75.06 Organización de datos: Temas primer parcial

Gonzalo Agustín Beviglia

17 de octubre de 2012

Índice

1. Teoria de la información **3**
1.1. Fuente . . . 3
1.2. Mensaje . . . 3
1.3. Código . . . 3
1.4. Clasificación de fuentes . . . 3
1.4.1. Según la naturaleza de los mensajes . . . 3
1.4.2. Según la relación existente entre mensajes . . . 3
1.5. Información . . . 3
1.6. Entropía de una fuente . . . 3

2. Compresión de datos **4**
2.1. Códigos . . . 4
2.2. Teorema fundamental de la compresión de datos . . . 4
2.3. Compresión estadística . . . 4

3. Métodos de compresión **5**
3.1. Huffman estático . . . 5
3.2. Huffman dinámico . . . 5
3.3. Métodos semi-estáticos . . . 5
3.4. Métodos con contexto de aparición . . . 5
3.4.1. Huffman estático de orden 1 . . . 5
3.4.2. Huffman dinámico de orden 1 . . . 6
3.5. Métodos no-estadísticos de compresión . . . 6
3.5.1. Run-Length-Encoding (RLE) . . . 6
3.5.2. Variante RLE(1) . . . 6
3.5.3. Variante RLE(2) - Pack bits . . . 6
3.5.4. LZ77 . . . 6
3.5.5. LZ78/LZW . . . 6
3.5.6. LZHuff/Deflate . . . 7
3.5.7. LZP (Predictor) . . . 7
3.6. Efecto caché . . . 7
3.7. Problema de los métodos estadísticos . . . 7

4. Compresión aritmética **8**
4.1. Compresión aritmética dinámica . . . 8
4.2. Implementación . . . 8
4.3. Modelos de orden superior . . . 8
4.4. Principio de localidad . . . 8
4.4.1. MTF (Move to front) . . . 9
4.5. Modelo de Shannon . . . 10
4.6. Modelo estructurado . . . 10

5. Archivos directos **11**
5.1. Resolución de colisiones . . . 11
5.1.1. Método lineal . . . 11
5.1.2. Método cuadrático . . . 11
5.1.3. Doble hashing . . . 11
5.1.4. Área de overflow independiente . . . 11
5.1.5. Área de overflow distribuida . . . 11
5.1.6. Encadenamiento de colisiones . . . 12
5.1.7. Buckets . . . 12
5.1.8. Método del Cuckoo . . . 12
5.1.9. N funciones de hashing . . . 12
5.1.10. Grafos dirigidos . . . 12
5.2. Construcción de funciones de hashing . . . 12
5.2.1. Fold and add . . . 12
5.2.2. Modelo multiplicativo . . . 13
5.2.3. Pearson . . . 13
5.2.4. Hashing deslizante . . . 13
5.2.5. Hashing perfecto . . . 13
5.2.6. Hashing extensible . . . 14

6. Índices invertidos **15**
6.1. Almacenamiento de punteros . . . 15
6.1.1. Código unario . . . 15
6.1.2. Código gamma . . . 15
6.1.3. Código delta . . . 15
6.1.4. Modelo global de tipo Bernoulli . . . 15
6.1.5. Códigos de Golomb . . . 16
6.1.6. Modelo local de tipo Bernoulli . . . 16
6.1.7. Modelo local de frecuencia observada . . . 16
6.1.8. Modelo de bytes de longitud variable . . . 16
6.2. Almacenamiento del léxico . . . 16
6.2.1. No realizar acción . . . 16
6.2.2. Concatenar todos los términos . . . 17
6.2.3. Front coding . . . 17
6.3. Resolución de consultas . . . 17
6.3.1. Consultas con distancias entre términos . . . 17
6.3.2. Consultas con comodines . . . 18

7. Signature files **19**
7.1. Resolución de consultas . . . 19
7.2. Bit slices . . . 19
7.2.1. Resolución de consultas . . . 19
7.3. Bitmaps . . . 20
7.4. Compresión de archivos con árboles de derivación binaria . . . 20

8. Page rank **21**

9. Evaluación de consultas **22**
9.1. Precisión . . . 22
9.2. Recall . . . 22

1. Teoria de la información

1.1. Fuente

Todo aquello que emite mensajes. Conjunto finito de mensajes.

1.2. Mensaje

Combinación de elementos de un lenguaje.

1.3. Código

Combinación de elementos de un alfabeto "B" de forma tal que si "m" es un mensaje del alfabeto "A":

$$f(m) = c, m \in A, c \in B$$

A "f" se la llama función de codificación.

1.4. Clasificación de fuentes

1.4.1. Según la naturaleza de los mensajes

- Determinística: se puede determinar el próximo mensaje a aparecer.
- Aleatoria: no hay certeza absoluta de que aparezca un mensaje en particular.

1.4.2. Según la relación existente entre mensajes

- Estructurada: los mensajes tienen cierta relación entre ellos.
- No estructurada o caótica: los mensajes no guardan relación entre sí.

1.5. Información

- En una fuente aleatoria y estructurada, no todos los mensajes tienen la misma probabilidad de ocurrencia.
- La cantidad de información de una fuente está relacionada con la probabilidad de ocurrencia de sus mensajes.
- Los mensajes más probables aportan menos información que los mensajes menos probables.
- Las fuentes aleatorias no estructuradas se suelen denominar de información pura.
- Todo lo que no sea información pura es redundante, y todas las fuentes (aleatorias estructuradas) tienen un cierto nivel de redundancia.

1.6. Entropía de una fuente

Sea P_i la probabilidad de ocurrencia de un mensaje M. Sea L_i la longitud en la cual se presenta el mensaje M. Definimos la entropía de la fuente como:

$$H = \sum_{i=1}^{n} P_i L_i$$

La entropía de una fuente mide la cantidad de información de la misma. Mide la mínima cantidad (en promedio) de caracteres a utilizar para representar cada mensaje de la fuente.

2. Compresión de datos

- No queremos perder información y, a la vez, acercarnos a la entropía.
- La compresión es reversible.
- Los alfabetos de entrada y de salida son $0, 1$.
- Los caracteres ocupan $8b = 1B$.
- Cada caracter del archivo lo representamos usando un código de longitud L_i.

La información no se puede comprimir. La idea es comprimir los datos redundantes de la fuente. La entropía de la fuente nos proporciona el nivel mínimo al cual podemos comprimir una fuente. No se puede comprimir una fuente más alla de su entropía (usando métodos estadísticos).
Como para comprimir buscamos minimizar la entropía, debemos minimizar la longitud de cada caracter (L_i).

$$L_i = f(P_i)/H \text{ sea mínimo.}$$

Se demuestra que:

$$L_i = -\log_2(P_i)$$

$$H = \sum_{i=1}^{n} P_i \log_2\left(\frac{1}{P_i}\right)$$

2.1. Códigos

Para representar a los caracteres usaremos códigos de distinta longitud.

- La longitud de cada caracter debe acercarse a $-\log_2(P_i)$.
- Nuestro código debe ser decodificable, no debe haber ambigüedades.
- Un código es prefijo si no existe un código x perteneciente al conjunto de códigos posibles de forma tal que los n primeros bits de x sean iguales a otro posible y para cualquiern n.
- Un código prefijo es un código decodificable.
- Un código es decodificable si cumple la desigualdad de Kraft

2.2. Teorema fundamental de la compresión de datos

Si un algoritmo comprime a un conjunto de datos en un cierto factor f, necesariamente expande algún otro conjunto de datos en un factor mayor que f.

2.3. Compresión estadística

Se basa únicamente en la probabilidad de ocurrencia de cada caracter. Los algoritmos que construyen código de longitud entera son el de Huffman (óptimo) y el de Shannon-Fano. La principal desventaja es el redondeo de bits realizado para llegar al número entero de bits por caracter.

3. Métodos de compresión

3.1. Huffman estático

- Se realiza una primer pasada para calcular la probabilidad ocurrencia de cada caracter en el documento a comprimir.
- Segunda pasada comprime el texto, basándose en la probabilidad de ocurrencia calculada.

Por ejemplo sea el texto: "AABBAAABBACCD"

$$f(A) = 6, f(B) = 4, f(C) = 2, f(D) = 1$$

$$P(A) = \frac{6}{13}, P(B) = \frac{4}{13}, P(C) = \frac{2}{13}, P(D) = \frac{1}{13}$$

En este método es necesario almacenar la tabla de frecuencias, además del código comprimido en el archivo destino. Sin la tabla de frecuencias, el archivo comprimido no se puede descomprimir. Como las computadoras escriben a nivel byte, tengo que rellenar el último byte. Con tabla de frecuencias se cuando debo parar de descomprimir.

3.2. Huffman dinámico

- Probabilidad inicial: todos los caracteres con igual probabilidad, por ejemplo.
- Puedo suponer un conjunto de caracteres, para no tener que representar a todos.

Por ejemplo sea el texto: "AABAABCDAABBC", supongo "A","B","C","D","E" como caracteres posibles. Cada vez que escribo un caracter, actualizo las frecuencias de aparición y vuelvo a armar el árbol. Cuando término de comprimir marco el fin del texto comprimido con un marcador, como por ejemplo, un 1 seguido de 0 para completar el byte. De esta manera le indico al descompresor cuando debe parar de descomprimir.

$$\underbrace{101010}_{\text{código}}\ \underbrace{10}_{\text{marcador}}$$

$$\underbrace{10101011}_{\text{código}}\underbrace{10000000}_{\text{marcador}}$$

3.3. Métodos semi-estáticos

Divido el texto a comprimir en bloques y codifico estáticamente por bloques.

3.4. Métodos con contexto de aparición

Algoritmos que tienen en cuenta n caracteres anteriores al actual. Métodos de orden n.

3.4.1. Huffman estático de orden 1

Voy a emitir n caracteres al principio, debido a que no tienen contexto (1 en este caso).
Por ejemplo: "ABCABD"

$$A \Rightarrow B = 2$$
$$B \Rightarrow C = 1, D = 1$$
$$C \Rightarrow A = 1$$

$$tf(A) + tf(B) + tf(C) + A + (\text{si o si B}) + \underbrace{0}_{\text{C}} + (\text{si o si A}) + (\text{si o si B}) + (\text{si o si D})$$

$$\Rightarrow tf(A), tf(B), tf(C), A0$$

3.4.2. Huffman dinámico de orden 1

Al comenzar, emito los primeros n caracteres, o les invento un contexto. Debo llevar cuenta de un árbol de Huffman dinámico por cada contexto posible. Luego de escribir un caracter correspondiente a un contexto, actualizo el árbol de ese contexto. Al descomprimir, también debo ir armándome los árboles de cada contexto.

3.5. Métodos no-estadísticos de compresión

3.5.1. Run-Length-Encoding (RLE)

"AAABCDEEEEE" ⇒ A3B1C1D1E5

3.5.2. Variante RLE(1)

Si vienen 2 caracteres iguales, lo que sigue es una longitud. "AAABCDEEEEE" ⇒ AA1BCDEE3

3.5.3. Variante RLE(2) - Pack bits

(id,longitud)
id=0 ⇒ caracteres hasta próximo metadato (+1)
id=1 ⇒ repetición del siguiente caracter (+2)
"AAAABCFGHIEEEEE" ⇒ $(1, \underbrace{000010}_{2+2})A(0, \underbrace{0000101}_{5+1})BCFGHI(1, \underbrace{0000011}_{3+2})E$

3.5.4. LZ77

Aprovecha secuencias repetivas. Lempel-Ziv.
Versión original
Codificacion: $(\underbrace{longitud}_{4b}, \underbrace{posicion}_{12b}, \underbrace{caractersiguiente}_{8b})$
"hola juan hola pedro hola" ⇒ $L0hL0oL0lL0aL0(espacio)L0jL0uL0uL0a\ \underbrace{L1P3n}_{\text{vs } L0aL0n}\ ...\underbrace{L5P9p}_{\text{hola}}$
"AAAAAA" ⇒ $L0AL4P0A$, la ventana se va corriendo a medida que escribo y leo.

Segunda versión
Bit de codificación:

- 0 = no hubo match (0+caracter). Ocupa $9b$.
- 1 = hubo match$(1 + \underbrace{posicion}_{12b} + \underbrace{longitud}_{4b})$. Ocupa $17b$.

"ABACABAAAAAAA" ⇒ $0A0B0A0C1P3L31P0L6$.

3.5.5. LZ78/LZW

Lleva cuenta de las apariciones de las secuencias de caracteres en una tabla. La tabla comienza codificando con la cantidad de bits necesarios para codificar al alfabeto elegido y luego se va expandiendo para soportar secuencias más largas.
Sean "A","B","C","D" únicos caracteres posibles.
"ABACABAA"
Tabla inicial: $A : 000, B : 001, C : 010, D : 011, E : 100$
Empiezo a leer el texto. Encuentro A. Como está en la tabla de aparición, me fijo si está la secuencia de A y el caracter que le sigue. Como AB no está, agrego esta secuencia a la tabla y escribo el código de A en el destino. Cuando la tabla se llena. Agrego un bit al comienzo de cada entrada de la tabla y sigo codificando con estos nuevos valores.
"ABACABAA" ⇒ $\underbrace{000}_{A}, \underbrace{001}_{B}, \underbrace{000}_{A}, \underbrace{0010}_{C}, \underbrace{0101}_{AB}, \underbrace{0000}_{A}, \underbrace{0000}_{A}$

3.5.6. LZHuff/Deflate

Las longitudes y los caracteres son codificados mediante árboles de Huffman. Las posiciones se codifican mediante un árbol de Huffman canónico, es decir, cuyas frecuencias no son actualizadas. Este método tiene como parámetros la longitud mínima para la cual se considera una repetición (L_{min}), la longitud máxima de repetición (L_{max}) y el tamaño de la ventana con la cual se consideran las repeticiones.
Al tener longitudes de repetición y caracteres en el mismo árbol, no necesito indicador de si lo que sigue es un caracter o una repetición. Lo que sabemos es que luego de una longitud, viene una posición tomada del árbol canónico.

3.5.7. LZP (Predictor)

- Intenta predecir las repeticiones con una tabla de contexto de 2 caracteres.
- Utiliza árboles de Huffman dinámico para contextos de orden 1.
- Utiliza un árbol de Huffman dinámico para la longitud de las repeticiones. Cuenta con una longitud máxima representable. Por ejemplo si $L_{max} = L_2 \Rightarrow L6 = L2L2L0, L3 = L2L1$.
- En la tabla se tiene registro de la última predicción que se encontró.

Ejemplo de tabla:

Ctx2	Ctx1	Posicion
A	B	2
B	A	3
A	C	4
C	A	5

Ejemplo de archivo comprimido:

A	B	L0 (A,B)	L0 (C,A)	L0 (A,C)	L0 (B,A)	L2L2L0
A	B	A	C	A	B	ACAB

3.6. Efecto caché

Se puede mejorar el LZP en base a que es más probable encontrar un match a una distancia de alrededor de 100 caracteres de la actual. Debería ir guardando la posición de predicción que más me conviene dentro de las que encontré.

3.7. Problema de los métodos estadísticos

Cuando la probabilidad de aparición de un caracter es mucho mayor que todas las demás, los métodos estadísticos no se acercan a la entropía. Esto es porque no es posible representar a un caracter mediante estos métodos en menos de un byte. Para solucionar este problema, aparecen los métodos de compresión aritmética.

4. Compresión aritmética

Cualquier archivo puede verse como un número decimal $[0, 1)$
Por ejemplo "AABA" $\Rightarrow P(A) = \frac{3}{4}, P(B) = \frac{1}{4}$.
Tengo que subdividir el intervalo $[0, 1)$ hasta que represente bien al texto.

- A: $[0, 1), P(A) = \frac{3}{4} \Rightarrow$ me quedo con $[0, 0{,}75)$.
- A: $[0, 0{,}75), P(A) = \frac{3}{4} \Rightarrow$ me quedo con $[0, (0{,}75)^2)$.
- B: $[0, 0{,}5625), P(B) = \frac{1}{4} \Rightarrow$ me quedo con $[0{,}4218, 0{,}5625)$.
- A: $[0, 0{,}5625), P(A) = \frac{3}{4} \Rightarrow$ me quedo con $[0{,}4218, 0{,}5272)$.

El archivo comprimido quedaría como: $tf(A) + tf(B) + \alpha_2 \in [0{,}4218, 0{,}5272) + relleno$.
Para descomprimir me fijo donde en que intervalos va cayendo el decimal comprimido y voy partiendo el intervalo. Para esto necesito haber guardado la tabla de frecuencias.

4.1. Compresión aritmética dinámica

Empiezo suponiendo equiprobabilidad de aparición de los caracteres del texto a comprimir. Luego de que subdividí un intervalo, actualizo la frecuencia de los caracteres. Primero se debe subdividir y luego actualizar frecuencias ya que es lo que hará el descompresor.

4.2. Implementación

- Voy actualizando el piso y el techo del intervalo en el que trabajo.
- Cuestión de precisión: $T = 0.\underline{265}778, P = 0.\underline{265}432$. El 265 no va a cambiar, lo descarto y utilizo $T = 778, P = 432$.

4.3. Modelos de orden superior

- Deben ser dinámicos.
- Problema de frecuencia cero (ZFP). Equiprobabilidad trae problemas porque se tarda mucho en aprender. Entonces lo ideal sería empezar con frecuencia 0, pero no sabemos comprimir algo con frecuencia 0. En esos casos utilizo un símbolo de escape con frecuencia 1.
- Llegado al escape, significa que no pude comprimir en ese orden de contexto. Paso a uno menor. Si ya pase por todos los ordenes de contexto y todavía no pude comprimir, comprimo con probabilidad $\frac{1}{257}$ y actualizo los modelos por los que pasé (PPM).

Las variantes de los métodos PPM (Prediction by partial matching) difieren en como se actualiza la frecuencia del escape en cada modelo.

Método	**Frecuencia escape**
PPMA/B	1
PPMC	cantidad de caracteres del modelo con frecuencia distinta de 0
PPMD	actualizo como un caracter más

4.4. Principio de localidad

Ciertas zonas del archivo tienen alta frecuencia de aparición de determinados caracteres. A este tipo de archivos se los llama archivos localizados. Existen métodos de compresión que aprovechan este principio

4.4.1. MTF (Move to front)

- Convierte el archivo en otro antes de comprimir. Utiliza un método estadístico para aprovechar la localidad.
- Si el archivo está localizado genera un archivo con caracteres bajos.

Método

- Forma un array y con todos los posibles caracteres, ordenados por valor.
- Se procesa cada caracter del archivo escribiendo la posición del caracter en y.
- Se modifica y poniendo el caracter procesado en la primera posición.

Por ejemplo: "AABAAC"

Caracter	y	**Salida**
A	ABC	0
A	ABC	0
B	ABC	1
A	BAC	1
A	ABC	0
C	ABC	2

Ahora debo comprimir 001102 mediante un método que aproveche la mayor probabilidad de aparición de los caracteres bajos.

Block sorting
Convierte un archivo (no necesariamente localizado) en uno localizado.

- Arma matriz M con S y todas sus rotaciones.
- Ordena M por columna.
- La salida es L, útima columna de M ordenada e I, fila donde esta S en M ordenada.

Por ejemplo: S="ABADCAB"

$$\begin{pmatrix} A & B & A & D & C & A & B \\ B & A & B & A & D & C & A \\ A & B & A & B & A & D & C \\ C & A & B & A & B & A & D \\ D & C & A & B & A & B & A \\ A & D & C & A & B & A & B \\ B & A & D & C & A & B & A \end{pmatrix} \Rightarrow \begin{pmatrix} A & B & A & B & A & D & C \\ A & B & A & D & C & A & B \\ A & D & C & A & B & A & B \\ B & A & B & A & D & C & A \\ B & A & D & C & A & B & A \\ C & A & B & A & B & A & D \\ D & C & A & B & A & D & A \end{pmatrix}$$

$$L = CBBAADA, I = 1$$

Block sorting inverso

- Si tengo L, entonces puedo ordenarlo y generar F (primera columna de M ordenada).
- Se genera vector T con las posiciones de los elementos de F en L

Se genera el string original haciendo:

```
index=I;
for( k=0 ; k<len(L) ; k++ ){
        pos = T[index];
        S[k] = L[pos];
        index = pos;
}
```

4.5. Modelo de Shannon

- Modelo 0: solo tiene el caracter 0. Emite 0 solo cuando el caracter es 0, sino emite 1. Solo se utiliza cuando el caracter anterior es 0.
- Modelo 1: idem Modelo 0 pero se utiliza cuando el caracter anterior no sea 0.
- Modelo 2: idem Modelo 1 pero con el 1.
- Modelo 3: idem Modelo 1 pero con el 2.
- Modelo 4: idem Modelo 1 pero con el 3.
- Modelo 5: todos los demas caracteres (252).

Las frecuencias de los modelos deben ser actualizadas luego de partir el intervalo.

4.6. Modelo estructurado

Tengo 8 niveles. Cada uno comenzando con el numero siguiente al nivel anterior hasta el numero $2^{nivel} - 1$. En todos los niveles tengo el escape para pasar al siguiente nivel.

N0	N1	N2	N3	N4	N5	N6	N7	N8
0	1	[2,3]	[4,7]	[8,15]	[16,31]	[32,63]	[64,127]	[128,255]

5. Archivos directos

Posicion de registro en archivo está determinada por una función de hashing.

$$pos = h(clave)$$

Puede darse que $c_1 \neq c_2$ pero $h(c_1) = h(c_2)$. Se dice que c_1 y c_2 son sinónimos y hay colisión entre ellos.

5.1. Resolución de colisiones

5.1.1. Método lineal

$$pos(i) = h(c) + i$$

- (-) Este método de resolución de colisiones genera aglomeramiento de registros en una zona reducida del archivo ("clustering"). Aumenta el promedio de accesos.
- (+) Proximidad fisica en las colisiones. Puedo llegar a obtener el registro buscado en un solo acceso físico al disco.

5.1.2. Método cuadrático

$$pos(i) = h(c) + i^2$$

- (-) Puede haber registro libre y no está garantizada la inserción. Se garantiza si el factor de carga es $< \frac{1}{2}$ y N (espacio de direcciones) es primo.
- (-) Genera clustering secundario, ya que prueba siempre las mismas posiciones.
- (-) Proximidad física entre colisiones empeora, pero no tanto.
- (+) Mejora un poco el clustering, aunque no totalmente.

5.1.3. Doble hashing

$$pos(i) = h_1(clave) + i \cdot h_2(clave)$$

- (+) Elimina clustering.
- (-) Elimina proximidad física.

Para garantizar inserción $h_1 \in (0, N), h_2 \in (0, M)$ con N,M coprimos.

5.1.4. Área de overflow independiente

Área destinada para colisiones.

- (+) Es más probable llegar en un acceso. Una posición no es ocupada por un registro que no sea sinónimo de este.
- (-) Estamos físicamente lejos de las colisiones.

5.1.5. Área de overflow distribuida

Por ejemplo, cada 3 datos, 2 de overflow. Si el área de overflow cercana está llena, sigo en la próxima.

$$Dato(n) = n + \frac{n}{\text{bloque datos}} \cdot (\text{bloque overflow})$$

- (-) Búsqueda secuencial en overflow. Aumenta cantidad de accesos.

5.1.6. Encadenamiento de colisiones

Con área de overflow
Cuando se trata de insertar o leer, se busca en la posición original. Si está ocupado, se busca en el encadenado de esta posición y así sucesivamente hasta que se lo encuentra, o el último sinónimo no tiene encadenado. Al dar de baja, reemplazo el registro por su encadenado. Reemplazo el dato y el encadenamiento del que di de baja por su siguiente.
Sin área de overflow
No tiene sentido porque si la posición en la que busco hay un no-sinónimo, tendria que buscar linealmente un sinónimo para encadenarlo.

5.1.7. Buckets

En cada posición del archivo guardo n registros. Cuando se llena el bucket, utilizo algún método para buscar posición libre (encadenamiento, lineal, etc). En un bucket solo hay sinónimos. Bit de metadata para cantidad de registros libres por bucket. Implementando un archivo con buckets es más probable obtener el registro que estoy buscando en un solo sector de mi archivo. Como consecuencia, el archivo directo ocupa una cantidad de espacio mayor.

5.1.8. Método del Cuckoo

Cuento con dos funciones h_1 y h_2. Inserto en una posición si no está ocupado. Si está ocupado inserto igual y muevo el que estaba ahí a la posición alternativa (de la otra función de hashing). Sigo así hasta que inserto satisfactoriamente o hasta que entro en un ciclo infinito.

- (+) Recupero dato en 1 o 2 accesos.
- (-) Tengo que rehashear si un ciclo tarda más que un número n de veces.

Para que no haya ciclos infinitos, el grafo formado por el número de registro como vértice y las funciones de hashing como aristas (si $h_1(R_3) = 2$ y $h_2(R_3) = 4$ habrá una arista entre el vértice 2 y el 4 debido al registro 3) tiene que ser un pseudo bosque. Un pseudo bosque es un conjunto disjunto de pseudo árboles. Un pseudo árbol es un árbol con hasta 1 ciclo.
El método del Cuckoo garantiza que una consulta será resuelta en 1 o 2 accesos (utilizando 2 funciones de hashing). Un registro está en la posición determinada por una función de hashing o en la posición determinada por la otra, sin excepciones. Para lograr esto, las altas, bajas y modificaciones son costosas.

5.1.9. N funciones de hashing

Al utilizar más de una función de hashing se debe utilizar una heurística para determinar que función determina la posición final de un registro. Por ejemplo, colocar al registro en la posición con menos colisiones.

5.1.10. Grafos dirigidos

$R_k/P_i \Rightarrow P_j$ el registro k puede ir tanto en la posición i como en la j. Elijo ponerlo (en este caso) en la j. De esta manera queda determinado un grafo dirigido. Hago movimientos dentro de este grafo para que los buckets me queden lo más parejo posible. Con este método se logran eficiencias de alrededor de un 90 %.

5.2. Construcción de funciones de hashing

Objetivo: convertir un string en un número natural de la forma más pareja posible.

5.2.1. Fold and add

Tomo de a dos o cuatro bytes y realizo alguna operación entre los bytes (OR/XOR).
"abcdefgh" $\Rightarrow (ab \oplus cd) \oplus (ef \oplus gh)$.

5.2.2. Modelo multiplicativo

1. String $\Rightarrow$ número=x
2. $x \cdot A$
3. $(x \cdot A) mod 1 \Rightarrow$ se queda con la parte decimal.
4. $N \cdot [(x \cdot A) mod 1]$

5.2.3. Pearson

1. Convierte string en números entre 0 y 255.
2. Utiliza una tabla de posiciones, que son permutaciones al azar de números entre 0 y 255.
3. Toma 1 byte y busca el valor en tabla correspondiente al byte tomado. Realiza un XOR entre el byte y el valor en tabla. Este es el nuevo índice.

```
index = 0;

while( c = char_string ){

        index = c ^ table[index];

}

return index;
```

5.2.4. Hashing deslizante

Función de hashing que toma el string de a bloques. La función de hashing opera dentro del bloque y, a medida que va avanzando, el valor de la función va modificandose a través de operaciones simples.
Por ejemplo, con $n = 4$ y B un número primo (101,103, por ejemplo), tenemos que:

$$h(c_0, c_1, c_2, c_3) = c_0 B^3 + c_1 B^2 + c_2 B + c_3$$
$$h(c_1, c_2, c_3, c_4) = (h(c_0, c_1, c_2, c_3) - c_0 B^3) B + c_4$$

5.2.5. Hashing perfecto

Hashing perfecto es aquel que:

- No genera colisiones entre las claves.
- Es mínima (N claves en N posiciones).
- Preserva el orden.

Método:

1. Elegir $M >> N$.
2. Sean 2 funciones de hashing h_1 y $h_2 \Rightarrow 0, ..., M-1$.
3. Aplicar a cada clave h_1 y h_2.
4. Armar un grafo de M vértices y N aristas.
5. Verificar que el grafo sea acíclico. Si tiene ciclos, cambio M o las funciones de hashing y lo armo de nuevo.
6. Rotular cada vértice del grafo de la siguiente manera: valor del vecino = (arista − vertice)$mod N$.

7. Queda definida función G como el valor del rótulo de cada vértice.

$$h = [G(h_1(S)) + G(h_2(S))] mod N$$

5.2.6. Hashing extensible

En general se usan buckets con $N \geq 2$. Se utiliza una función de hashing y una tabla auxiliar.

- Se comienza con dos buckets y analizando el primer bit de cada registro. Ambas posibilidades (0 y 1) comienzan apuntando al mismo bucket.
- Cuando se llena el bucket se apunta el bit que genero la nueva colisión al nuevo bucket y se reordena.
- Cuando se llena este bucket, se agrega otro bit a la decodificación y otro bucket. El nuevo bucket, nuevamente, es asignado a la colisión que lo provocó.

6. Índices invertidos

- Documento: unidad mínima recuperable.
- Término: unidad mínima consultable.
- Archivo invertido: almaceno los términos (léxico) y los punteros a los documentos.

6.1. Almacenamiento de punteros

Los valores de los punteros siempre se almacenan como listas de distancias. Por ejemplo, la lista de punteros "1,2,3,8,9,11,12,19" se almacena como: "1,1,1,5,1,3,1,7". Esta representación es buena para comprimir. Las distancias pequeñas aparecen en los términos que se encuentran en varios documentos.
A continuación se presentan alternativas para almacenar dichas distancias.

6.1.1. Código unario

- $1 = 1$
- $2 = 01$
- $3 = 001$
- $n = (n-1)01$

6.1.2. Código gamma

$$\gamma(x) = 1 + \lfloor \log_2(x) \rfloor \text{ en unario, concatenado con } x - 2^{\lfloor \log_2(x) \rfloor} \text{ en binario de } \lfloor \log_2(x) \rfloor \text{ bits.}$$

Receta: $\gamma(x) \Rightarrow$ escribo x en binario. Si ocupa k bits, agrego $k-1$ ceros adelante.
Ejemplo: $\gamma(93) = \underbrace{000000}_{6b} \ \underbrace{1011101}_{93_2 \text{ ocupa } 7b}$
La vuelta: $\underbrace{00001}_{5 \text{ en unario}} \ 1011 \Rightarrow \underbrace{\lfloor \log_2(x) \rfloor}_{4} + 1 = 5 \Rightarrow x - 2^4 = 1011_2 \Rightarrow x = 27$

6.1.3. Código delta

$$\delta(x) = 1 + \lfloor \log_2(x) \rfloor \text{ en } \gamma, \text{ concatenado con } x - 2^{\lfloor \log_2(x) \rfloor} \text{ en binario de } \lfloor \log_2(x) \rfloor \text{ bits.}$$

6.1.4. Modelo global de tipo Bernoulli

Calcular la probabilidad de la distancia x. Sea p la probabilidad de que un término aparezca en un documento.

$$p = \frac{\text{cantidad de punteros que tengo}}{\text{documentos} \cdot \text{terminos}} = \frac{f}{n \cdot N} = p$$

Si $p > 0{,}5$ conviene indexar las no ocurrencias de los términos en los documentos.

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

6.1.5. Códigos de Golomb

Tienen como fin aproximar a la compresión artimética.

$$G(x,b) = q+1 \text{ en unario, concatenado con } r \text{ en binario prefijo.}$$

$$q = \lfloor \frac{x-1}{b} \rfloor$$

$$r = x - qb - 1$$

Binario prefijo: Huffman equiprobable de todos los restos posibles de dividir x y b. $(0, 1, ..., b-1)$.

$$b = \lceil \frac{\log(2-p)}{-\log(1-p)} \rceil$$

Por ejemplo: $G(17,6) \Rightarrow q = \lfloor \frac{17-1}{6} \rfloor = 2, r = 17 - 12 - 1 = 4$
3 en unario es 001, 4 en binario prefijo es 110 $\Rightarrow G(17,6) = 001110$.

6.1.6. Modelo local de tipo Bernoulli

Utilizo un b_t para cada término.

$$p_t = \frac{\text{documentos en los que aparece el termino}}{\text{total documentos}} = \frac{f_t}{N} = p_t$$

$$b_t = \lceil \frac{\log(2-p_t)}{-\log(1-p_t)} \rceil$$

6.1.7. Modelo local de frecuencia observada

Por cada entrada de términos, se comprimen las distancias utilizando árboles de Huffman. Si bien el nivel de compresión es alto, la cantidad de información adicional a guardar por cada entrada hace que el método se comporte más o menos igual que los códigos gamma, siendo estos últimos más fáciles de implementar.

6.1.8. Modelo de bytes de longitud variable

Almacenar los punteros como números de longitud variable orientados al byte. Estos códigos ocupan una cantidad entera de bytes. El formato es el siguiente.

(flag, numero binario) : si flag= 1, el número sigue en el próximo byte, si flag= 0 el número termina en el byte actual.

6.2. Almacenamiento del léxico

6.2.1. No realizar acción

Guardo todos los términos en la cantidad de bytes del término más largo. La estructura del archivo quedaria de la siguiente manera:

Término	**Frecuencia**	**Offset (bits)**
amazónica	1	0
de	2	3
el	2	5
la	3	7

Offset en código γ:

0	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9

6.2.2. Concatenar todos los términos

Offset término (bytes)	Frecuencia	Offset (en bits)
0	1	0
9	2	3
11	2	5
13	3	7

Léxico concatenado:

a	m	a	z	o	n	i	c	a	d	e	e	l	l	a
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Offset en código γ:

0	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9

Para buscar un término debo hacer busqueda binaria en la estructura de offsets.

6.2.3. Front coding

El método aprovecha la repetición de caracteres entre términos consecutivos. Para poder realizar búsqueda binaria en el índice, se utiliza front coding de n-bloques. Cada n términos fuerzo a escribir un término completo (cantidad de bytes iguales al anterior es 0).
Para hacer búsqueda binaria tengo que posicionarme en algún bloque y reconstruir las palabras de ese bloque. Me fijo si la palabra que busco está en el bloque en el que me posicioné. Este método tarda $O(\log(N+K)) = O(\log(N)$. La performance es la misma que la de la búsqueda binaria.
Por ejemplo, sean los términos: "cara","caratula","carta","cartilla","carton"

Cantidad bytes iguales	Cantidad bytes distintos	Offset bytes	Frecuencia	Offset bits
0	4	0		
4	4	4		
3	2	8		
4	4	10		
4	2	14		

c	a	r	a	t	u	l	a	t	a	i	l	l	a	o	n
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

6.3. Resolución de consultas

- Recibo término a buscar.
- Realizo búsqueda binaria en el índice hasta encontrar el término.
- Guardo la frecuencia del término en el índice. Esta frecuencia es la cantidad de códigos γ que debo interpretar.
- Convierto los código γ necesarios.
- Convierto las distancias de los códigos γ en punteros a archivo.
- Si el índice está almacenando ocurrencias, los documentos decodificados son los del paso anterior. Si el índice almacena no-ocurrencias, debo complementar los documentos encontrados en el paso anterior.

6.3.1. Consultas con distancias entre términos

Para obtener información de la distancia entre los términos debo, cuando estoy realizando el índice, almacenar información acerca de la posición del término en el documento. Así, cada documento almacenado en código γ estará seguido por el código γ de la posición en el mismo.
Por ejemplo, en la siguiente entrada: "boca",doc2,f1,p4 en el offset de los punteros a los documentos tendré que almacenar $\gamma(2)$ y $\gamma(4)$. El método es análogo para términos con frecuencia distinta de 1.

Para realizar las consultas, se realizan los mismos pasos que en el caso en el que no se guardan las posiciones, solo que, en este caso, es necesario decodificar un código γ adicional por cada aparición del término en un documento. Luego de obtener la información acerca de los términos consultados se procede a identificar los documentos en los que las posiciones de los términos difieran en menos (en módulo) que la distancia consultada.

6.3.2. Consultas con comodines

N-gramas

Para realizar consultas utilizando comodines (*), se debe generar un índice de n-gramas adicional al índice de términos. Como primer paso, se debe descomponer a todos los términos en n-gramas, de la siguiente manera:
"boca" ⇒ $b bo oc ca a$
"loca" ⇒ $l lo oc ca a$
Se genera un nuevo índice, con los n-gramas tomando el papel de "términos" y los términos en si tomando el papel de "documentos".
Al realizar una consulta, se debe descomponer el string consultado en n-gramas de la misma manera en la que se descompusieron los términos. Se procede a buscar los n-gramas dentro del índice construido y se obtienen los términos en los que aparece cada n-grama. De aquí, se realiza una intersección de los términos encontrados, para contar con los términos que presentan todos los n-gramas consultados. Se debe verificar que los términos encontrados cumplan con la consulta, ya que puede haber falsos positivos.

Léxico rotado

Para construir el índice debo rotar todo término del léxico de la siguiente manera:
"boca" ⇒ $boca, oca$b, cabo, aboc
"roca" ⇒ $roca, oca$r, caro, aroc
De esta manera puedo resolver consultas del tipo: "*oca*". Para hacerlo, necesito rotar el léxico de la consulta realizada y fijarme que n-gramas de los términos cumplen con la condición. "*oca*" ⇒ "oca**" ⇒ "oca$b", "oca$l".

7. Signature files

- Un signature por documento.
- Tamaño del signature es N bits.
- Tengo M funciones de hash que devuelven valores entre 0 y $N-1$.
- Se procede a aplicar todas las funciones de hash sobre cada término del documento, obteniéndose así el signature de cada término poniendo en 1 las posiciones determinadas por las funciones de hash.
- Se obtiene el signature del documento haciendo un OR lógico entre todos los términos del documento.

Por ejemplo (abstracto), sea T_1 tal que: $H_1(T_1) = 2, H_2(T_1) = 7, H_3(T_1) = 3$, con $N = 8$, tenemos que:
$S(T_1) = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline \end{array} = 00110001$ idem con $T_2, ..., T_k$ y luego: $S(D_1) = T_1 \vee T_2 \vee ... \vee T_k$.

7.1. Resolución de consultas

Sea T_C el término consultado:

- Aplico las M funciones de hashing sobre $T_C \Rightarrow H_1(T_C)...H_M(T_C)$.
- Obtengo el signature del término consultado $S(T_C)$.
- Realizo la operación lógica AND entre $S(T_C)$ y $S(D_i)$ para todo documento i.
- Si $S(T_C) \wedge S(D_i) = S(T_C) \Rightarrow D_i$ es candidato a contener al término consultado.

7.2. Bit slices

- Se guardan los signature files por bit.
- Todos los i bits de los k documentos están juntos (bit slice).
- Signatures de N bits $\Rightarrow N$ bit slices.

Por ejemplo, si:
$S(D_1) = 00100110$
$S(D_2) = 01100000$
$S(D_3) = 11000100$
$S(D_4) = 01100110$
Entonces:
$SL(0) = 0010$
$SL(1) = 0111$
$SL(2) = 1101$
$SL(3) = 0000$
$SL(4) = 0000$
$SL(5) = 1011$
$SL(6) = 1001$
$SL(7) = 0000$

7.2.1. Resolución de consultas

Sea T_C el término consultado.

- Aplico las M funciones de hashing sobre T_C, obteniendo $H_1(T_C)...H_M(T_C)$.
- Obtengo los bit slices $SL(i)$ que dan las M funciones de hashing $H_j(T_C) = i$.
- Realizo un AND lógico entre todos los $SL(i)$. Si tengo un 1 en la posición L, esto significa que DOC_L es candidato a contener a T_C.

Por ejemplo, con los signatures del ejemplo anterior:
Sea T_C tal que $H_1(T_C) = 1$ y $H_2(T_C) = 5 \Rightarrow SL(1) = 0111, SL(5) = 1011$
$0111 \wedge 1011 = 0011 \Rightarrow DOC_2$ y DOC_3 son candidatos.

7.3. Bitmaps

Si tengo N documentos, necesitaré N bits por término. En el mapa de bits se coloca un 0 si el término no está en el documento y un 1 si está.
Por ejemplo, si tenemos los siguientes documentos:

1	"esto es un documento"
2	"esto tambien"
3	"documento es"

Obtendremos el siguiente mapa de bits:

	1	2	3
documento	1	0	1
es	1	0	1
esto	1	1	0
tambien	0	1	0
un	1	0	0

Para resolver consultas booleanas, debo operar entre las filas del mapa de bits.

7.4. Compresión de archivos con árboles de derivación binaria

Tiene como fin principal comprimir N bits en 1 bit. Se escribe 0 si los N bits son 0 o 1 si alguno de los N bits es 1.
Por ejemplo, con $N = 2$ y con $S = 00101100011100$

```
0 0 1 0 1 1 0 0 0 1 1 1 0 0
 0   1   1   0   1   1   0   0(relleno)
   1       1       1       0
       1               1
```

Recorro desde la raiz y guardo todo lo que no sea 00
11111001101110110111
Para descomprimir rearmo el árbol. Si tengo un 1 debo buscar 2 números en la tira que guarde. Si tengo 0 inserto un 00.

```
       1               1
   1       1       1       0
 0   1   1   0   1   1   0   0
0 0 1 0 1 1 0 0 0 1 1 1 0 0 0 0
```

Debo guardar la cantidad de bits que comprimir: $\Rightarrow$ 00101100011100.

8. Page rank

$$PR(A) = (1 - \alpha) + \alpha \left(\frac{PR(T_1)}{C(T_1)} + ... \frac{PR(T_n)}{C(T_n)} \right)$$

Con:

- α factor de amortiguación.
- $T_1 ... T_n$ páginas que poseen links a A.
- $C(T_i)$ cantidad de links en la página T.

Se necesita de un factor de amortiguación α para controlar la convergencia del método, ya que este es iterativo. Se necesita que sea un término iterativo debido a que una página puede depender del page rank de otra y vice versa. Debo utilizar un page rank semilla (1 por ejemplo) para todas las páginas y comenzar a iterar hasta alcanzar la convergencia.

9. Evaluación de consultas

9.1. Precisión

Definición: Cantidad de documentos relevantes recuperados sobre el total de documentos recuperados.
Por ejemplo, si obtuve 10 resultados, y 7 de esos eran relevantes $\Rightarrow$ Precision = 0,7.

9.2. Recall

Definición: Cantidad de documentos relevantes recuperados sobre el total de documentos relevantes.
Por ejemplo, si de 10 resultados obtenidos, 7 son relevantes y fueron recuperados y otros 5 eran relevantes y no fueron recuperados $\Rightarrow$ Recall = $\frac{7}{12}$.