mucho el espacio ocupado por el índice y solamente requiere de un diccionario con la raíz de los términos a utilizarse tanto en la generación del índice como en las consultas (para transformar los términos de las consultas en sus raíces). En cuanto a los resultados que provoca, puede dar buenos resultados ya que devuelve documentos que no contienen exactamente los términos ingresados pero que pueden hablar del tema; pero a veces esto es justamente lo que no se quiere si se está buscando la palabra exacta ingresada

# Resolución de Consultas

La razón por la cual se construye un índice es justamente para poder resolver consultas sobre todos los documentos. En esta sección vamos a analizar distintos tipos de consultas y como resolverlas en forma eficiente con las estructuras de índice aprendidas

## Consultas Booleanas

La consulta base que se efectúa en este tipo de archivos es "Qué documentos tienen el termino X". Como vimos en las secciones previas, la forma de resolver este tipo de consulta es:

- En índices invertidos, se ubica el termino en el índice (con búsqueda binaria u otro método dependiendo del modo de almacenamiento del léxico utilizado), luego se accede a la lista de distancias relacionadas con dicho termino y esta se convierte en lista de documentos
- En signature files se hashea el termino, se buscan en los bit slices los documentos que tengan encendidos los mismos bits que dicho hash y se busca dentro de ellos cuales documentos realmente contienen dicho termino

Este tipo de consulta, que llamaremos consulta puntual, pertenece al grupo de consultas booleanas. La característica de este grupo es que en la consulta se especifica completamente los términos que deben o no estar y el resultado es simplemente el grupo de documentos que matchean con la consulta, sin ningún tipo de orden o jerarquía entre ellos. El resto de los tipos de consultas booleanas son:

- Consulta conjuntiva: Es una consulta el tipo "Término1 AND Término2 AND… AND TérminoN" donde se piden los documentos que contienen todos los términos de la consulta. Para resolver este tipo de consultas se efectúa una consulta puntual para el primer termino, luego otra consulta puntual para el segundo y solo se mantienen los documentos que aparecen en ambas consultas. Si se repite para todos los términos de la consulta, al final se tendrá la intersección entre los conjuntos de documentos devueltos por cada consulta puntual y que son los que satisfacen la consulta conjuntiva. Una forma de optimizar esta consulta es comenzar por el término de menor frecuencia, ya que de esta forma se trabajará con una menor cantidad de documentos y se requerirá menos tiempo para procesarla.
- Consulta disyuntiva: Es una consulta del tipo "Termino1 OR Termino2 OR … OR TerminoN" donde se piden los documentos que contengan al menos uno de los términos de la consulta. El método de resolución es similar al anterior salvo que se mantienen todos los documentos devueltos por alguna consulta puntual, resultando en una unión y no una intersección
- Consulta compuesta: Es una combinación de consultas conjuntivas y disyuntivas como por ejemplo "(Termino1 AND Termino2) OR Termino3". Para resolver este tipo de consultas se debe tener en cuenta el orden de los paréntesis e ir resolviéndolas con uniones y/o intersecciones de conjuntos de documentos. también en caso de que se especifique que un término no debe estar, deberán quitarse del conjunto los documentos devueltos por la consulta puntual de dicho término

# Wildcards

Las consultas con wildcards permiten utilizar "comodines" para indicar por ejemplo "todos los términos que comiencen con a y terminen con p", que se puede especificar como "a*p" si utilizamos el * como dicho comodín.

Este tipo de consultas es difícil de resolver con la estructura de índice vista ya que se debería utilizar la fuerza bruta, probando todos los términos que comienzan con a y ver si terminan con p, en cuyo caso se obtendría su lista de documentos. Para consultas del tipo "*os", como no se tiene ningún comienzo, se debe ver término por término si matchea con el patrón de búsqueda, cosa que en indicies de gran volumen es muy poco optimo. Sin embargo con algunos agregados podemos hacer que el tiempo de resolución de consultas con wildcards disminuya notablemente (aunque el espacio que ocupe el índice aumentará como consecuencia de esto)

Es importante notar que para wildcards trabajaremos únicamente con archivos invertidos ya que es imposible hashear un termino incompleto, así que para signature files la única opción es la búsqueda por fuerza bruta en todos los documentos

## N-Gramas

La primera solución del problema es conocida como N-gramas. Estos son una tira de N caracteres que aparecen en algún término. Por ejemplo, utilizando digramas (N=2), el termino "sacar" se divide en los digramas $s,sa,ac,ca,ar,r$ (el $ es un caracter especial utilizado para indicar el comienzo o fin de término)

El método de los N-gramas consiste en agregar al índice invertido un segundo nivel de índice en el que, así como de cada termino se guardaba en que documentos aparecía, de cada N-grama se guarda en que términos se encuentra. Por ejemplo si los términos fueran paz, pepa y pez, el segundo índice será: (para una mejor comprensión, los números de termino no fueron convertidos a distancias)

| Digrama | Ocurrencias | términos |
|---|---|---|
| $p | 3 | 1,2,3 |
| a$ | 1 | 2 |
| az | 1 | 1 |
| ep | 1 | 2 |
| ez | 1 | 3 |
| pa | 2 | 1,2 |
| pe | 2 | 2,3 |
| z$ | 2 | 1,3 |

Para resolver una consulta, se divide el patrón en sus digramas y se hace una consulta conjuntiva (los términos donde se encuentran todos los digramas) de ellos en el índice secundario. Por ejemplo, para la consulta "p*z", esta se descompone en 2 digramas y se hace la consulta "$p AND z$" en el índice secundario; luego se hace la intersección entre los conjuntos de términos (1,2,3) y (1,3) , con lo que se obtienen los términos "paz" y "pez".

Algo importante a notar es que el hecho de que un termino tenga todos los digramas buscados no significa que corresponda con el patrón de búsqueda. Por ejemplo, es bastante obvio que el termino "satisfacer" no corresponde con la consulta "sac*r", pero sin embargo los digramas de la consulta ($s, sa, ac, r$) están relacionados con dicho termino (ya que sus digramas son $s, sa, at, ti, is, sf, fa, ac, ce, er y r$). Por eso es que el segundo paso del método es revisar los términos de la

primer consulta realmente matcheen con el patrón de búsqueda. En general los casos en que ocurre esto no son muchos, pero es necesario este paso para no devolver documentos que no tienen que ver con la consulta. Además al haberse descartado una gran cantidad de términos, no es demasiado costoso.

El tercer y ultimo paso es, con los términos que matchean con el patrón, hacer una consulta disyuntiva en el indicie primario. Al finalizar este paso tendremos todos los documentos con al menos 1 término que matchee con el patrón de búsqueda.

El espacio del índice secundario generalmente es de un 50% del espacio del archivo invertido original, con lo que es importante ver si realmente es necesario que se puedan resolver este tipo de consultas (por ejemplo, en un buscador de paginas de Internet no tiene un gran uso), pero en caso de usarse, el tiempo de resolución de una consulta suele ser del 1% de lo que se tardaría por fuerza bruta

## Léxico rotado

La segunda solución al problema es la de léxicos rotados, que si bien ocupan inclusive mas espacio que los n gramas, hacen que las consultas se resuelvan aún más rápidamente. En este método también se utiliza un índice secundario en el que se guardan todas las rotaciones de un término. Por ejemplo, para el termino “cama” se guardan las rotaciones $cama, cama$, ama$c, ma$ca y a$cam, cada una apuntando al numero de termino que le corresponda a cama (notar que apuntan a un solo termino ya que no hay otro con la misma rotación)

Para resolver una consulta se rota el patrón (al que se le agrega el caracter especial de fin) hasta que los wildcards queden al final. Luego se busca en el índice secundario las rotaciones que comiencen de la misma forma (con una búsqueda binaria avanzada por ejemplo) y se obtienen los términos que matchean con el patrón, con los que se efectúa finalmente una consulta disyuntiva en el índice primario. Por ejemplo si la consulta es “ca*a” se rota hasta obtener “a$ca*” y luego se busca en el índice las entradas que comiencen con “a$ca”. El termino “cama” entonces será devuelto ya que hay una rotación (a$cam) que será devuelta en la búsqueda.

Los métodos para resolver consultas con 2 wildcards son mas complicados ya que si estos se encuentran separados (por ejemplo “c*m*”), se deberá buscar primero las rotaciones que comiencen igual hasta el primer wildcard, y luego ir probando una por una a ver si matchean con el patrón rotado completo. Siguiendo el ejemplo anterior, como hay un wildcard al final no es necesario rotar el patrón, luego se buscan las rotaciones que comiencen con c y se busca si la rotación matchea con “c*m*”. Así la consulta devolverá cama (por existir la rotación cama$) pero no casa, ya que pese a existir la rotación “casa$”, no matchea con “c*m*”

El léxico rotado tiene un tiempo de respuesta del 0.1% en comparación con la búsqueda por fuerza bruta pero ocupa un 250% mas que el índice invertido original

# Consultas Ranquedas

En las consultas ranqueadas se ingresa una lista de términos y la respuesta es una lista de documentos ordenadas de acuerdo a un grado de importancia que se obtiene en base a los términos buscados. En esta lista pueden haber documentos que no contienen a todos los términos buscados. Este tipo de consultas es muy utilizada por ejemplo en buscadores web, ya que uno ingresa algunas palabras claves que espera encontrar en la pagina y obtiene una lista de paginas generalmente relacionadas con el tema que buscaba. Sabemos ya como obtener que documentos tienen los términos de la consulta, lo que nos falta es alguna forma de poder clasificar a los documentos que se devuelvan, asignándole un puntaje a cada uno y luego ordenándolos de mayor a menor puntaje. A continuación veremos distintas metodologías utilizadas para resolver el problema, que buscan que el usuario final obtenga la respuesta que buscaba, cosa que no es tan simple de lograr. Utilizaremos

el siguiente ejemplo donde cada línea será un documento y cada palabra un termino. Para identificar mas fácil a cada uno de los 5 términos (Alberto, Bartolo, Cesar, Demian y Ernesto) los llamaremos a,b,c,d y e de acuerdo a la letra inicial de ellos

Alberto Cesar Alberto
Ernesto Alberto Bartolo Demian Alberto
Bartolo Demian Alberto
Bartolo Bartolo Alberto Alberto Bartolo Bartolo Alberto Demian Demian Ernesto
Ernesto Alberto Bartolo Demian Bartolo

## Coordinate Matching

En este método, el puntaje de un documento es la cantidad de términos de la consulta que contiene. Por ejemplo para la consulta (Ernesto, Alberto, Cesar) los puntajes son:

| Documento | Puntaje |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 1 |
| 4 | 2 |
| 5 | 2 |

Con lo cual los documentos mas relevantes son el uno, el dos, el cuatro y el cinco con igual puntaje.

Esta técnica podría verse también de una forma vectorial utilizando vectores de tantas dimensiones como términos hay. El puntaje de la consulta para cada documento sería el producto interno entre 2 vectores, el primero el vector de la consulta que tiene un 1 en la componente i si el término i se encuentra en la consulta y un 0 en el caso contrario; el segundo el vector de cada documento que tiene un 1 en la componente i si el término i se encuentra en el documento y un 0 si no. Para el ejemplo, el vector de la consulta (Q) y los de cada documento(Di) son:

Q=(1,0,1,0,1)
D1=(1,0,1,0,0)
D2=(1,1,0,1,1)
D3=(1,1,0,1,0)
D4=(1,1,0,1,1)
D5=(1,1,0,1,1)

Como ejemplo, el puntaje del documento 2 para dicha consulta es
D2 . Q = (1,1,0,1,1) . (1,0,1,0,1) = 1+0+0+0+1 = 2

Es importante ver que en esta forma vectorial los términos que aparezcan en las consultas y que no existan en ningún documento se ignorarán, por ende la consulta “Alberto Cesar Ernesto Rolo” se vería igual en forma vectorial aunque en la practica implicaría mas acceso a disco para ver que el término “Rolo” no está en ningún documento

Esta técnica tiene 3 problemas, que son:

- En primer lugar no tiene en cuenta la frecuencia de los términos, o sea la cantidad de veces que un termino aparece en el mismo documento. Utilizando el ejemplo, querríamos que el documento 2 tuviera más puntaje que el 5 ya que el término Alberto aparece 1 vez más que en el 5

- Asume que todos los términos tienen la misma importancia, cosa que no suele cumplirse. Por ejemplo en la consulta “trucos para el doom”, el término mas importante es “doom” ya que una página de trucos de otros juegos no es lo que se busca. En el ejemplo, el término “Cesar” solo aparece en el documento 1, por lo tanto cualquier consulta que involucre “Cesar” debería darle un muy buen puntaje a dicho documento ya que es el único que contiene uno de los 3 términos de la búsqueda.
- Finalmente, este método favorece a los documentos más largos ya que tienen mas cantidad de termino y por ende mas probabilidades de tener un termino de la consulta.

Veremos a continuación distintas formas de solucionar estos problemas

## Producto Interno

Resolver el primer problema implica una modificación al archivo invertido, ya que en la entrada de cada termino se debe indicar no solo en que documentos aparece sino con que frecuencia lo hace. Un entrada del archivo invertido modificada sería de la forma.

Demian;(1,0)(2,1);(3,1);(4,2);(5,1)

Obviamente no se guardaría directamente así sino con pares distancia-frecuencia (sabiendo que para los términos donde no aparece su frecuencia es 0). Con este tipo de índice se podría cambiar el vector de cada documento indicando la frecuencia del término en él. Para el ejemplo sería:

D1=(2,0,1,0,0)
D2=(2,1,0,1,1)
D3=(1,1,0,1,0)
D4=(3,4,0,2,1)
D5=(1,2,0,1,1)

Si hacemos el producto interno con nuestra consulta, que sigue siendo (1,0,1,0,1), obtenemos.

D1 = (2,0,1,0,0) . (1,0,1,0,1) = 3
D2 = (2,1,0,1,1) . (1,0,1,0,1) = 3
D3 = (1,1,0,1,0) . (1,0,1,0,1) = 1
D4 = (3,4,0,2,1) . (1,0,1,0,1) = 4
D5 = (1,2,0,1,1) . (1,0,1,0,1) = 2

Por ende los nuevos puntajes son:

| Documento | Puntaje |
|---|---|
| 1 | 3 |
| 2 | 3 |
| 3 | 1 |
| 4 | 4 |
| 5 | 2 |

Ahora el documento con mayor puntaje es el 4, pero esto ocurre únicamente por ser el más largo, problema que aún no hemos resuelto. Sin embargo vemos que el primer problema se pudo solucionar ya que el documento 2 ahora tiene mas puntaje que el 5.

# Producto Interno Mejorado

En el producto interno mejorado se comienza a tener en cuenta la importancia de un termino, utilizando una formula de George Zipf que dice que un término es más importante mientras menos frecuencia de aparición tenga. La formula utilizara para calcular el peso (o la importancia) de un término es:

$$wt = \log_{10} \frac{N}{ft}$$

Siendo wt el peso del término, N la cantidad de objetos del universo (en nuestro caso la cantidad total de documentos) y ft la frecuencia del objeto dentro del universo (en nuestro caso, la cantidad de documentos en los que aparece el término). Aplicando esta fórmula, modificaremos el método del producto interno haciendo que el peso de un termino para un documento sea la frecuencia del termino en dicho documento multiplicada por el peso del termino:

$$wtd = ftd * \log_{10} \frac{N}{ft}$$

Siendo:

ftd: La frecuencia del término dentro del documento.
N: La cantidad de documentos en total.
ft: La frecuencia del término (cantidad de documentos donde aparece)

Para poder utilizar el producto interno mejorado será necesario, al momento de construir el índice, calcular el peso de cada termino y guardarlo para ahorrarse el hacer dicho calculo en cada consulta. Usando este esquema el peso de cada término es:

Peso(A) = log(5/5) = 0
Peso(B) = log(5/4) = 0.1
Peso(C) = log(5/1) = 0.7
Peso(D) = log(5/4) = 0.1
Peso(E) = log(5/3) = 0.2

El vector de cada documento es de la forma:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| D1 = ( | 0 | ; | 0 | ; | 0.7 | ; | 0 | ; | 0 | ) |
| D2 = ( | 0 | ; | 0.1 | ; | 0 | ; | 0.1 | ; | 0.2 | ) |
| D3 = ( | 0 | ; | 0.1 | ; | 0 | ; | 0.1 | ; | 0 | ) |
| D4 = ( | 0 | ; | 0.4 | ; | 0 | ; | 0.2 | ; | 0.2 | ) |
| D5 = ( | 0 | ; | 0.2 | ; | 0 | ; | 0.1 | ; | 0.2 | ) |

Como se puede ver, el término más importante es el C que aparece únicamente en el documento 1. En cambio el término A tiene una importancia de 0, esto es porque aparece en todos los documentos y por ende incluir o no en una consulta a dicho documento no cambiará en nada el resultado de esta. Este método también sirve para asignarle un peso muy bajo a los términos muy frecuentes, pero sería un error pensar que al utilizar este método es innecesario eliminar las stop words considerando que como su peso será muy pequeño no influirán en la consulta, ya que aunque es cierto que influirán muy poco igualmente estarían ocupando espacio innecesario en nuestro índice

Haciendo el producto interno entre los nuevos vectores de cada documento se obtienen los puntajes, que son:

| Documento | Puntaje |
|---|---|
| 1 | 0.7 |
| 2 | 0.2 |
| 3 | 0 |
| 4 | 0.2 |
| 5 | 0.2 |

Logramos con esto que el primer documento sea el más relevante (cosa que queríamos ya que en él aparece un término muy importante) pero todavía nos falta corregir el último punto ya que se ve que los documentos largos con gran cantidad de palabras siguen siendo teniendo ventaja sobre otros documentos mas chicos (se ve ya que el documento 4 tiene el mismo puntaje que el 5 pese a que tiene muchos mas términos que no importan para la consulta). Una solución posible consiste en dividir el resultado del producto interno por la longitud del documento, pero existe un método con mejores resultados.

## Modelos de espacios vectoriales – método del coseno

Dado que venimos calculando el peso de cada documento en forma vectorial, podemos utilizar una visión mas algebraica para encarar el problema. Siendo que cada componente de los vectores con que trabajamos es el peso de un termino en dicho documento, podríamos pensar que el contenido de dos vectores que tienen aproximadamente la misma dirección es similar. Lo que nos va a interesar entonces es la dirección de los vectores y para evaluar cuan distinta es la dirección de un vector con otro nos importará cuanto vale el ángulo entre ellos. Podemos obtener el coseno de dicho ángulo utilizando la ecuación de producto cartesiano entre los vectores X e Y:

$$X * Y = |X| * |Y| * \cos f$$

$$\cos f = \frac{X * Y}{|X| * |Y|}$$

$$\cos f = \frac{\sum Xi * Yi}{\sqrt{\sum Xi^2} * \sqrt{\sum Yi^2}}$$

Cuanto mayor es el coseno menor es el ángulo así que más parecida es la dirección de los vectores, con lo cual se puede utilizar el coseno del ángulo entre el vector de un documento y el vector consulta como puntaje del documento, ya que mientras mas cercano a 1 sea, significa que más parecidos son dichos vectores y entonces que mas se ajusta el documento a la consulta. Siendo Q el vector consulta y W el vector documento el puntaje (coseno) que obtiene el documento es:.

$$\cos f = \frac{Q * D}{|Q| * |D|}$$

$$= \frac{1}{Wq * Wd} * \sum Wqi * Wdi$$

$$Wd = \sqrt{\sum Wdi^2}$$

$$Wq = \sqrt{\sum Wqi^2}$$

$$Wdi = fdti * \log_{10} \frac{N}{fti}$$

$$Wqi = \log_{10} \frac{N}{ti} * Qi$$

Un cambio que se ve es que ahora en el vector de la consulta (Q) se tienen en cuenta los pesos de cada término en vez de poner unos y ceros. Esto es porque ahora se están comparando angulos y el angulo entre el vector de un documento que tuviera justamente los términos que busco y el vector de la consulta con unos y ceros no sería de 0 grados (que daría el puntaje mas alto que puede tener un documento). Esta modificación al vector de consulta es necesaria para que los angulos el vector de consulta y documentos que deban tener un buen puntaje sea más reducido. Entonces, el vector Q será

Q = ( 0 ; 0 ; 0.7 ; 0 ; 0.2 )

Para utilizar el método se deberá primero calcular la norma de cada vector como la raiz cuadrada de la suma del cuadrado de cada componente. Como ejemplo para el documento 4 su norma es $\sqrt{0^2 + 0.4^2 + 0^2 + 0.2^2 + 0.2^2} = 0.2 = 0.49$

Las normas son:

| | | |
|---|---|---|
| D1\| = 0.7 | \|D2\|= 0.24 | \|D3\|=0.14 |
| \|D4\|=0.49 | \|D5\|= 0.3 | \|Q\| = 0.73 |

Igualmente se ve que como en todos los cálculos de puntaje se esta dividiendo por la norma de Q, que no varía de documento en documento, esto podría no hacerse y en cambio utilizar como puntaje el valor del coseno del ángulo multiplicado por la norma de Q. Las posiciones de los documentos serían las mismas pero en este apunte igualmente se dividirá por la norma de Q para evitar confusiones ya que sería raro que el método del coseno diera un valor mayor a 1 como puntaje

Con los vectores de los documentos (son los mismos que en el producto interno mejorado), el vector de la consulta calculado previamente y las normas de cada vector, se puede determinar el puntaje de cada término como el producto interno del vector del documento y el vector de la consulta dividido por la norma del vector de la consulta y del vector del documento. Por ejemplo

para el documento 4 su puntaje es D1 . Q / (|D1| * |Q|) = 0.04 / ( 0.49 * 0.73) = 0.11. Los puntajes de cada término son:

| Documento | Puntaje |
|---|---|
| 1 | 0.96 |
| 2 | 0.23 |
| 3 | 0 |
| 4 | 0.11 |
| 5 | 0.18 |

Finalmente logramos lo que buscábamos, el documento 1 es el mas relevante por contener a un término de gran importancia y se está teniendo en cuenta la frecuencia de los términos en los documentos sin por eso privilegiar a los más largos (Se ve en el caso del 4 que no recibió un gran puntaje por tener una gran cantidad de términos que no eran buscados por la consulta).

## Phrase queries (consultas de frase)

En las phrase queries no solo importa que todos los términos de la consulta aparezcan sino que además importa que sigan el orden especificado. Estas consultas son bastante frecuentes y en caso de no ser bien tratadas suelen traer grandes problemas. Por ejemplo, si en vez de usar una búsqueda que tenga en cuenta el orden se hace una simple consulta conjuntiva, al buscar información sobre el libro "2001 una odisea del espacio" el sistema de búsqueda de la librería podría devolver el siguiente documento:

"La Odisea. Autor: Homero. Editorial Espacio. Sumario: Se relata el retorno del ingenioso Ulises a su tierra natal luego de la guerra de Troya. Ultima edición: Agosto 2001"

Como se puede apreciar, el libro devuelto no tiene nada que ver con el buscado. No es una opción aceptable el solo efectuar consultas sobre el título, ya que alguien puede querer buscar todos los libros de una editorial dada, o de un autor, especificando el nombre completo que consta de varios términos y que debe cumplir un orden.

Una técnica para resolver este tipo de consultas implica una nueva reestructuración del índice. Ahora, para cada término, no solo se guardará en qué documentos aparece sino que para cada documento se guardará la frecuencia del término dentro del documento y los offsets que indican en que posición del documento se encuentra dicho término. Supongamos que el primer documento es:

"quiero y no quiero querer a quien no queriendo quiero, he querido sin querer y estoy sin querer queriendo"

Para el término "quiero" tendríamos la siguiente entrada:

quiero – Doc1 – Freq=3 – Offsets: 1 | 4 | 10

Que indica que el término "Quiero" aparece 3 veces en el documento 1, en el primer, cuarto y décimo término dentro del mismo. Utilizando las técnicas vistas anteriormente, se utilizaría un método para almacenar el término "quiero", algún tipo de codificación para las distancias, y faltaría definir una codificación para los offsets. Como se puede apreciar, estos son siempre crecientes con lo que se podrían transformar a "distancias de offsets" en los que para el ejemplo anterior tendríamos las distancias 1, 3 y 6. Luego se podría utilizar cualquiera de las técnicas previamente vistas para compresión de distancias, teniendo en cuenta que para los offsets lo más probable no son las distancias muy bajas, sino distancias del orden 10 a 100 (utilizar unario seria excesivamente costoso en cuanto a espacio en disco consumido).

La resolución de consultas con este método, similar a la de consultas conjuntivas, seria de la siguiente forma:

- Nuevamente, con la intención de utilizar la menor cantidad posible de memoria, se ordenan los términos de la consulta en cuanto a su frecuencia, pero sin perder el dato de que posición ocupan en la consulta

• Para el primer término (el menos frecuente) se obtienen los documentos en que aparece junto con todos los offsets para cada documento

• Luego, mientras queden otros términos sin analizar, se toma el de menor frecuencia de ellos y se recorre su lista de documentos, quedándose únicamente con los documentos en que aparecieron todos los términos anteriores y que además presenten al menos un offset que sea candidato a tener la frase, es decir, un offset tal que todos los términos ya analizados aparezcan en el orden necesario con respecto a él. Todos los offsets que no sean candidatos, serán descartados. Si un documento no aparece en ambos conjuntos o aparece pero no queda con offsets candidatos, se lo descartará

Para comprender esto último, supongamos que hacemos la consulta “Odisea del espacio”, lo mas probable es que el término odisea sea el menos frecuente. Por simplicidad, supongamos que la consulta devuelve 1 solo documento con los offsets 10, 20 y 30. En el siguiente paso, tomamos el segundo término menos frecuente que seguramente será “espacio” y supongamos que nos devuelve el mismo documento pero con los offsets 11, 22 y 33. Nos quedaremos entonces con el documento y únicamente el offset 20, ya que en el offset 10 comienza la frase “Odisea espacio” y en el término 33 la frase “Odisea XXXX XXXX espacio” donde XXXX es un término por ahora desconocido, pero que no es ni “Odisea” ni “espacio”, por lo que no concordará con la frase total. En cambio en el offset 20 comienza la frase “Odisea XXXX espacio”, que es candidata a matchear con la consulta. únicamente se devolverá el documento si también aparece en la lista de documentos del término “del” y con el offset 21.

Si el primer término a buscar (el menos frecuente) no fuera el primero de la consulta, no se deberán almacenar sus offsets sino que se deberán restarle a esos valores la posición del término dentro de la consulta menos 1 ya que esos son los offsets candidatos para ser inicio de la frase. Siguiendo el ejemplo anterior, si el término menos frecuente fuera “espacio”, en vez de guardar los offsets 11, 22 y 33 se guardarían los offsets 9, 20 y 31 (se le restó 2 porque “espacio” es el tercer término de la consulta).

Para ahorrarse este tipo de cuentas se efectúa una normalización que consiste en preprocesar la lista de offsets de un termino, restándole a cada uno la posición del término en la consulta. Al hacer esto, lo que se está manteniendo en memoria son los offsets en que comienzan las frases candidatas, con lo que al comparar contra una nueva lista normalizada se guardan los documentos en que al menos coincida 1 offset normalizado

Si bien el método analizado anteriormente logra resolver las phrase queries, no es tan eficiente ya que se deberá manejar una gran cantidad de offsets para resolver la consulta

## Los índices nextword

En los índices nextword se indexan los términos de a pares. Es un índice de tres niveles, el primero es un término al que denominaremos inicial, luego el segundo nivel es otro término al que denominaremos final, y finalmente en el tercer nivel se encuentran los documentos y offsets en que se encuentran el par de términos. Por ejemplo, una entrada de dicho índice podría ser

Donovan => Term = 1 (Mc – Doc1 – Frec1 – 6)

Mc => Term = 3 - (Donalds – Doc3 – Freq=1 – 3; Nabb – Doc1 – Freq=2 – 7 | 9 – Doc2 – Freq=1 – 5 ; Nair – Doc2 – Freq=1 – 14)

De esta entrada se ve que la frase “Mc Donalds” aparece en el 3er término del documento 3, “Mc Nabb” aparece en el 7mo y 9no término del documento 1 y en el 5to del documento 2, “Mc Nair” aparece en el 14vo término del documento 1 y “Donovan Mc” aparece en el 6to término del documento 1. Para resolver la consulta “Donovan Mc Nabb” se podría subdividir en consultas “Donovan Mc” y “Mc Nabb”. Utilizando una resolución similar a la vista previamente, llegaremos a que hay una ocurrencia de esta frase desde el sexto offset del documento 1.

Resolver una consulta con nextwords es mucho mas rápido, ya que la cantidad de offsets para un par de términos es mucho menor en comparación con la modificación al índice invertido

anteriormente propuesta. Además, se da el caso de que el promedio de longitud de phrase queries es muy similar a 2 términos por consulta, con lo cual no siempre habrá que hacer más de un acceso al índice nextword. Sin embargo, este método necesita de mucho más espacio en disco, aproximadamente el doble del que ocupa un índice invertido común, ya que además de los offsets se debe guardar la cantidad de términos finales que tiene un término inicial y se estarán repitiendo varias veces un mismo término (aunque obviamente no se utilizará la cadena de caracteres sino el offset en el léxico o algo similar; el repetir varias veces dicho offset implica un mayor uso de disco)

Nos queda entonces encontrar un punto medio entre el "poco" espacio en disco utilizado por la modificación propuesta a los índices invertidos, pero con su mala performance, y los buenos tiempos de respuesta de los nextwords, pero con su gran tamaño. Justamente lo que haremos será utilizar un poco de cada una de las técnicas

Los nextwords tienen una mayor performance cuando los pares de término son muy frecuentemente utilizados, como "en el", "de la", etc. Esto es debido a que la gran cantidad de offsets de cada frase se puede acceder de forma rápida, cuando con la alternativa de los índices invertidos se deberían analizar muchos más offsets asociados con cada término por separado. Sin embargo, para términos no tan probables, la cantidad de offsets en el índice invertido será de un orden similar a la que se pueda encontrar en un par de términos que lo incluyeran. Es de suponer entonces que de archivar nextwords únicamente de términos muy frecuentes nos aumentará la performance y no aumentará tanto el espacio utilizado. De hecho, al probar esta idea se ha visto que con utilizar nextwords de únicamente los 3 términos más frecuentes se logró disminuir el tiempo de consulta promedio a la mitad, aumentando el espacio en disco en sólo un 10% comparado con el índice invertido modificado. En base a las necesidades de tiempo de respuesta y a las limitaciones de espacio, se puede definir qué cantidad de términos frecuentes utilizar en stopwords para tener la mejor performance posible

Una ultima consideración es que si es necesario que el sistema resuelva phrase queries, el hacer stopping puede traer problemas ya que faltarán los offsets de dichos términos y deberá revisarse si el offset candidato realmente tiene la frase completa accediendo al documento y procesándolo, cosa que será mas costoso ya que no se cuenta con una posición relativa en bytes como para hacer un seek sino que se cuenta con un número de término y se deberán leer los N-1 términos anteriores hasta llegar al inicio de la frase candidata. Guardar offsets en bytes en vez que en números de término no es aceptable ya que se aumenta demasiado el espacio ocupado en disco

# Consultas por proximidad

Todo lo visto para resolver phrase queries puede utilizarse para resolver consultas de proximidad. En esta no solo se desea que los términos se encuentren en el documento sino que además se quiere que se encuentren próximos. Como ejemplo, si alguien busca "cura AND gripe", no le interesará un documento que en el primer párrafo habla de la cura de otra enfermedad y en el ultimo se habla de la gripe en general sin tener en cuenta su cura

Para resolver este tipo de consultas, contando con los offsets se debe definir un umbral de proximidad (por ejemplo, 2 términos están próximos si distan ente si como mucho 100 términos) y al momento de efectuar la consulta devolver documentos que cumplan además esta condición. Una alternativa es dejarle definir al usuario qué considera como próximo y utilizar ese valor como parámetro

Otro uso importante de este tipo de consultas es en las rankeadas, pudiendose aumentar el puntaje de un documento si 2 o más términos se encuentran próximos (utilizando algun tipo de ponderacion junto con la importancia de cada termino). Esta idea es muy utilizada y da muy buenos resultados para el usuario final

# Referencias

Algunos de los papers consultados para la realización del apunte son:


- Self indexing Inverted Files for fast text retrieval – Alistair Moffat, Justin Zobel – Feb 1994
- Fast ranking in limited space - Alistair Moffat, Justin Zobel – May 1993
- Efficient phrase querying with an auxiliary index – Dirk Bahle, Hugh Williams, Justin Zobel – 2001
- Compression and fast indexing for multi-gigabyte text databases