Sistemas de recuperación total de textos

Full retrieval text systems

INDICE:

Introducción

Este apunte trata sobre un campo específico de las bases de datos en el cual se han realizado avances muy importantes en los últimos 5 años, estos sistemas denominados también sistemas de manejo de texto son grandes bases de datos en las cuales la información e encuentra almacenada fundamentalmente en forma de textos. Algunas características fundamentales de estas bases de datos y que permiten distinguirlas de las demás son:

- Enormes volúmenes de información en forma de textos.
- Frecuentemente los textos son acompañados de imágenes o gráficos.
- Se requieren índices especiales que permitan realizar consultas sobre los textos almacenados.
- El espacio ocupado, tanto por los textos como por los índices es crítico.
- Las consultas deben resolverse en forma rápida aun para colecciones de gran tamaño.

Varios sistemas que presentaban este tipo de requerimientos fueron implantados usando bases de datos relacionales o incluso archivos planos resultando en un fracaso absoluto. Un ejemplo muy actual y que muestra el enorme espectro que cubren estos sistemas son las enciclopedias, tutoriales o textos que se distribuyen en Cd-Rom, este tipo de aplicación cubre todos y cada uno de los requisitos que mencionábamos antes.

Este tipo de sistemas se caracterizan por reunir una enorme cantidad de información textual y el objetivo de automatizar esta información es poder realizar consultas sobre los datos que manualmente serían irrealizables o bien insumirían una enorme cantidad de tiempo. Por ejemplo:

- Cuales son los textos de mayor relevancia para un conjunto de temas.
- En qué textos puede encontrarse una determinada palabra.
- En qué textos pueden encontrarse dos o mas palabras o bien una palabra u otra.

Para la lingüística, por ejemplo analizar en que contextos utilizo un autor una determinada palabra es un trabajo de suma importancia, este tipo de trabajo se denomina concordancia. Una concordancia de las obras de Shakespeare insumió en el siglo pasado la vida de una persona, sus hijos y los hijos de estos. Hoy en día estas otrora maratónicas tareas ya no cobran vidas y pueden realizarse en lapsos de tiempo realmente sorprendentes.

Básicamente estos sistemas están compuestos por textos índices almacenados en disco y un sistema o motor de interpretación, optimización, resolución de consultas y recuperación de información en base a las consultas. Un abogado que quiera obtener aquellos artículos legales en los cuales se trate un determinado tema no va a poder entender que la consulta insuma varias semanas ni tampoco que se necesiten 20Gb de espacio en disco para resolverla, aunque suene exagerado esto es exactamente lo que se necesitaba para dicha consulta usando una base relacional para guardar los textos.

Los sistemas de recuperación total de textos pueden construirse en forma independiente, anexarse como modulo a un sistema de mayor envergadura o bien anexarse como una funcionalidad más de una base de datos relacional. Curiosamente las bases de datos orientadas a objetos parecen mas flexibles para incorporar estas funciones a los objetos de tipo 'texto', mas allá de esta facilidad la usabilidad real de una base de datos orientada a objetos permanece en discusión.

En este apunte trataremos los temas más delicados de este tipo de sistemas: el almacenamiento de los textos, el almacenamiento y construcción de los índices y la resolución de

consultas. Muchos de los temas aquí tratados son bastante nuevos y resultan muy interesantes no solo para este tipo de sistemas sino también en otros campos.

Pre-requisitos: Este es un apunte netamente técnico, si bien los temas han sido tratados en profundidad y con ejemplos en casi todos los casos hay algunos temas que el lector debe conocer para poder estudiar en profundidad este tema. En este apunte suponemos que el lector tiene conocimientos de programación y estructuras de datos complejas (árboles, tries, hash-tables etc). Nociones básicas de cálculo logarítmico. Conocimientos de probabilidad y estadística. Conocimientos de teoría de la información y compresión de datos, códigos prefijos, códigos de huffman, compresión aritmética. Manejo de índices, índices invertidos árboles B, B+. Sort interno y externo, Natural Selection, Replacement Selection, Key-Sort. etc.

Términos y Documentos

La construcción de un índice se hace en base a dos palabras que debemos definir: términos y documentos. En líneas generales, un índice sirve para buscar dentro de un texto determinado uno o más términos, los resultados son aquellos documentos que matchean la consulta formulada. Dependiendo de la aplicación para la cual vaya a usarse la base de datos los términos y los documentos pueden ser definidos en forma muy variada. Por ejemplo podemos decir que un término es una palabra y que el documento está especificado por un número de párrafo o bien que los términos son temas y que el documento es un determinado archivo.

Término: Un término es la unidad mínima de información que puede buscarse en un texto. En el 99% de los casos se define que un término es una palabra. En general se limita la longitud de los términos a 256 caracteres y además no se consideran términos a los números de más de 4 dígitos.

Documento: Un documento es la unidad mínima de información que se recupera del texto. Al realizarse una consulta la misma devuelve el o los documentos que matchean con la consulta formulada. En general un documento es una línea, un párrafo, una página o bien un archivo completo de texto.

Índices Invertidos

Un índice invertido es un listado de todos los términos de un texto acompañados por todos los números de documento en los cuales aparecen dichos términos. Se lo denomina así por ser la "inversión" de la forma de mostrar los datos, se pasa de tener muchos documentos pudiendo para cada uno de ellos obtener cada uno con sus términos a tener muchos términos pudiendo para cada uno de ellos obtener los documentos en que aparecen. La forma de un índice invertido es:

Término1 + Número_ocurrencias + Documento1 + Documento2 ..... + DocumentoN.
....
TérminoN

Por ejemplo, si tenemos el siguiente fragmento de texto:

> En algunos casos, la verdad es la única
> fuente de iluminación con la que pueden
> contar algunos tristes personajes que
> ni siquiera cuentan con la iluminación
> natural con la que todos nacemos.

Supongamos que nuestros términos son las palabras del texto y que definimos que un documento es una línea. Tendremos pues 5 documentos en total. El índice invertido para el fragmento en cuestión sería de la forma:

Palabra	**Ocurrencias**	**Documentos.**
En	1	1
algunos	1	1
casos	1	1
la	4	1,2,4,5
verdad	1	1
es	1	1
única	1	1
fuente	1	2
de	1	2
iluminación	2	2,4
con	3	2,4,5
que	3	2,3,5
pueden	1	2
contar	1	3
algunos	1	3
tristes	1	3
personajes	1	3
ni	1	4
siquiera	1	4
cuentan	1	4
natural	1	5
todos	1	5
nacemos	1	5

Cada uno de los números de documento que acompañan a los términos indica un número de línea en el texto donde aparece el término, como puede verse si un término aparece mas de una vez en una línea solo se incluye una única referencia. A cada una de estas referencias la denominaremos 'puntero'

Además del ejemplo muy breve de nuestro pequeñísimo fragmento de texto vamos a ejemplificar en este apunte los temas analizados a partir de datos de tres bases de datos reales que llamaremos Biblia, GNU y TREC. Biblia es el texto completo de la Biblia, considerando cada versículo como un documento. GNU es una colección de textos breves sobre computación que recopilo GNU, cada documentos es un archivo independiente. TREC, por último es una gran colección de artículos de todo tipo (finanzas, ciencia y tecnología). Los datos estadísticos de estas tres colecciones son:

	Bible	**GNU**	**TREC**
Documentos	31102	64267	742358
Términos	884988	2570939	335856749
Términos distintos	9020	47064	538244
Punteros	699131	2228135	136010026
Tamaño	4,33 Mb	14,05 Mb	2,054 Gb

El contenido de los punteros puede ser simplemente un número de documento o puede hacerse más específico, por ejemplo indicando número de documento, línea y palabra. Con esta información adicional podemos hacer consultas por proximidad, es decir dos o mas palabras que no estén separadas por mas de x cantidad de palabras. La información que contienen los punteros define la granularidad del índice, a mayor granularidad más precisión para las consultas y la información recuperada pero a la vez mayor espacio se necesita para el índice. Para simplificar vamos a suponer que cada puntero indica únicamente un número de documento.

Para estimar el espacio ocupado por un archivo invertido podemos hacer el siguiente análisis: cada palabra en inglés o castellano ocupa en promedio 5 caracteres y esta seguida por uno o dos espacios en blanco y/o signos de puntuación. Si cada puntero ocupa 32 bits tenemos 4 bytes de punteros por cada 6 bytes de datos, esto es verdad si consideramos que cada término no aparece mas de una vez en cada documento lo cual no es verdad, con esta última consideración se puede estimar que la longitud de un archivo invertido es de aproximadamente el 55% de la longitud del archivo de datos.

Compresión de Archivos Invertidos.

Para bases de datos grandes, y para las que no son grandes en realidad también, el tamaño de un archivo invertido es relativamente excesivo, para la base TREC, por ejemplo, el archivo invertido ocuparía sin comprimir alrededor de 1,5Gb, lo cual definitivamente es mucho espacio. Para reducir el tamaño de los archivos invertidos se almacenan los mismos en forma comprimida.

Sabemos que un archivo comprimido esta formado por términos y luego una serie de números por término indicando aquellos documentos en los cuales puede encontrarse el término, en primer lugar vamos a analizar como comprimir la lista de números de documentos.

Compresión de punteros.

Supongamos que el término "elefante" aparece en un texto en los documentos 3,5,20,21,23,76,77,78. La entrada en un índice invertido para dicho término es de la forma:

Elefante;8;3,5,20,21,23,76,77,78

El 8 es la frecuencia del término y corresponde a la cantidad de documentos en donde aparece el término, siempre que hablemos de frecuencia de un término en este apunte lo haremos contando cantidad de documentos y no ocurrencias individuales del término en todo el texto.

Como puede verse la lista de punteros esta formada por números en orden ascendente, otra forma de expresar la misma secuencia es escribir el primer número de documento y luego la distancia hasta el próximo número y así sucesivamente. En nuestro caso quedaría.

Elefante;8;3,2,15,1,2,53,1,1

La nueva lista la denominaremos lista de 'distancias'. Aunque no se note a simple vista la lista de distancias se puede comprimir con mayor facilidad que la lista de punteros. El motivo de esto reside en que para las palabras mas frecuentes (las que aparecen muchas veces en el texto) vamos a tener una lista de distancias cortas, mientras que las palabras más raras presentaran una lista de distancias relativamente grandes en promedio. De esta forma las distancias cortas son mucho más probables que las distancias largas y podemos aprovechar esta característica para comprimir la lista de distancias.

Analizaremos varios modelos de compresión para la lista de distancias. En primer lugar debemos distinguir dos tipos de modelos:

- Modelos globales:
  Utilizan el mismo modelo de compresión para todas las entradas del archivo invertido.
- Modelos locales:
  El modelo de compresión es distinto para cada entrada del archivo invertido y depende de algún parámetro almacenado en la entrada, como por ejemplo la frecuencia del término.

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

Modelos Globales.

Modelo Binario Plano.

Este es en realidad el modelo mas simple y en realidad no representa compresión alguna, lo que se hace es sabiendo que hay N documentos en el texto representar a cada puntero en log(N) bits.

Este modelo es bastante deficiente, en primer lugar supone que todas las distancias son equiprobables cosa que como vimos no es cierta y además usa longitudes fijas para todas las distancias. Si bien es el método mas simple (es terriblemente simple) es totalmente desaconsejable ya que como veremos es mucho mas conveniente un método que asigne a las distancias longitudes variables de acuerdo a su probabilidad.

Códigos Unarios.

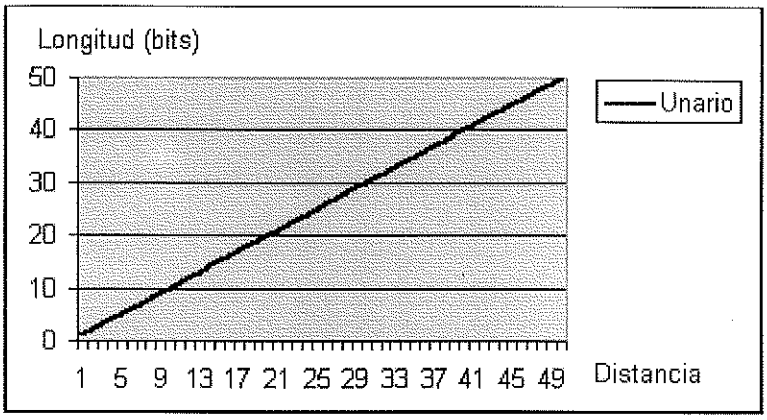
Este es un modelo en el cual cuanto mas chica es la distancia menos bits se necesitan para representarla.

En unario una distancia $X \geq 1$ se codifica como X-1 bits en uno y un bit en cero.

Ejemplos:

0 = 1
10 = 2
110 = 3
1110 = 4 etc...

Se ve que la longitud de una distancia X es de X bits por lo que la longitud en bits utilizada para representar las distancias es lineal. El código unario funciona bien para términos que aparecen en muchos documentos pero es muy malo cuando se encuentra una distancia grande



Código Gamma.

En un código gamma una distancia x se representa como:

- 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.

Ejemplo para X=9.

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

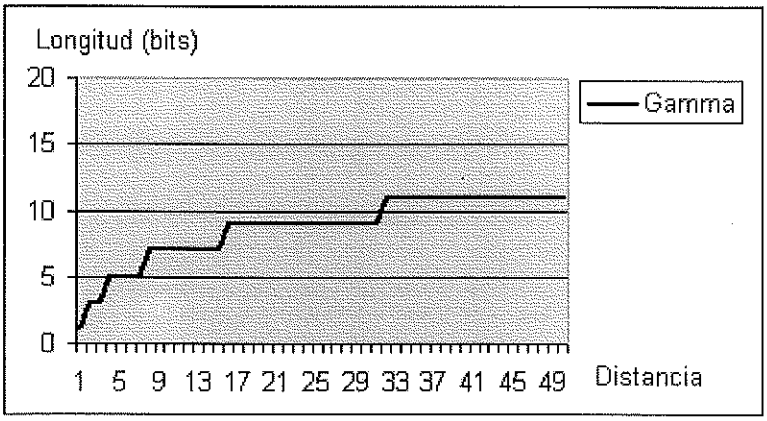
Seguido de 9-2^3 = 9-8 = 1 en binario usando 3 bits.

X = 1110 001 (7 bits)

Para decodificar se lee primero un número "c" en unario. Y luego se lee "b" en binario en c-1 bits. El número X se obtiene como X=2^(c-1)+b

Ejemplos:
1 = 0
2 = 100
3 = 101
4 = 11000
5 = 11001
6 = 11010
7 = 11011
8 = 1110000
9 = 1110001
etc.

La distancia x se representa en gamma usando $1+2\lfloor \log(x) \rfloor$ bits.



Códigos Delta.

Los códigos delta son similares a los códigos gamma, la distinción consiste en que la parte que en Gamma se expresaba en unario en el código delta se expresa en gamma. Con esto se

Ejemplo:
1110 001 = 9 en gamma.
1110 en unario es 4, 4 en gamma es 11000
9 en delta es 11000 001

Por lo tanto 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 $1+2\lfloor \log^2(2x) \rfloor + \lfloor \log(x) \rfloor$

Modelo global tipo Bernoulli.

Supongamos que es posible conocer el número 'f' que es la cantidad de punteros (distancias) a almacenar. Podemos entonces calcular la probabilidad de que un documento contenga una determinada palabra como un proceso Bernoulli en donde las palabras se distribuyen en forma pareja en el texto. La probabilidad se calcula como el número f dividido la cantidad de documentos del texto y luego dividido por la cantidad de términos del texto.

$p = f/N*n$

f = cantidad de punteros a almacenar.
N = cantidad de documentos.
n = cantidad de términos.

Notemos que p es un dato perteneciente a toda la base y es igual para todas las entradas del archivo invertido, por eso este es un modelo global. Para Biblia por ejemplo f=699131, N=31102 y n=9020. Por lo que p = 0,0025

De aquí se pude calcular la probabilidad de una distancia X como la probabilidad de X-1 no ocurrencias de probabilidad 1-p y luego una ocurrencia del término de probabilidad p.

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

Y la formula presenta una distribución geométrica. A partir de dichas probabilidades se podrían comprimir las distancias usando compresión aritmética, lo cual seria un tanto complejo, para simplificar el proceso de compresión de distancias se recurre a los Códigos de Golomb, llamados así por su descubridor A.Golomb un californiano que invento estos códigos en 1966.

Códigos de Golomb.

Para un determinado parámetro "b" un número X se codifica como.

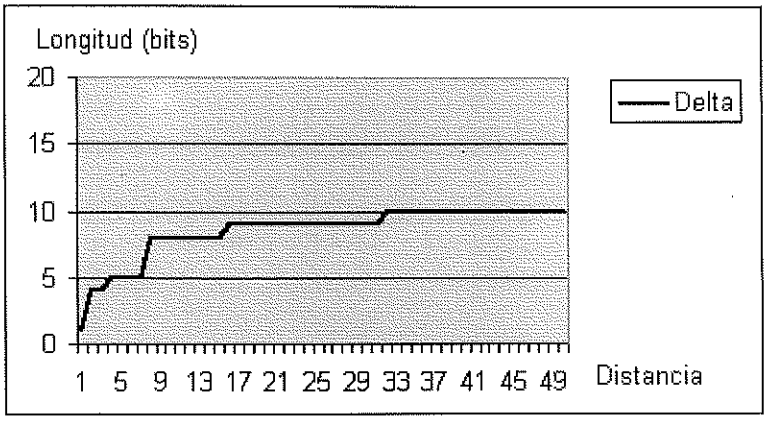
q+1 en unario.
r en binario en forma prefija.

Siendo $q=\left\lfloor \frac{X-1}{b} \right\rfloor$

r= x-qb-1

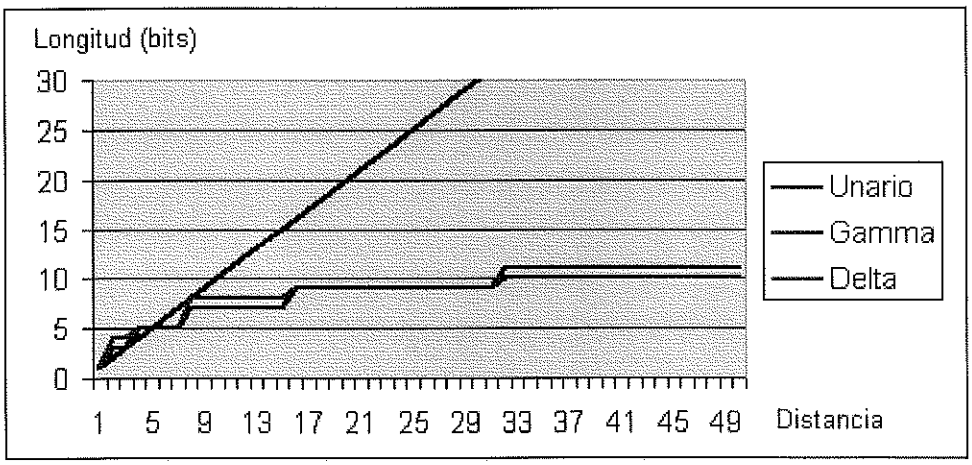
Por ejemplo para b = 3 hay tres restos (r) posibles (0,1,2) que en forma prefija se codifican como 0,10,11. Para b=6 hay seis restos posibles que en notación prefija son: 00, 01, 100, 101, 110, 111.

Ejemplos: Si tomamos b=3, para la distancia X=9, q=2 y r=2, por lo que la distancia 9 en golomb con b = 3 se codifica como 110 11. En cambio para b = 6 tenemos q = 1 y r = 2 por lo que se codifica como 10 100. En la siguiente tabla se muestran más ejemplos



Comparativo.

Número	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 gamma 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.

Número	b=3	b=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

El problema consiste, pues, en encontrar el "b" optimo para representar las distancias en función del número "p" obtenido del proceso Bernoulli. Se demuestra (aquí no por suerte) que "b" es óptimo si satisface:

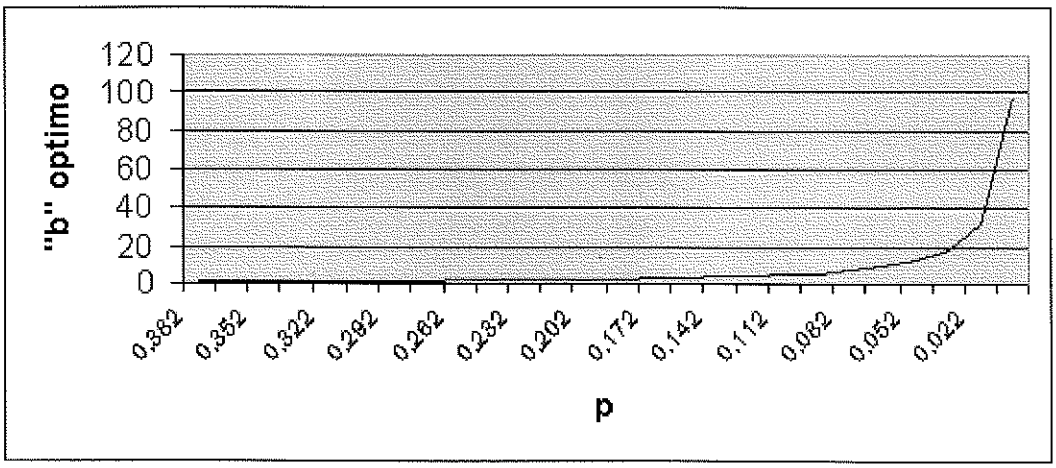
$(1-p)^b + (1-p)^{(b+1)} <= 1 < (1-p)^{(b-1)} + (1-p)^b$

De aquí puede despejarse "b":

$b = \log(2-p) / -\log(1-p)$

Tabla de "b" óptimos según el rango de "p"

Rango	"b" optimo.
0,3820 ...	1
0,2452 – 0,3819	2
0,1810 – 0,2451	3
0,1434 – 0,1809	4
0,1188 – 0,1433	5
0,1014 – 0,1187	6
0,0841 – 0,1013	7
0,0785 – 0,0840	8
0,0705 – 0,0784	9
0,0640 – 0,0704	10



En colecciones grandes donde p es muy chico el "b" óptimo tiende a ser un número grande, para Biblia por ejemplo el "b" óptimo es 278 y para TREC el "b" óptimo es 2036.

Si se diera el caso en el cual p>0,5 usando este modelo es conveniente invertir el archivo invertido!!! Es decir incluir por cada término punteros a los documentos que NO contienen dicho término.

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.

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 modelo de compresión usado por este modelo es bastante sencillo, se cuenta la frecuencia de cada distancia en el archivo invertido (leyendo todas las entradas) y luego se representan dichas distancias usando Huffman o bien compresión aritmética para codificar las distancias con longitud mínima.

En teoría este modelo parece ser muy bueno, sin embargo es apenas mejor que el código Delta, esto se debe a que hacer una estadística sobre las distancias mas frecuentes sin considerar la frecuencia de los términos es incorrecto. Los términos mas frecuentes tendrán muchos mas punteros (distancias) que aquellos que son menos probables (esto es obvio) entonces puede que sea mas conveniente (y de hecho lo es) representar con menor longitud a las distancias menores de los términos mas frecuentes sin importar cuales son dichas distancias.

Esto es precisamente lo que hacen los modelos locales.

Modelos Locales.

Modelo Local tipo Bernoulli.

En este modelo partimos de la suposición de que se conoce ft, frecuencia del término "t" en el texto (o sea en todos los N documentos). Luego para este modelo se utilizan los códigos de Golomb calculando "b" a partir de la frecuencia del texto. (Y no a partir de un p genérico para todo el texto como en el modelo global).

Dado ft => p = ft/N (probabilidad de ocurrencia de dicho término)

Y luego se calcula "b" en base a "p".

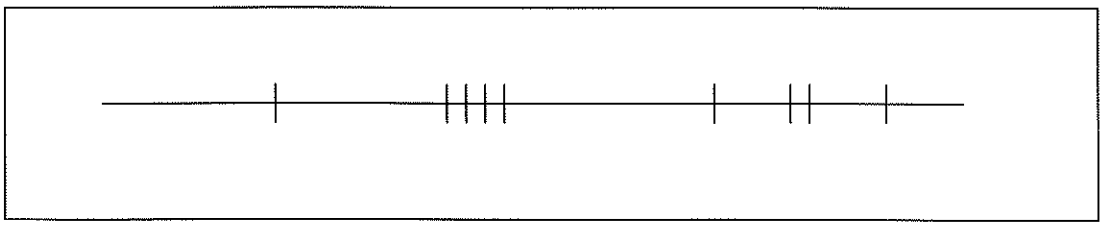
Por ejemplo en el texto GNU la palabra 'Hapax' usa b=514563 (19, 20 o 21 bits por distancia) mientras que algunos términos muy frecuentes usan b=7 (3 bits por distancia). Esto es favorable al promedio de 11 bits que se obtenía con "b" = 2036 en el modelo global.

Si un término es muy común un valor de "b"=1 degenera en un código unario para las distancias. Con lo cual lo que se guarda es en realidad un bitmap del término. De aquí puede deducirse que un archivo invertido jamás podrá ocupar mas espacio que un bitmap.

Para cada entrada del invertido hay que almacenar ft ya que de allí surgen los cálculos para poder decodificar las distancias. Como es común que existan muchas frecuencias chicas para guardar los ft en el archivo invertido es conveniente usar el código Gamma.

Modelo Local Tipo Bernoulli escalonado.

El problema de los modelos tipo Bernoulli reside en que es poco racional suponer que los términos se distribuyen uniformemente en los documentos. Hay documentos que tratan un tema específico, textos en los cuales en un capitulo se menciona mucho a un personaje que luego muere, etc. Si graficamos las posiciones en donde ocurre una determinada palabra en un texto podemos observar algo de la forma:



Este fenómeno se conoce como 'clustering' y aunque parezca extraño ocurre con la gran mayoría de las palabras de cualquier texto. Una forma de adaptar el proceso Bernoulli a este fenómeno es tomar un vector que es en cierta forma una combinación de los códigos gamma y los de Golomb de la forma V=(b,2b,4b,....2^ib) y luego codificar usando códigos de Golomb (o sea, unario y prefijo). Tomando al número "b" como la distancia media de cada entrada del archivo invertido nos aseguramos que al menos la mitad de las distancias caen en el primer bucket del vector. Para este modelo hay que guardar "b" en cada entrada, como "b" es grande para los términos mas raros conviene en realidad guardar N/b usando códigos Gamma.

Modelo Hiperbólico Local.

En este modelo para una distancia x, P(x) se calcula como.

$P(x) = u/x$

siendo

$u = 1/\ln(m+1)+0,5772$

m es la distancia máxima probable (cantidad de términos del texto).

Las distancias se comprimen en función de su probabilidad usando compresión aritmética.

Este modelo representa las probabilidades usando una distribución hiperbólica que ajusta mejor al fenómeno de clustering que el proceso Bernoulli. Sin embargo es el modelo mas difícil de implementar ya que no es posible usar los códigos de Golomb y es necesario aplicar compresión aritmética, la complejidad y lentitud que insume implementar este modelo es causa suficiente para dejarlo de lado aunque teóricamente es muy eficiente.

Modelo Local de Frecuencia Observada.

Este es el mas refinado de todos los modelos, para cada entrada del archivo se calcula la frecuencia de cada distancia y luego en base a dichas frecuencias se comprime cada distancia usando Huffman o bien compresión aritmética. Si bien este modelo es evidentemente el mejor tenemos la desventaja de tener que guardar muchos datos en cada entrada del archivo invertido.

El modelo requiere almacenar la cantidad de distancias y luego las longitudes Huffman de los códigos de cada distancia. Teniendo en cuanta lo que ocupa cada uno de los modelos a guardar en cada una de las entradas este modelo termina siendo parecido en cuanto al nivel de compresión a los códigos gamma y es evidente que aquellos son mucho más sencillos de implementar.

El problema de este modelo es que se tienen que almacenar muchos modelos, para solucionar este problema se recurre a una técnica denominada 'Batching'.

Batching.

La idea de esta técnica es que compartir el mismo modelo para los términos de igual frecuencia, de forma tal que para los N términos de igual frecuencia solo se almacena un modelo. De esta forma se necesitan tantos modelos como frecuencias distintas haya. Una mejora es guardar un modelo no por frecuencias iguales sino por log(ft) iguales. Con lo cual se reduce un poco más la cantidad de modelos a guardar. Otra posibilidad es usar otra base para el logaritmo y una forma muy usada es log 1,618033 (ft) es decir el logaritmo en base al número áureo (aparece en todos lados). Este logaritmo se basa en la serie de Fibonacci por lo cual para las entradas de frecuencia 1 se guarda un modelo, lo mismo para las entradas de frecuencias 2 y 3. Luego se guarda un único modelo para las frecuencias 4 y 5. Otro para las frecuencias 6,7,8 etc... siguiendo la serie de Fibonacci.

Los modelos basados en esta técnica no son demasiado complejos (ya que puede usarse Huffman) y resultan extremadamente eficientes.

Resumen.

A continuación resumimos en una tabla la cantidad de bits por puntero en promedio utilizados por cada uno de los métodos vistos para las colecciones Biblia, GNU y Trec.

método	Bible	GNU	TREC
Unario. (global)	264	920	1719
Binario, Flat Model. (global)	15,00	16,00	20,00
Bernoulli. (global)	9,67	11,65	12,61
Gamma. (global)	6,55	5,69	6,43
Delta. (global)	6,26	5,08	6,19
Frecuencia observada. (global)	5,92	4,83	5,83
Bernoulli. (local)	6,13	6,17	5,73
Hiperbólico. (local)	5,77	5,17	5,74
Bernoulli escalonado. (local)	5,68	4,71	5,28
Batched. (local)	5,61	4,65	5,27

Como puede observarse hay varios modelos que funcionan bastante bien, la elección del modelo a utilizar dependerá del tiempo que se disponga para implementar el método, la forma en que se vaya a utilizar el índice y el grado de ahorro de espacio en disco que quiera utilizarse. En líneas generales nosotros recomendamos el modelo local tipo Bernoulli usando códigos de Golomb, aunque hay muchos otros sistemas que funcionan muy bien con otros métodos.

De esta forma cerramos el tema de compresión de los punteros del archivo invertido, ahora resta analizar la forma en la que se van a almacenar los términos para definir la estructura del índice.

Almacenamiento de los términos (léxico).

En esta sección se analizan diversos métodos para guardar los términos, quienes junto con los punteros conforman el índice invertido, en esta sección suponemos que la forma en la cual se van a almacenar los punteros ya fue decidida y es independiente de la forma en la que se codifiquen las palabras. Para los cálculos estimativos de esta sección omitimos el espacio necesario para guardar los punteros (que es constante pues no depende de como se guarden los strings o términos).

Términos de longitud fija.

Una primera solución consiste en definir una longitud máxima fija para los términos y luego almacenar en el archivo invertido el término, su frecuencia y los punteros (a otra zona del archivo donde se encuentran las distancias de dicho término) de la forma.

TÉRMINO + FRECUENCIA + PUNTEROS.

Ejemplo:

Término	Freq	Punteros
casa	20	xxx
casado	5	xxx
cascada	3	xxx
cascote	1	xxx
casita	2	xxx

Esta solución si bien es la mas simple desperdicia muchísimo espacio en disco ya que la longitud promedio de un término en el índice es de alrededor de 8 bytes (notar que la longitud promedio de una palabra en el texto es menor ya que las palabras cortas son mas frecuentes, en el índice cada palabra se almacena solo una vez por lo que el promedio crece), con lo cual si tenemos una longitud máxima de alrededor de 20 caracteres desperdiciamos en promedio 12 bytes por entrada. El desperdicio en disco es enorme, para la base TREC el espacio necesario para guardar los términos (sin contar los punteros) es de 28Mb.

Concatenación de términos.

Una solución mejor consiste en concatenar todos los términos del índice en una parte del índice y en otra almacenar la frecuencia, un offset a la sección de strings y los punteros. Nuestro ejemplo quedaría de la forma.

STRINGS: casacasadocascadacascotecasita

Frecuencia	Offset al string	Punteros
20	0	xxx
5	4	xxx
3	10	xxx
1	17	xxx
2	24	xxx

La tabla puede guardarse por ejemplo a continuación de los strings (almacenando al principio de los strings la longitud en bytes de los mismos). Con esta técnica necesitamos 4 bytes extras por entrada (para el offset) pero ahorramos los 12 bytes que se desperdiciaban en promedio. Para el mismo índice de la base TREC que ejemplificábamos antes el espacio se reduce de 28 a 20Mb, pero esto aun es mucho, la próxima solución consiste en ahorrar offsets representando en la tabla no todos los términos sino uno de cada "n" términos de la siguiente forma.

La estructura es muy similar a la anterior pero con cambios leves, en primer lugar la tabla contendrá una entrada por cada "n" palabras del texto. A cada término, a su vez hay que pre-concatenarle su longitud (o utilizar algún carácter que no pueda existir en el termino, como el carácter 0 si se guardan palabras). Luego lo que se guarda es: los strings, una tabla de offsets y la tabla de frecuencias y punteros (en donde se mantiene una entrada por término)

Ejemplo usando N=2.

STRINGS: 4casa6casado7cascada7cascote6casita

Frecuencia	Punteros
20	xxx
5	xxx
3	xxx
1	xxx
2	xxx

Offset
0
12
28

Con esta técnica ahorramos un poco mas de espacio ya que la longitud se puede codificar con menos bytes que el offset, y el índice de TREC pasa de 20 a 18 Mb. Sin embargo hay que notar que mientras más espacio ahorramos, más tiempo se perderán luego con las consultas. Esta técnica hace que las búsquedas en el índice sean un tanto más complejas al agregarse el procesamiento secuencia de un bloque de "n" términos. Sin embargo esto no complica realmente mucho la eficiencia de las búsquedas en el índice ya que una búsqueda binaria usando la tabla de offsets y recorriendo secuencialmente cada bloquecito de términos apuntado por un offset no es demasiado más lenta que una búsqueda binaria normal.

Front Coding.

Un detalle muy aprovechable para comprimir los términos es que los mismos se guardan ordenados alfabéticamente y las palabras que son consecutivas unas de otras suelen compartir varios caracteres en común, para aprovechar esta característica se utiliza una técnica denominada Front coding, en la cual lo que se guarda es la cantidad de caracteres que se repiten del término anterior, la cantidad de caracteres nuevos, cuales son dichos caracteres y luego obviamente la frecuencia y los punteros.

Para nuestro ejemplo queda algo de la forma:

Repetidos	Distintos	Chars	Freq	Punteros
0	4	casa	20	xxx
4	2	do	5	xxx
3	4	cada	3	xxx
4	3	ote	1	xxx
3	3	ita	2	xxx

Para guardar "Chars", que es de longitud variable, se puede hacer otra vez concatenación de términos que aunque agrega otra vez el espacio de los punteros, esta vez el léxico es mucho menor si se lo compara con la concatenacion de terminos sin front coding. Al tener los terminos ordenados, la cantidad de caracteres ahorrados en promedio es muy probable que se ahorren mas caracteres por utilizar front coding que lo que se pierde por agregar el campo "distintos" a cada entrada. Sin embargo este metodo tiene una desventaja notablemente seria: ya no es posible realizar búsquedas binarias. Para combinar las ventajas en acceso de la búsqueda binaria con el ahorro de espacio del Front-Coding se utiliza una técnica denominada Front-coding parcial.

Front Coding Parcial.

En un esquema de front coding parcial los términos se almacenan usando front coding con la excepción de que uno de cada "n" términos se almacenan completos. De esta forma vuelve a ser posible la búsqueda binaria. Lo que se hace simplemente es cada "n" términos poner repetidos=0, poner en distintos la longitud del término y en Chars el término completo. Con este método se puede usar una búsqueda binaria muy similar a la que se usaba antes al guardar 1 de 4 offsets, como ya habíamos mencionado el desperdicio en eficiencia es muy poco.

Usando n=4 con Front-Coding parcial el índice de TREC pasa de 18 Mb a 15 Mb.

Todas estas técnicas sirven para almacenar los strings en la forma mas comprimida posible y lograr almacenar los términos en la menor cantidad de espacio posible, los 15Mb a los que llegamos si bien es mucho menos que los 28 Mb originales sigue siendo una cantidad de espacio importante, con todo lo hecho parecería que la única forma de ahorrar aun mas espacio es eliminando completamente los strings y, aunque suene increíble, eso es exactamente lo que vamos a hacer a continuación.

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 podra 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:

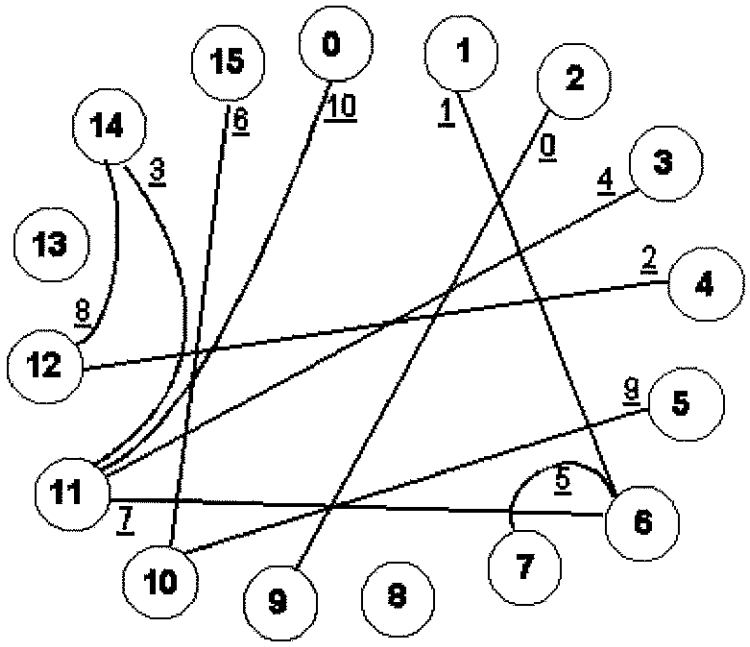
Términos	h1(s)	h2(s)
Arnaldo	2	9
Bartolo	6	1
Celsa	4	12
Edgar	14	11
Emilce	3	11
Hilda	7	6
Humberto	10	15
Olga	6	11
Rene	12	14
Sandro	5	10
Victor	0	11

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)



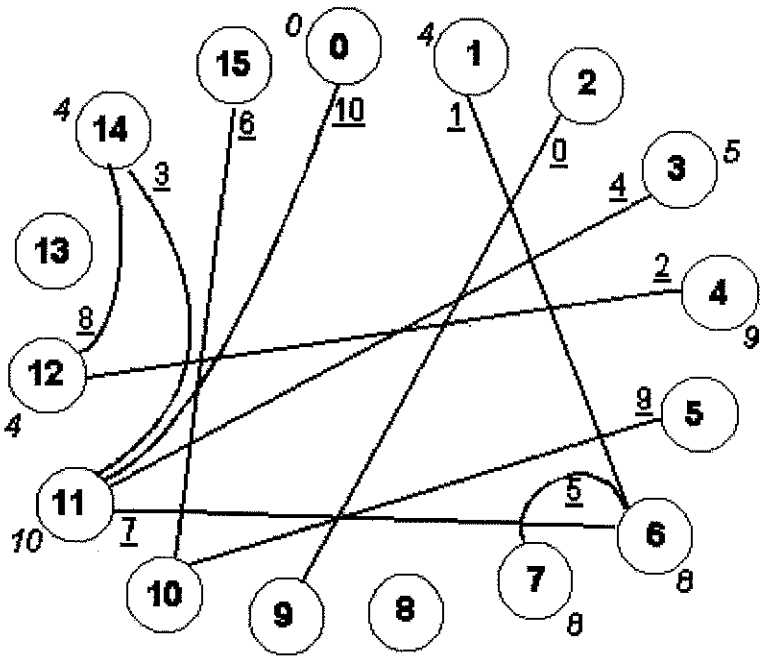
Paso 3: Se comprueba que el grafico es acíclico. Si hubiera quedado cíclico se deberían elegir nuevas funciones h1 y h2, en nuestro caso cambiando los valores de los vectores random

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

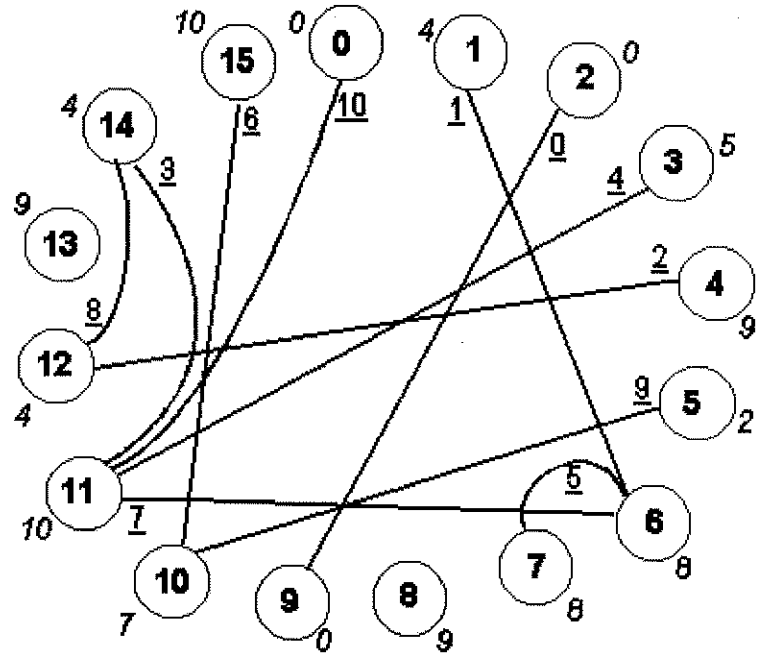
El grafo por ahora esta quedando de la forma:



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 terminos (o sea un n muy grande) la cantidad de grafos promedios a testear quedaria 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

El algoritmo descripto para construcción de funciones de hashing mínimas, perfectas y preservadoras del orden tiene innumerables aplicaciones, puede aplicarse para resolver hash-joins, para construcción de archivos directos etc

Estructura del archivo invertido. (resumen).

La estructura del archivo invertido se define especificando.

Como se comprimen los punteros.
Como se almacenan los términos.

Para la compresión de punteros podemos usar:

Modelo binario plano global.
Código unario.
Código gamma.
Código Delta.
Modelo global tipo Bernoulli.
Modelo local tipo Bernoulli.
Modelo local tipo Bernoulli escalonado.
Modelo global de frecuencia observada.
Modelo local hiperbólico.
Batching.

Para almacenar los términos se pueden aplicar las siguientes técnicas.

Almacenamiento de strings de longitud fija.
Concatenación de términos.
Concatenación de términos con indexación parcial.
Front-Coding.
Front-Coding parcial.
Hashing perfecto.

Una vez que se define la estructura que va a tener un índice invertido resta definir la forma en la cual se va a construir el índice a partir del texto.

Construcción de Índices Invertidos.

Una vez definida la estructura del índice la tarea de construcción del mismo puede parecer trivial, sin embargo como vamos a observar algunas soluciones un tanto simplistas pueden generar un problema de gran envergadura al trabajar con volúmenes de datos muy grandes. En la sección anterior queríamos construir un índice funcional y que ocupe la menor cantidad de espacio posible, ahora vamos a tratar de construirlo en forma veloz!

Inversión por transposición de matrices.

Aunque el nombre suene complejo este es el método trivial para invertir un texto. Supongamos que tenemos el texto:

Pedro y Pablo. Pedro corre. Pablo respira. Pedro corre y respira. Pedro corre Pedro.

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 fase se construye una matriz a medida que se va leyendo el texto de la forma.

	Pedro	y	Pablo	corre	respira
1	1	1	1	0	0
2	1	0	0	1	0
3	0	0	1	0	1
4	1	1	0	1	1
5	2	0	0	1	0

Para invertir el texto basta con trasponer la matriz.

	1	2	3	4	5
Pedro	1	1	0	1	2
Y	1	0	0	1	0
Pablo	1	0	1	0	0
corre	0	1	0	1	1
Respira	0	0	1	1	0

De la matriz traspuesta se obtiene en forma automática el índice invertido. Esta solución funciona y su implementación es trivial, sin embargo no es muy eficiente.

Suponiendo que la matriz se almacena en memoria para el texto Biblia esta solución insume 4x9020x31102 bytes (4 bytes para almacenar cada número, 9020 términos y 31102 documentos) aproximadamente un Giga-byte. Para TREC requeriría 1.4 tera-bytes. Abandonemos, pues, la idea de tener la matriz en memoria y guardemos la matriz en disco. en este caso y suponiendo que el espacio no es problema (lo cual es harto dudoso) tenemos un pequeño problema de tiempos, para el texto Biblia necesitaríamos alrededor de un mes para generar el índice, mientras que para TREC se

necesitarían 127 años. Con esta técnica los tiempos de procesamiento manual de la edad media parecen ser mejores. Como entenderán esta solución queda descartada.

Inversión usando listas enlazadas.

A esta solución se la conoce también como solución del estudiante, ya que la mayoría de los estudiantes de computación con conocimientos mínimos de estructuras de datos plantean este tipo de solución cuando tienen que invertir un archivo de datos.

En este esquema se utiliza una tabla de hashing o un árbol n-ario para ir almacenando los términos a medida que se lee el texto, asociado a cada término hay una lista enlazada donde se almacenan los números de los documentos en donde se observo dicho término. La implementación de este método es muy sencilla y con solamente describir el algoritmo es evidente que el mismo es mejor solución que la utilización de inmensas matrices.

Hay varias formas de implementar este algoritmo, pero lo que nos interesa es analizar su rendimiento. En cuanto a tiempos es muy eficiente, para Biblia por ejemplo necesita aproximadamente 30 segundos para construir el índice.

Lamentablemente este método consume realmente muchísima memoria, la cantidad de punteros que se utilizan escapa a la imaginación de cualquier estudiante, para Biblia el algoritmo necesita 4 giga-bytes de memoria, no parece ser razonable.

La única solución que evita el consumo casi apocalíptico de memoria de este método es guardar las listas enlazadas en disco, con lo cual y como podrán imaginarse empezamos nuevamente a tener problemas de tiempos. Para Biblia el método insumiría unas 4 horas (aceptable) mientras que para TREC se necesitarían alrededor de seis semanas. Seis semanas es mucho menos que 127 años pero lamentablemente para los estudiantes que pensaron en este algoritmo no es un tiempo que podamos aceptar.

De pronto el tema parece ser mas interesante, luego de analizar dos soluciones (una muy ineficiente a simple vista y otra que parecía razonable) nos encontramos con que nuestro mejor algoritmo hasta el momento tarda una cierta cantidad de semanas en generar el índice invertido, afortunadamente existen otras soluciones.

Inversión por sort.

La cantidad de información que se maneja en el proceso de inversión es tan grande que resulta imposible realizar todo el proceso en memoria, los métodos anteriores eran eficientes si se hacían en memoria pero se volvían lentísimos al implementarse sobre disco debido a que realizaban muchísimos accesos al disco. La idea de la inversión por sort es realizar la mayoría de los accesos al disco en forma secuencial de forma tal de reducir el tiempo insumido accediendo al disco.

La inversión por sort consta de tres fases.

- Extracción del léxico y construcción del archivo auxiliar.
- Sort.
- Construcción del índice invertido.

Fase1: Extracción del léxico y construcción del archivo auxiliar.

En esta etapa el archivo de texto es procesado secuencialmente, a medida que se lee el archivo de texto los términos se extraen y se almacenan en el léxico de acuerdo a la forma que se haya elegido (front-coding por ejemplo) a cada término se le asigna un número de término. Como este numero de termino se utilizará posteriormente para un Sort, será necesario ordenar el léxico así el archivo invertido final quedara también ordenado. A medida que se procesan los términos se

escribe un archivo auxiliar formado por triplas de la forma <t,d,fd> donde 't' es el número de término , 'd' es el número de documento y 'fd' es la frecuencia de 't' en el documento 'd'.

Fase2: Sort.

En esta etapa se procede a ordenar el archivo auxiliar construido en la primera etapa de acuerdo a 't' y 'd'. Este proceso de ordenamiento es un sort externo con características bien definidas, el archivo a ordenar es muy grande y las claves son dos números de unos 3 o 4 bytes cada uno (claves chicas). Como el proceso de sort es crítico para este algoritmo de inversión es conveniente analizar que tipo de sort externo realizar. , los algoritmos competitivos de sort-externo son:

Natural selection.
Replacement selection.
Key-Sort.
Postman-sort.

Muchos sistemas que utilizan inversión por sort prestan poca atención al mecanismo de sort a utilizar lo cual constituye un grave error ya que el etapa mas costosa (en tiempo y espacio) de todo el método, en este apunte vamos a analizar las cuatro alternativas de sort posibles (descartando el sort interno por ser siempre inferior al replacement selection en volúmenes de datos abundantes) evaluando cual es la técnica mas apropiada. Un sort por Natural Selection tampoco parece buena alternativa ya que de todos los métodos es el único que requiere de espacio extra en disco para generar las particiones, si consideramos el tamaño del archivo(s) de texto mas el espacio extra necesario para el archivo auxiliar mas el espacio del índice es evidente que no podemos darnos el lujo de además reservarle una parte del disco al sort.

Queda elegir entre un replacement selection y un key-sort, el key-sort usualmente no es una buena opción para un sort externo, pero aquí adquiere un protagonismo inusual ya que las claves son realmente muy chicas, cuando las claves son muy chicas un sort de este tipo podría llegar a ser conveniente. Lo malo del key-sort es que luego de levantar las claves en memoria y ordenarlas realiza accesos al disco en forma random para ordenar el archivo, como dijimos que la idea de este proceso de inversión era minimizar los accesos al disco tenemos que descartar el algoritmo de key-sort inmediatamente.

Por lo tanto el algoritmo elegido para la etapa de sort es el Replacement selection. Para el merge final se utilizará optimal merge

Fase 3: Construcción del índice invertido.

En esta etapa a partir del léxico (obtenido en la fase 1) y del archivo auxiliar ordenado se construye el archivo invertido. Para ello hay que recorrer el archivo auxiliar ordenado (en forma secuencial) hasta que el número de término cambie, cuando ello ocurre se escribe la entrada correspondiente al término en el archivo invertido que consistirá en el término (o su equivalente según el método que se use), la frecuencia del mismo (suma de todas las frecuencias leídas del archivo auxiliar) y los documentos en donde apareció el texto. Una vez generadas todas las entradas puede eliminarse el archivo auxiliar. Luego de esto puede ser necesario regenerar el índice si el método elegido para guardar los punteros exige una segunda pasada. Con este método la generación del índice pasa de seis semanas a aproximadamente veinte horas. Por ejemplo:

Pedro y Pablo. Pedro corre. Pablo respira. Pedro corre y respira. Pedro corre Pedro

El léxico es de la forma:

Pedro
y
Pablo
corre
respira

Que ordenado queda:

corre
Pablo
Pedro
respira
y

El archivo auxiliar que se genera en la fase uno es de la forma:

No-Termino	Documento	Freq
3	1	1
5	1	1
2	1	1
3	2	1
1	2	1
2	3	1
4	3	1
3	4	1
1	4	1
5	4	1
4	4	1
3	5	2
1	5	1

Ordenando:

No-Termino	Documento	Freq
1	2	1
1	4	1
1	5	1
2	1	1
2	3	1
3	1	1
3	2	1
3	4	1
3	5	2
4	3	1
4	4	1
5	1	1
5	4	1

De aquí podemos generar el índice invertido en forma directa.

Termino	Freq	Docs
corre	3	2,4,5
Pablo	2	1,3
Pedro	5	1,2,4,5
respira	2	3,4
y	2	1,4

Como vimos, el proceso de inversión de un archivo muy grande es un proceso muy costoso y que insume varias horas de trabajo, afortunadamente este suele ser un proceso que se realiza solo una vez, el mejor algoritmo de inversión consiste en averiguar si el proceso puede hacerse en memoria, en cuyo caso se utiliza el algoritmo del estudiante, en caso contrario si se tiene que hacer en disco se usa la inversión por sort.

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 correspondiente al documento, que surge de los términos que lo integran. Uno de los parámetros a definir es el tamaño del signature, que puede ser de 32 bits, 64 bits, 128 bits etc.

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 "había una vez una vaca fea", 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.

había =	0100 0000 1000 0100
una =	0001 0000 0000 1000
vez =	0011 0000 0000 0010
una =	0000 0101 0010 0000
vaca =	0100 0000 0000 0000
fea =	0001 0001 0000 1000

Como se ve en los términos "una", "vez" y "vaca" 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 0101 1010 1110 (notar que es innecesario hashear mas de una vez un mismo término, como "una" 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.

Otras consultas con conectores lógicos AND y OR son particularmente complejas con signature-files ya que la lógica a utilizar debe ser ternaria (Si, No, Puede ser) el problema surge cuando la mayoría de los documentos quedan caratulados como 'puede-ser' y hay que buscar el o los términos dentro del texto.

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 que son:

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 recuperamos los slices: 00110, 11111 y 00110. Con esto lo que tenemos son los bits 0, 1 y 3 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 2 y 3 (los dos bits en uno). De esta forma solo se hacen solamente tres accesos al disco (uno por cada función de hashing).

Los signature-files implementados usando bit-slices son una opción interesante para construir índices al ocupar poco espacio y poder manejarse en forma eficiente usando bit-slices. Una combinación de signature-files y archivos invertidos puede generar un ambiente muy propicio para resolver diversas consultas. La utilización de signature-files como único método de indexación de archivos de texto es dudosa ya que la resolución de consultas se haría muy trabajosa e ineficiente, sistemas implementados en base a signature-files son famosos por tardar horas en la resolución de algunas consultas simples.

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

Bitmaps.

Un bitmap es una forma de índice muy particular, para construir un bitmap que sirva para un archivo de texto tenemos que tener una entrada por término (igual que en un archivo invertido), pero lo que se guarda para cada término es un vector de N bits (siendo N la cantidad de documentos del texto) donde cada bit es 0 o 1 según se encuentre o no el término en el documento. Por ejemplo tomando cada línea como documento y cada palabra como termino:

Pedro y Pablo.
Pedro corre.
Pablo respira.
Pedro corre y respira.
Pedro corre Pedro.

El bitmap correspondiente es:

Termino	1	2	3	4	5
pedro	1	1	0	1	1
y	1	0	0	1	0
pablo	1	0	1	0	0
corre	0	1	0	1	1
respira	0	0	1	1	0

Los bitmaps son excepcionalmente cómodos cuando se los utiliza para resolver consultas, por ejemplo para las consultas booleanas basta con operar haciendo Ors y ANDs de los bitmaps de los términos buscados para obtener los documentos que son resultado de la consulta. El problema es que son un tanto extravagantes en cuanto al espacio que consumen, para TREC por ejemplo un bitmap ocuparía unos 40 giga-bytes, aproximadamente 20 veces mas que el texto que indexa. Por ello si se usan bitmaps debe usárselos comprimidos.

Compresión de bitmaps.

Cada entrada en un índice del tipo bitmap tiene la particular característica de estar formada por algunos unos y una enorme cantidad de ceros, para comprimir este tipo de información puede usarse un árbol de derivación binaria.

Árboles de derivación binaria.

Supongamos que tenemos la siguiente tira en binario. (64 bits).

0000 0010 0000 0011 1000 0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000

La técnica de compresión por derivación lo que hace es ir armando un árbol en el cual el bit-vector se procesa en cada paso tomando N bits. N bloques de N bits forman un nodo de N bits en el cual cada bit indica si en el hijo que le corresponde hay o no algún uno. Por ejemplo, si N=4 la primera fase es:

0101 1010 0000 0000

0000 0010 0000 0011 1000 0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000

Una aclaración es que si en cualquier momento la cantidad de bits de un nivel no es múltiplo de N, el último nodo del nivel superior no tendrá N bits. Lo que se debe hacer en este caso es completar dicho nodo con ceros. Hecha la aclaración continuamos comprimiendo el nivel superior del árbol hasta que el nuevo nivel tenga 1 solo nodo, que es justamente lo que ocurrirá:

1100

0101 1010 0000 0000

0000 0010 0000 0011 1000 0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000

Una vez que se llega a la raíz, lo que se almacena son todos los niveles del árbol uno atrás de otro pero obviando los nodos que son todos ceros. Esto es porque cuando un nodo esta compuesto únicamente por ceros, entonces el bit correspondiente a dicho nodo en el nivel superior será un cero. Por ende, cuando el descompresor lea ese bit sabrá que ese nodo esta compuesto únicamente de ceros y puede reconstruirlo. Volviendo al ejemplo entonces, lo que se guardará es:

1100, 0101, 1010, 0010,0011,1000,0100 (7x4=28 bits)

Para descomprimir se procede de la siguiente forma: el primer elemento de 4 bits es 1100 lo cual implica que en el nivel inferior los dos hijos de la izquierda tienen algún uno mientras que los dos hijos de la derecha están en cero. Los próximos ocho bits (2 nodos) corresponden entonces a los dos hijos izquierdos.

1100

0101 1010 0000 0000

Luego se repite lo mismo que antes. Se sabe que el primer grupo de 4 bits es cero, luego hay un grupo que tiene unos que se debe leer y es 0010, luego un grupo de cuatro ceros, luego el 0011 y luego el 1000 luego cuatro ceros, el 0100 y después 9 grupos de cuatro ceros.

En el ejemplo se agrupaba un bit cada cuatro, en las aplicaciones reales suele usarse un factor de 8 que comprime mas y es mas fácil de implementar ya que cada byte representa 8 bytes y así sucesivamente.

En este caso comprimimos 64 bits en 28, en general el nivel de compresión de este método es de alrededor del 75%, con lo cual nuestro bitmap gigante pasa a ocupar unos 5 giga-bytes, como puede verse es mucho mas de lo que ocupa el archivo invertido y al comprimir el bitmap ya no es tan fácil operar con el para realizar consultas, Los sistemas de manejo de textos en general no suelen utilizar bitmaps debido al enorme costo que insume almacenarlos.

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.

Stemming.

Una técnica usada al construir algunos índices invertidos es el stemming, proceso por el cual todos los términos de un texto antes de agregarse al léxico son reducidos a una raíz común para todas las palabras de una determinada familia. Por ejemplo compresión, comprimido, compresor y comprimir son palabras con una raíz en común.

El proceso de stemming tiene dos ventajas al usarse para la construcción de un índice, en primer lugar se reduce mucho el espacio ocupado por el índice al disminuir notablemente la cantidad de términos del texto (que ahora mas que términos son raíces de términos). En segundo lugar consultas que busquen la palabra 'compresión' van a originar como resultado documentos que contengan la palabra buscada y otros que contengan palabras de la familia de la palabra buscada. La principal desventaja del método es que no es posible obtener aquellos documentos que contengan exactamente una determinada palabra sin realizar una segunda pasada secuencial para descartar aquellos que en realidad tenían palabras de la familia.

La técnica de stemming se usa mucho en sistemas de recuperación total de textos, en algunos casos se considera la posibilidad de una búsqueda 'exacta' opcional en la cual se realiza un filtrado de los documentos devueltos por la consulta. En general el poder obtener documentos que tengan palabras relacionadas con un determinado tema, es considerado una opción muy aceptada por los usuarios, muchas veces el proceso de stemming origina comentarios del tipo 'el sistema es inteligente'.

Los algoritmos de stemming son particularmente complejos, muchos sistemas de manejo de textos usan un módulo o biblioteca de stemming adquirido a alguna compañía que se dedica a tal cosa.

Stop-Words.

Las stop-words son palabras que si bien resultan muy frecuentes en un texto no son muy utilizadas para realizar consultas (el, lo, la, y, de, con, etc). Una posibilidad al construir el índice consiste en eliminar del índice a las stop-words. Con este proceso se ahorra mucho espacio en el índice pero al desaparecer palabras se pierde fidelidad en las consultas. Salvo que se necesite un ahorro de espacio notable esta técnica es poco recomendable.

Case-Folding.

Case-Folding es un método muy sencillo para ahorrar espacio en el índice. Consiste, simplemente, en convertir todos los términos a minúsculas, con lo cual se reduce en un buen número la cantidad de términos en el léxico con el consiguiente ahorro de espacio en el índice. La gran mayoría de los sistemas de manejo de textos usan case-folding (por no decir todos), en muchos se agrega a la consulta la opción de 'case-sensitive' en cuyo caso se buscan los documentos 'candidatos' a partir del índice y luego se busca dentro de los documentos para verificar si la palabra coincide con la buscada diferenciando, ahora si, mayúsculas de minúsculas.

Inversión por sort: Compresión del archivo auxiliar.

Como bien lo dice el titulo, esta optimizacion sirve exclusivamente para los indices invertidos cuando se decide generarlos utilizando el metodo de inversion por sort.

Al realizar el proceso de inversión de un archivo de texto utilizando sort externo uno de los problemas que se presentan es la necesidad de mucho espacio en disco para la creación del archivo auxiliar, esto se puede solucionar en cierta forma comprimiendo el archivo auxiliar a medida que se va generando. En general con un buen algoritmo de compresión el espacio necesario para el archivo auxiliar puede reducirse en mas de un 70%.

Resolución de Consultas.

La razón por la cual se construye un índice para un archivo de texto es para facilitar la resolución de consultas, en esta sección vamos a analizar distintos tipos de consultas y como resolverlas en forma eficiente usando un índice invertido, como verán, las consultas que los usuarios quieren hacer muchas veces no pueden ser previstas por los diseñadores del sistema por lo que para algunas consultas especiales puede que sea necesario modificar la forma en la que construimos el índice.

Consultas Booleanas.

Una consulta booleana es aquella donde el usuario pregunta cual o cuales son los documentos que matchean una determinada consulta, la respuesta consiste en los números de documento que satisfacen la consulta o bien un mensaje de 'no-hay documentos'. A este tipo de consulta se la llama consulta booleana no por la forma en la cual se enuncia (usando conectores lógicos) sino por la respuesta que se obtiene, no existen respuestas del estilo 'los documentos que mas se acercan a la consulta son...', la respuesta es totalmente determinística.

Consultas puntuales.

Una consulta puntual consiste en buscar una determinada palabra en un texto. Por ejemplo 'perro', la respuesta consiste en los documentos que contienen tal palabra o un mensaje que indique que no pudo resolverse la consulta.

Usando un índice invertido este tipo de consulta se resuelve en forma sencilla y eficiente con una búsqueda binaria de la palabra en el índice. Si se usa hashing basta con solamente hashear la palabra y recuperar los punteros de la entrada correspondiente, luego hay que verificar si la palabra existe o no en alguno de los documentos apuntados (una palabra que no pertenecía al léxico puede causar una colisión con una palabra ya existente puesto que la función solo es perfecta para las palabras del léxico), para lo cual se la buscará en el documento más corto para que el tiempo de búsqueda sea mínimo. Este tipo de consulta no plantea mayores inconvenientes.

Consultas conjuntivas.

Una consulta conjuntiva es de la forma:

Palabra1 AND Palabra2 AND .... PalabraN.

Es decir que se buscan aquellos documentos que contienen a un grupo de palabras indicado por el usuario.

Para resolver esta consulta el algoritmo mas eficiente consiste es recuperar los punteros de las palabras en orden de frecuencia ascendente (empezando por los punteros de la palabra menos frecuente), luego se buscan los punteros de la siguiente palabra y se eliminan de la primera lista aquellos punteros que no figuran en la segunda lista. Cuando no quedan mas palabras la lista de punteros contiene los números de aquellos documentos que matchean la consulta.

La razón por la cual se comienza con la palabra menos frecuente es que dicha palabra es la que menos punteros suele tener con lo cual siempre trabajamos con la menor cantidad de punteros posibles (la lista por ejemplo puede quedar vacía restando procesar muchas palabras cada una de

ellas con muchos punteros). De esta forma se minimiza el tiempo y la cantidad de memoria necesarios para resolver la consulta.

Consultas disyuntivas.

Una consulta disyuntiva es de la forma.

Palabra1 OR Palabra2 OR ... OR PalabraN

Es decir que se buscan los documentos que contengan alguna de las palabras indicadas en la consulta.

Aquí el orden en el cual se procesan las palabras es poco relevante ya que la cantidad de punteros a manejar como máximo será la misma independientemente del orden utilizado. El algoritmo es sencillo, se toma la primer palabra, se recuperan los punteros, luego la segunda, se hace una unión de los dos conjuntos de punteros, luego la tercera etc. El resultado final es una lista con todos los números de documentos en donde se encontraron alguna de las palabras mencionadas.

Consultas compuestas.

Una consulta compuesta es una combinación de consultas conjuntivas y disyuntivas de la forma.

(Palabra1 OR Palabra2) AND (Palabra3 OR Palabra4) ... etc...

La resolución de estas consultas se hace resolviendo los paréntesis en el orden indicado y realizando la consulta correspondiente en cada nivel. Si en un mismo nivel de paréntesis hay que resolver una consulta disyuntiva y otra conjuntiva y no importa el orden en que se hagan se elige siempre primero la consulta conjuntiva ya que es la que tiene mas chances de dar como resultado cero documentos con lo cual se simplifica la evaluación de la expresión. Con el esquema de archivo + índice invertido este tipo de consultas no plantea problemas.

Wildcards.

Una consulta booleana puede permitir buscar palabras usando comodines como por ejemplo:

perr*

En cuyo caso se piden aquellos documentos que contienen palabras que comienzan con perr*. Este tipo de consulta elimina de nuestros algoritmos la posibilidad de hacer una búsqueda binaria, por lo que se deberá utilizar otro método para resolver la consulta

Búsqueda por Fuerza Bruta.

La búsqueda por fuerza bruta es la forma mas sencilla de resolver una consulta con wildcards, simplemente se recorre todo el léxico del índice en búsqueda de palabras que matcheen con el patrón buscado. Notemos que este método no puede utilizarse si para almacenar el léxico se utilizó una función de hashing, esta es una de las desventajas de usar hashing para almacenar los términos.

El problema del algoritmo de búsqueda por fuerza bruta es que es lento, pero sin alterar la estructura del índice es la única solución viable. A continuación estudiamos dos formas de modificar el archivo índice para optimizar las consultas con wildcards.

N-Gramas.

La primera solución a estudiar implica no almacenar palabra en el índice sino N-gramas, cada palabra es dividida en una cantidad N de caracteres llamados n-gramas. Por ejemplo la palabra "sacar" se divide en los digramas s,sa,ac,ca,ar,r (el $ es un caracter especial de comienzo-fin de término). Cada entrada del índice invertido corresponde a un N-grama, lo que se almacena son los números de documentos en donde aparece cada N-grama. En general el índice de N-gramas apunta a su vez al léxico del archivo invertido.

De esta forma si queremos buscar algo de la forma "sac*r" podemos hacer una consulta booleana conjuntiva de:

$s AND sa AND ac AND r$

A los resultados de esta consulta se los deberá examinar para ver que realmente correspondan con la consulta ya que pueden haber matcheado términos que no concuerdan con lo buscado. Siguiendo el ejemplo anterior, la palabra "satisfacer" sería devuelta por la consulta conjuntiva ya que se representa con los digramas $s, sa, ac y r$ (entre otros) pero definitivamente no corresponde con sac*r por lo que deberá ser descartada.

Finalmente, con los términos obtenidos luego de filtrar los que no corresponden se obtendrán (por medio de una disyuntiva) aquellos documentos donde aparecen y ese será el resultado de la consulta

La utilización de N-gramas implica necesitar alrededor de un 50% de espacio adicional al del archivo invertido. Por ejemplo, la utilización de tri-gramas para TREC necesitó de un 55% de espacio adicional en disco y redujo el tiempo promedio necesario para una consulta con wildcards de 6 segundos (fuerza bruta) a solo 0.05 segundos. La utilización de n-gramas debe decidirse en función de la relación espacio-tiempo con la que se trabaje.

Léxico rotado.

Otra solución posible para resolver consultas con wildcards que insume aun mas espacio que el uso de n-gramas es el uso de léxico rotados. Para la palabra labor, lo que se guardaría es

$labor,abor$l,borla,orlab,r$labo

Cada rotación es una entrada en el índice invertido. Una consulta de tipo labo*r implicaría rotar el string hasta dejar el asterisco al final y luego buscar en el índice todas las entradas cuyos primeros caracteres sean "r$labo".

Consultas con dos wildcards como por ejemplo "*abo*" pueden resolverse fácilmente, ya que al rotar quedando los dos asteriscos al final solo se buscan rotaciones que empiecen con: "abo" (o bien $abo según como se interprete el asterisco inicial). En cambio no se pueden resolver consultas del estilo "ar*le*", o con mas de dos wildcards.

Usando un léxico rotado la misma consulta para TREC que antes insumía 0,05 segundos tarda ahora 0,006 segundos pero el archivo índice requiere un 250% mas de espacio que el invertido simple.

Consultas Ranquedas.

En una consulta ranqueada el usuario ingresa una lista de palabras y el sistema responde con los N documentos mas relevantes para la lista de palabras indicadas aun cuando algunas palabras no figuren en ningún documento. Por ejemplo, "indice,inversion,compresion,consulta" es una buena lista de palabras para que este apunte figure al tope del ranking de documentos relevantes.

Este tipo de consulta es muy cómoda para el usuario ya que con una lista de palabras puede definirse el estilo de los documentos que se quieren buscar, refinando la lista a buscar y consultando los documentos que se devuelven en las consultas puede tenerse un acceso muy cómodo y practico a la información almacenada. Un buen ejemplo de este tipo de consulta lo constituyen los buscadores de información en la Web, dada una lista de palabras es muy atractivo poder obtener una lista con los sites mas relevantes para la lista de palabras consultada.

Estudiaremos a continuación basándonos en este ejemplo métodos para resolver consultas ranqueadas de forma tal que la respuesta que recibe el usuario sea la que el usuario espera recibir (lo cual nunca es fácil). Tomando como documentos a cada línea y como términos a cada palabra, tenemos cinco términos (Alberto, Bartolo, Cesar, Demian y Ernesto, que llamaremos a,b,c,d y e) distribuidos en 5 documentos.

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.

Esta aproximación es la mas sencilla y puede resolverse usando un índice invertido tradicional, la forma en la cual se calcula el puntaje de cada documento es contando simplemente cuantas de las palabras de la consulta contiene el documento. 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 termino 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 termino "Rolo" no está en ningún documento

Esta técnica simple presenta tres inconvenientes.

- En primer lugar no tiene en cuenta la frecuencia de los términos, si un término aparece cuatro veces o una vez en un documento este esquema lo considera igual poniendo un uno en la posición correspondiente del vector. 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
- No tiene en cuenta que algunos términos son mas importantes que otros. 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.
- Por último los documentos mas largos son favorecidos por este esquema, ya que al tener mayor longitud tienen mayor cantidad de términos y son mas propensos a matchear una mayor cantidad de términos de la consulta.

Veremos a continuación como solucionar estos inconvenientes

Producto Interno.

Para resolver el primer problema sería bueno que en el archivo invertido en la entrada de cada término se indique no solo los números de documentos donde dicho término aparece sino también la cantidad de veces que aparece dicho término (frecuencia del término en el documento)

Un entrada del archivo invertido para el texto del ejemplo 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 en lugar de unos y ceros la cantidad de veces que aparece cada 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 puntajes son:

Documento	Puntaje
1	3
2	3
3	1
4	4
5	2

El documento mas relevante es ahora el 4 pero esto ocurre simplemente porque el mismo contiene muchas palabras y, como todavía se favorecen a los documentos largos, logra obtener mas puntaje que el resto. Sin embargo, el primer problema se pudo solucionar y se ve que el documento 2 ahora tiene mas puntaje que el 5.

Producto Interno Mejorado

En el año 1949 George Zipf publicó un libro de gran éxito llamado 'Principio del esfuerzo mínimo', en el cual afirmaba tras estudiar numerosas distribuciones que la importancia de un objeto era inversamente proporcional a su frecuencia. Cuanto mas frecuente o abundante era una cierta cosa menos importancia tenía. La ley que enunció denominada ley de Zipf mide la importancia de un determinado objeto como:

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

Siendo N la cantidad de objetos del universo y ft la frecuencia del objeto 't' dentro del universo. Si se aplica la ley de Zipf a nuestro problema en lugar de asignarle a cada término en el vector de documentos un número de acuerdo a la cantidad de veces que ocurre asignémosle un número que exprese la importancia del término multiplicada por la cantidad de veces que ocurre. A este número lo llamaremos wtd que se calculará como

$$wtd = ftd * wt$$

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

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)

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 termino 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 terminos 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.

Hasta el momento calculamos el puntaje de un documento como el producto interno entre el vector del documento D y el vector consultas Q. Como el calculo del puntaje de cada documento se hace en función de dos vectores pasemos a ver todo el problema desde el punto de vista algebraico.

Modelos de espacios vectoriales – método del coseno

Dados dos vectores (el de la consulta y el de un documento x) podemos calcular el puntaje del documento con la distancia euclidiana.

$$P(Q,D) = \sqrt{\sum |wqi * wdi|^2}$$

Esta medida en realidad indica cuan distintos son los vectores por lo que para el puntaje lo correcto sería calcular el recíproco. La distancia euclidiana sufre del problema inverso al método del producto interno, tiende a perjudicar a los documentos largos ya que el vector consulta es en general de módulo chico.

Lo que realmente interesa es la dirección de los vectores y no su longitud, es decir que debemos medir la distancia en dirección entre los dos vectores. La distancia en dirección es un concepto geométrico sencillo, se limita simplemente a medir el ángulo entre los dos vectores. De la propiedad.

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

Podemos despejar el coseno del ángulo como:

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

$$\cos\phi = \frac{\sum Xi * Yi}{\sqrt{\sum Xi^{1}} * \sqrt{\sum Yi^{2}}}$$

Cuanto mayor es el coseno menor es el ángulo así que más parecida es la dirección de los vectores, este método de denomina 'Medición del Coseno'. Siendo Q el vector consulta y W el vector documento el puntaje (coseno) que obtiene el documento es:

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

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

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

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

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

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

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 termino 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 termino 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 termino 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).