

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE XXX
SANTIAGO - CHILE

**“Practical Alphabet-Partitioned Rank/Select Data Structures with
Support for Distributed Computation”**

Erick Sepúlveda

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL

PROFESOR GUÍA:

Diego Arroyuelo

PROFESOR CORREFERENTE:

Marcelo Mendoza

Septiembre - 2017

Agradecimientos

A mi hermano, por siempre creer en mí. Espero no defraudarte.

A mi papá, que pese a todo siempre ha estado presente de una forma u otra, apoyándome. Gracias.

A mi profesor guía Diego Arroyuelo, cuya pasión por su campo y atención a los detalles es inigualable. Fué toda una experiencia trabajar con usted. Muy agradecido por todo el apoyo prestado.

A Simon Gog, por no dudar en responder dudas respecto al uso de su biblioteca sds1.

A todos los profesores de me hicieron clases alguna vez, en especial a Juan Bahamondes, Juan Guadalupe, Jose Luis Martí, Andres Moreira y Diego Arroyuelo, profesores ejemplares cada uno con su estilo propio.

Y obviamente a mi madre, a quien dedico no sólo este trabajo, sino que también el trabajo de los años que llevó obtener esta carrera que culmina con esto, y el trabajo que vendrá en los años futuros. Gracias por todo *perla*, soy lo que soy gracias a ti. Eres la mejor.

Agradecimientos

Resumen

En esta memoria se presentará una estructura de datos sucinta para representación de largas cadenas de texto con aplicaciones prácticas en indexamiento y búsqueda de texto entre otras.

Se mostrará como se comporta experimentalmente en cuanto a espacio utilizado y tiempo de respuesta en relación a otras estructuras usadas con los mismos fines. Cabe destacar que la estructura presenta características que la hacen viable en ambientes distribuidos, al prescindir de una estructura central sacrificando solo un poco de espacio.

Además se muestra una aplicación concreta, calculando la intersección de listas invertidas.

Su implementación puede ser revisada en

<http://github.com/ericksepulveda/asap>.

Palabras Clave: wavelet tree, vector de bits, estructuras sucintas, alphabet partitioning, distribuido, intersección, sdsl, entropía de orden cero

Abstract

In this project it will be introduced a wavelet tree/bit vector based succinct data structure that uses an alphabet partitioning strategy, to represent long string sequences with practical applications in text indexing and search among others.

Its performance, space and time wise, will be shown and compared against other data structures used with similar purposes. It is worth noting that this data structure also shows characteristics that make it viable on distributed environments by being able to prescind from a central data structure at the expense of only a little space.

Furthermore, it will be shown how it performs in a concrete application, intersecting inverted lists.

The code is available for any to check and use at

<http://github.com/ericksepulveda/asap>.

Keywords: wavelet tree, bit vector, succinct data structure, alphabet partitioning, distributed, intersection, sdsl, zero order entropy

Tabla de Contenido

Introducción	1
1 Antecedentes	2
1.1 Definición del Problema	2
1.2 Aplicaciones	2
1.3 Objetivos	3
1.3.1 Objetivo Principal	3
1.3.2 Objetivos Secundarios	3
2 Estado del Arte	4
2.1 Compresión de Strings	4
2.2 Rank y select sobre secuencias de bits	5
2.2.1 Vectores de bits	5
2.3 Rank y select sobre secuencias arbitrarias	7
2.3.1 Wavelet Trees	7
2.3.2 Representación de Golynski, Munro y Rao	11
2.4 Alphabet Partitioning	13
2.4.1 Consideraciones Prácticas	14
2.4.2 Ejemplo	15
2.4.3 Uso de espacio	17
3 Una implementación práctica para <i>rank</i> y <i>select</i> basada en Particiona-	
miento del alfabeto	19
3.1 Definición de la Estructura de Datos	19
3.2 Ejemplo	21
3.3 Uso de espacio y tiempos de respuesta	23

4	Resultados Experimentales	24
4.1	Hardware empleado	24
4.2	Código	24
4.3	Texto representado	24
4.4	Resultados	25
4.5	Experimentos con Rank y Select	26
4.6	Experimentos con Access	29
4.7	Aplicación 1: Extracción de fragmentos en bases de datos textuales . . .	30
4.8	Aplicación 2: Intersección de listas invertidas en Recuperación de la Información	31
4.9	Aplicación 3: Computación distribuida de rank y select	34
5	Conclusiones y trabajo futuro	39
	Referencias Bibliográficas	41

Índice de Figuras

Figura 10	Resultados experimentales para la operacion <i>rank</i> con consultas desde un <i>query log</i> . El eje x empieza en $H_0(s) = 12,45$ bits.	27
Figura 11	Resultados experimentales para la operacion <i>rank</i> con consultas aleatorias. El eje x empieza en $H_0(s) = 12,45$ bits.	28
Figura 12	Resultados experimentales para la operacion <i>select</i> con consul- tas desde un <i>querylog</i> . El eje x empieza en $H_0(s) = 12,45$ bits.	28
Figura 13	Resultados experimentales para la operacion <i>select</i> con consul- tas aleatorias. El eje x empieza en $H_0(s) = 12,45$ bits.	29

Figura 14	Resultados experimentales para la operacion access. El eje x empieza en $H_0(s) = 12,45$ bits.	30
Figura 15	Resultados experimentales para extracción de fragmentos con largo 100. El eje x empieza en $H_0(s) = 12,45$ bits.	32
Figura 16	Resultados experimentales para extracción de fragmentos con largo 200. El eje x empieza en $H_0(s) = 12,45$ bits.	32
Figura 17	Resultados experimentales para extracción de fragmentos con largo 300. El eje x empieza en $H_0(s) = 12,45$ bits.	33
Figura 18	Resultados experimentales para extracción de fragmentos con largo 500. El eje x empieza en $H_0(s) = 12,45$ bits.	33
Figura 19	Resultados experimentales para extracción de fragmentos con largo 1000. El eje x empieza en $H_0(s) = 12,45$ bits.	34
Figura 20	Resultados experimentales para la intersección de listas invertidas	36

Índice de Tablas

Tabla 1	Símbolo c , su correspondiente frecuencia n_c y su ranking $r(c)$ dentro del string de ejemplo, y la partición asignada $\lfloor \log r(c) \rfloor$	15
Tabla 2	Resultados experimentas para la implementación de AP y $ASAP$ en ambiente distribuido Los tiempos están en promedio de microsegundos por operación (para <i>snippet</i> es promedio de microsegundos por símbolo extraído). En este caso $ASAP$ usa wavelet matrix en los s_l y Huffman shaped wavelet tree en m . El particionamiento es dense-23 , lo que generó 46 particiones (número de nodos)	38

Introducción

Los wavelet trees [13] son un tipo de estructura de datos sucinta esto quiere decir que utilizan espacio cercano a la cota inferior teórica necesaria para representar un conjunto de datos. Esta propiedad es de importancia cuando el volumen de datos es particularmente grande lo cual hace que mantenerlos en memoria se dificulte. Para esto, una representación como los wavelet trees es de ayuda tanto porque el espacio que utilizan es muy acotado, como porque son capaces de responder consultas de manera eficiente en su estado comprimido.

La implementación a mejorar está descrita a fondo en [4]. Esta implementación en particular es de interés por saberse eficiente. Se buscará mejorar respecto al espacio utilizado y los tiempos de respuesta de las consultas *access*, *rank* y *select*, u obtener un *tradeoff* competitivo.

De modo de ser capaz de realizar esta mejora se deberán estudiar los wavelet trees, sus propiedades, su funcionamiento, implementaciones conocidas y en particular la implementación a mejorar en profundidad.

Para poner en práctica la implementación propuesta se utilizará la librería Succinct Data Structure Library(SDSL) [7] para C++, que implementa estructuras de datos de tipo sucintas acorde a como han sido presentadas en diferentes papers que pueden ser revisados en [7].

1. Antecedentes

1.1. Definición del Problema

Dado un string $s[1..n]$ sobre un alfabeto $\Sigma = \{0, \dots, \sigma - 1\}$, la operación $s.rank_c(i)$ calcula el número de ocurrencias del símbolo $c \in \Sigma$ en $s[1..i]$. Por otro lado, la operación $s.select_c(j)$ entrega la posición de la j -ésima ocurrencia del símbolo c . Finalmente, $s.access(i)$ entrega el símbolo $s[i]$.

El problema es implementar una solución que sea capaz de responder eficientemente estas consultas, usando el menor espacio posible, haciendo su uso viable en la práctica e, idealmente, que sea competitiva con las estructuras usadas actualmente para los mismos fines.

1.2. Aplicaciones

Las principales aplicaciones para este problema se dan en campos como la bioinformática, minería de datos y recuperación de la información, donde se debe trabajar con grandes volúmenes de datos, por lo que un problema con el que se encuentran es la limitación en memoria que impide tener todos los datos simultáneamente accesibles y disponibles para ser consultados.

Aplicaciones concretas son *snippet extraction* en bases de datos de texto [3], procesamiento de consultas en recuperación de la información [1, 2], árboles cardinales, búsqueda en texto, representaciones de grafos [4] e índices invertidos de posiciones, entre otras.

Una posible solución al problema planteado (y quizás la más conocida) son los *wavelet trees*. Se recomienda revisar [13], donde se muestran no sólo las posibles aplicaciones, sino que también las ventajas que exhiben los *wavelet trees* en ellas.

1.3. Objetivos

Para este trabajo se definieron los siguientes objetivos:

1.3.1. Objetivo Principal

- Implementar una solución para el problema de *rank* y *select* sobre strings, basada en la técnica de *Alphabet Partitioning*.

1.3.2. Objetivos Secundarios

- Evaluar las implementaciones existentes para resolver el problema de *rank* y *select* sobre strings, respecto a su uso de memoria y el tiempo que necesitan para responder consultas de *rank*, *select* y *access*.
- Evaluar y comparar el desempeño de la estructura implementada.
- Comprobar su rendimiento en aplicaciones reales, comparándolo con estructuras similares bajo las mismas condiciones.
- Evaluar su viabilidad en ambientes distribuidos.
- Agregar la implementación propuesta a la librería SDSL, actualmente la de mayor uso para estructuras de datos sucintas.

2. Estado del Arte

A continuación se presentará el estado del arte respecto a estructuras de datos para soportar las operaciones *rank* y *select*, tanto para secuencias de bits en particular como para strings en general.

Es importante mencionar que todas las estructuras de datos que estudiaremos están implementadas en la Succinct Data Structure Library [7] la cual es una librería con estructuras de datos sucintas disponibles para usar en C++. La misma es una herramienta altamente configurable a través de parámetros de *template* y se documenta directamente por medio de los papers que postulan las estructuras de datos implementadas. Dicha librería es la principal para la comunidad de estructuras de datos sucintas y comprimidas.

2.1. Compresión de Strings

Para poder medir qué tan bueno es un resultado en términos de compresión, se considerará la entropía de orden cero $H_0(s)$, que representa la cantidad de bits en promedio que necesita como mínimo cada símbolo de la cadena a representar. Se calcula mediante la siguiente fórmula:

$$H_0(s) = \sum \frac{n_c}{n} \log \frac{n}{n_c} \quad (1)$$

donde n_c corresponde a la frecuencia del carácter c en la cadena s , y n es el largo de la cadena. La suma sólo incluye los símbolos que ocurren en s , tales que $n_c > 0$. Notar que el valor de $H_0(s)$ cumple con que $H_0(s) \leq \log \sigma$ y entre mayor sea la diferencia entre ambos valores, más compresible será el texto.

2.2. Rank y select sobre secuencias de bits

2.2.1. Vectores de bits

Dado un vector de bits $B[0..n - 1]$ se definen las siguientes operaciones sobre él:

- $B.rank(i)$: Cuenta la cantidad de 1s hasta la posición i .
- $B.select(i)$: Entrega la posición del i -ésimo 1 dentro del vector de bits.
- $B.access(i)$: Entrega el bit en la posición i .

Se definen como **vectores de bits densos** aquellos en los que la cantidad de 1s y 0s es similar, y por otro lado los **vectores de bits poco densos** (*sparse*) son aquellos en los que la cantidad de 1s es mucho menor a la cantidad de 0s.

2.2.1.1. SArray

Sadakane y Okanohara proponen esta alternativa que tiene dos variantes; *SArray* para vectores de bits poco densos (*sparse*) y *DArray* para vectores de bits densos. *SArray* usa *DArray* en partes de su estructura [15].

Para construir **SArray** se toma como entrada un vector de bits $B[0..n - 1]$ con m 1s ($m \ll n$). Se define un vector $x[0..m - 1]$ construido tal que $x[i] = B.select(i + 1)$ con $i \in [0..m - 1]$, esto es, el vector x contiene las posiciones de todos los 1s en B . Luego, cada valor en x se divide en $z = \lfloor \log m \rfloor$ bits superiores y $w = \lceil \log \frac{n}{m} \rceil$ bits inferiores. Los w bits inferiores de cada valor en x son almacenados explícitamente en $L[0..m - 1]$ usando un total de mw bits.

Los z bits superiores de cada valor en x son almacenados en un arreglo de bits $H[0..2m - 1]$ donde $H[x_i/2^w + i] = 1$ y los otros son 0. Esto se traduce como una codificación unaria de la cantidad de ocurrencias de cada combinación de los bits superiores.

El uso de espacio es $m \log \frac{n}{m} + 2m + o(m)$ bits. *Select* se puede responder en tiempo $O(1)$ mientras que *rank* y *access* se pueden resolver en tiempo $O(\log \frac{n}{m})$. Los resultados

experimentales mostrados en [15] muestran que el espacio usado por esta estructura es cercano a $H_0(B)$.

Para **DArray** el vector de bits de input tiene la forma $B[0..n-1]$ con $m \simeq n/2$ 1s. Se particiona H en bloques en los que cada uno tiene L 1s respectivamente y finalmente se crean los vectores de bits $P_l[0..n/L-1]$ donde $P_l[i]$ es la posición del $(iL+1)$ -ésimo 1.

2.2.1.2. RRR

La representación RRR (Por Raman, Raman y Rao [17]) toma como entrada un vector de bits $B[0..n-1]$, el cual es particionado en bloques de tamaño b , los cuales a su vez son agrupados en superbloques de tamaño f . A cada bloque se le asigna un par (C, O) donde C corresponde a la clase del bloque. Esta es calculada usando la operación **popcount**, que cuenta la cantidad de bits seteados en 1 dentro del bloque.

Dado el tamaño de bloque b , existen $\binom{b}{C}$ posibles bloques que tienen C bits en 1. Se tiene una tabla E , que almacena cada posible combinación de los C bits, para cada C . Así, para determinar cuál es la combinación representada, primero se accede a las combinaciones de la clase C y luego O actúa como índice que selecciona la permutación correcta de los C bits. Aquí se almacena además los resultados de *rank* a nivel de bloque (por lo tanto usan $\log b$ bits por *rank*). Los valores de C siempre usaran $\log b + 1$ bits, mientras que los valores de O varían en tamaño requiriendo $\log \binom{b}{C}$ bits.

Para cada superbloque se almacena el resultado de *rank* hasta el inicio del mismo, a nivel global.

De esta forma para responder $rank(i)$ se procede de la siguiente manera:

1. Se calcula el bloque al que pertenece i como $i_b = \frac{i}{b}$.
2. Se calcula el superbloque al que pertenece i_b como $i_s = \frac{i_b}{f}$.
3. El resultado de la consulta se inicializa con el valor de *rank* correspondiente al almacenado en el superbloque.

4. Se itera sobre los bloques del superbloque correspondiente, añadiendo al resultado C mientras que no se llegue al bloque i_b .
5. Al llegar a i_b se suma solo el resultado de $rank$ hasta la posición i mód b . Esto se hace accediendo a la entrada de la tabla E con (C, O) donde los resultados estaban previamente computados.

La estructura usa espacio $nH_0(B) + o(n)$ bits, y es capaz de responder las consultas $rank$, $select$ y $access$ en tiempo $O(1)$.

2.3. Rank y select sobre secuencias arbitrarias

2.3.1. Wavelet Trees

Los wavelet trees son árboles de vectores de bits. Se considerará la representación de un string s de largo n , con un alfabeto Σ de tamaño σ . El espacio usado por esta estructura es $n \log \sigma + o(n \log \sigma)$ bits [13]. Cada nivel del árbol usa, como cota superior, n bits, y si el árbol es balanceado tendrá altura $\log \sigma$, lo que resulta en los $n \log \sigma$ bits. Los $o(n \log \sigma)$ bits restantes están dados por diferentes estructuras de soporte, junto con los mismos punteros que dan forma al árbol. Soporta las consultas $rank$, $select$ y $access$ en tiempo $O(\log \sigma)$.

El proceso de construcción de los wavelet trees empieza dividiendo el alfabeto en dos partes transformando el string original en un vector B de n bits, donde $B[i] = 0$ corresponde a un símbolo $s[i]$ en la primera partición del alfabeto, y cada $B[i] = 1$ corresponde a un símbolo de la segunda partición del mismo.

El proceso es recursivo hasta obtener particiones que contienen sólo un símbolo, declarándose éste un nodo hoja del wavelet tree.

2.3.1.1. Wavelet trees: Operaciones

Dentro de esta estructura de datos se definen tres operaciones básicas:

- $s.access(i)$: Entrega el carácter ubicado en la posición i .
- $s.rank_c(i)$: Cuenta las ocurrencias del carácter c hasta la posición i .
- $s.select_c(i)$: Entrega la posición de la i -ésima ocurrencia del carácter c .

Estas tres operaciones pueden computarse dentro de un vector de bits en tiempo $O(1)$ usando espacio adicional proporcional a $o(n)$ [5] [11], logrando así el tiempo antes nombrado $O(\log \sigma)$ al tener que recorrer el árbol en profundidad.

Dado que el wavelet tree reemplaza al texto que representa, para calcular $s.access(i)$ y obtener el símbolo $s[i] = c$ se debe recorrer el árbol desde la raíz al nodo hoja que representa a c . Para hacerlo, se empieza por la raíz, y se chequea si $B_{raiz}[i] = 0$. Si es así, entonces $s[i] = c$ está en el sub-árbol izquierdo v_l de la raíz, por lo que el proceso continua en v_l buscando el símbolo en la posición $B_{raiz}.rank_0(i)$. Si no, el proceso continua en el sub-árbol derecho v_r , buscando el símbolo en la posición $B_{raiz}.rank_1(i)$. Este proceso es recursivo hasta llegar a la hoja que contiene $s[i] = c$.

Para calcular $s.rank_c(i)$ para cualquier $c \in \Sigma$, se procede casi igual que con $access$, usando la codificación binaria que le fué asignada a c que determina su posición como hoja en el árbol. Por otro lado, para calcular $s.select_c(i)$ para cualquier $c \in \Sigma$, se navega el wavelet tree partiendo por la hoja que representa a c . Si la hoja era un sub-árbol izquierdo, entonces en el vector de bits superior se debe calcular $B.select_0(i)$. Si no, entonces se calcula $B.select_1(i)$. Este resultado corresponde al resultado de la consulta $s.select_c(i)$ hasta esa altura, por lo que si el vector de bits visitado no correspondía a B_{raiz} , el proceso se repite recursivamente hasta llegar a la raíz.

2.3.1.2. Huffman Shaped Wavelet Tree

En [12] se plantea el uso de árboles con la forma de Huffman en vez de balanceados. Esta forma obtiene en promedio un tiempo de respuesta de $O(H_0(s))$ en vez de $O(\log \sigma)$.

Si recordamos el algoritmo de Huffman, éste codifica cada símbolo de la secuencia de manera de reducir la cantidad de bits necesarios para ello. Para esto considera la

frecuencia por símbolo n_c y crea un árbol donde los símbolos menos frecuentes terminan siendo hojas con una profundidad mayor a los símbolos más frecuentes que son ubicados más cerca del nodo raíz. Definimos entonces, para cada símbolo $c \in \Sigma$ su correspondiente nodo hoja tendrá profundidad h_c de forma que $\sum_{c \in \Sigma} h_c n_c \leq H_0(s) + 1$.

Notar que cada ocurrencia de cada símbolo $c \in \Sigma$ aparece en exactamente h_c vectores de bits, desde la raíz hasta el nodo hoja que lo contiene. Por lo tanto, cuesta h_c bits representar cada símbolo c repartido entre distintos vectores de bits. Haciendo la suma para todos los caracteres se obtiene $\sum_{c \in \Sigma} h_c n_c$ bits por lo que la compresión alcanzada es la misma que con la compresión del algoritmo de Huffman. Por lo tanto el espacio total utilizado es $n(H_0(s) + 1)(1 + o(1))$ bits.

Recordar que el tiempo en responder las consultas en un wavelet tree está dado por la profundidad de la hoja que corresponde al símbolo consultado. Dada la codificación de Huffman, en este caso los símbolos más frecuentes se verán beneficiados mientras que los menos frecuentes se verán perjudicados.

2.3.1.3. Wavelet Matrix

En [6] se describen las wavelet matrices, las que consisten en una implementación de wavelet trees para alfabetos grandes. En los wavelet trees, para almacenar la topología de los mismos son necesarios $O(\sigma \log n)$ bits adicionales, considerando punteros de $O(\log n)$ bits. Este término puede ser obviado para alfabetos pequeños, pero cuando el alfabeto tiene un tamaño considerable frente al tamaño del texto, se transforma en un problema.

Un primer acercamiento para resolver esto es, dado un árbol balanceado, concatenar todos los vectores de bits $B_{(a,b)}$ (a y b son símbolos, $B_{(a,b)}$ denota al vector de bits que tiene al intervalo de símbolos dados por $[a, b]$) por nivel l en un único B_l , descartando la topología. Sin embargo esto genera los problemas de que al querer recorrer el árbol, se tiene que identificar en qué rango de B_l empieza y termina el vector de bit $B_{(a,b)}$. Además en los niveles más profundos del árbol los B_l pueden tener discontinuidades

debido a que pueden haber nodos hojas con profundidad $\lceil \log \sigma \rceil - 1$, generando ruido en el determinamiento de los rangos.

La *Wavelet Matrix* resuelve estos problemas manteniendo casi la misma velocidad que un wavelet tree basado en punteros. Aquí los vectores de bit $B_l[1, n]$ con $l \in [1, \lceil \log \sigma \rceil]$ son construidos concatenando los vectores de bit $B_{(a,b)}$ con un orden en particular: todos los nodos en el nivel l que son hijos izquierdos son puestos antes que los nodos que son hijos derechos. Entre hijos izquierdos el orden está dado por el orden de sus nodos padres. Además de este ordenamiento se necesita guardar la cantidad de 0s por nivel z_l .

Para calcular $s.\text{access}(i)$ en $O(\log(\sigma))$ se parte en el primer nivel ($l = 1$) donde se sabe que $s.\text{access}(i) \in [a, b] = [1, \sigma]$. Si $B_l[i] = 0$, entonces se sustituye i por $B_l.\text{rank}_0(i)$ y $[a, b] \leftarrow [a, \lfloor (a+b)/2 \rfloor]$, esto es, se continúa la búsqueda con los 0s, que contienen la primera mitad de los símbolos del alfabeto actual. Si de lo contrario $B_l[i] = 1$ entonces las sustituciones realizadas son $i \leftarrow 1 + B_l.\text{rank}_1(i)$ y $[a, b] \leftarrow [\lfloor (a+b)/2 \rfloor + 1, b]$, continuando la búsqueda con los 1s que corresponden a la segunda mitad de los símbolos del alfabeto actual. Cuando se llegue a tener el intervalo donde $a = b$, entonces el algoritmo ha terminado y $s.\text{access}(i) = a$.

Para calcular $s.\text{rank}_c(i)$ se necesita además saber donde comienza el vector de bits $B_{(a,b)}$. En la práctica es más simple mantener la posición previa. Se parte con esta posición $p \leftarrow 0$, en el nivel $l = 1$ con $[a, b] = [1, \sigma]$. Si $c \leq \lfloor (a+b)/2 \rfloor$ se continúa por la izquierda actualizando $i \leftarrow B_l.\text{rank}_0(i)$, $[a, b] \leftarrow [a, \lfloor (a+b)/2 \rfloor]$ y $p \leftarrow B_l.\text{rank}_0(p)$. Si de lo contrario, $c > \lfloor (a+b)/2 \rfloor$ se continúa por la derecha actualizando $i \leftarrow z_l + B_l.\text{rank}_1(i)$, $[a, b] \leftarrow [\lfloor (a+b)/2 \rfloor + 1, b]$ y $p \leftarrow z_l + B_l.\text{rank}_1(p)$. Nuevamente, cuando se a un intervalo con la forma $[a, b] = [c, c]$, la respuesta de $s.\text{rank}_c(i)$ estará dada por $i - p$.

El espacio usado por esta estructura es $nH_0(s) + o(n \log \sigma)$.

2.3.2. Representación de Golynski, Munro y Rao

En [8] se considera una cadena s de n enteros en el rango $[1..t]$, con $t < n$. La cadena es representada a través de una tabla T de dimensiones $t \times n$. $T[c, i]$ indica si el entero c ocurre en la posición i . Concatenando todas las filas se obtiene el vector de bits A de largo tn . Las operaciones $s.rank_c(i)$ y $s.select_c(i)$ pueden ser calculadas usando A así:

$$s.rank_c(i) \equiv A.rank((c-1)n + i) - A.rank((c-1)n), \quad (2)$$

y además

$$s.select_c(i) \equiv A.select(A.rank((c-1)n + i) - (c-1)n) \quad (3)$$

Dividiendo A en bloques de tamaño t es posible definir una versión restringida de $rank$ y $select$ llamados $rank-b$ y $select-b$ relacionadas a estos bloques. $rank-b(i) = rank(i)$ si i es un múltiplo de t . Si no lo es, es indefinido. $select-b(i)$ es el número del bloque en el que el i -ésimo 1 está. Así, $rank-b$ queda definido por:

$$rank-b(it) = A.rank(it), \quad (4)$$

y $select-b$

$$select-b(t) = \left\lfloor \frac{A.select(i)}{t} \right\rfloor \quad (5)$$

Definimos la cardinalidad del bloque i como $k_i = rank-b(it) - rank-b((i-1)t)$. A partir de esto se construye el vector de bits B donde están todas las cardinalidades de los bloques en unario ($B = 1^{k_1}01^{k_2}0 \dots 1^{k_n}0$). Con esto podemos redefinir las funciones anteriores:

$$rank-b(it) = B.rank(B.select_0(i)), \quad (6)$$

y:

$$select-b(t) = B.rank_0(B.select(i)) \quad (7)$$

Esta estructura usa espacio $2n + o(n)$ bits.

Para soportar *rank* y *select* completamente hace falta poder hacer operaciones dentro de los bloques de *A*. Para esto se definen las operaciones *rank-l_i*:

$$rank-l_i(j) = A.rank(j + it) - A.rank(it) \quad (8)$$

y *select-l_i*

$$select-l_i(j) = A.select(j + A.rank((i - 1)t)) \quad (9)$$

Y con esto finalmente es posible definir *rank*:

$$rank(j) = rank-b(it) + rank-l_i(j - it), \text{ con } i = \lfloor j/t \rfloor \quad (10)$$

y *select*:

$$select(j) = select-l_i(j - rank-b((i - 1)t) + (i - 1)t, \text{ con } i = select-b(j) \quad (11)$$

Esta estructura se muestra más eficiente que las otras alternativas, al igual que la

Wavelet Matrix, para alfabetos más grandes. Requiere $n(\log \sigma + o(\log \sigma))$ bits de espacio, soportando la operación *rank* en tiempo $O(\log \log \sigma)$, mientras que las operaciones *select* y *access* son soportadas en tiempo $O(1)$ (esto entre otros *trade-offs*, ver el paper original para más detalles). Más tarde Grossi et al. [10] lograron compresión de mayor orden, esto es, $nH_k(s) + o(n \log \sigma)$ bits. Las operaciones *rank* y *select* son soportadas en tiempo $O(\log \log \sigma)$, mientras que la operación *access* es soportada en tiempo $O(1)$.

2.4. Alphabet Partitioning

Barbay et al. [4] presentaron una estructura de datos para *rank* y *select* sobre strings, la cual está basada en la técnica *Alphabet Partitioning* [18], la cual particiona el alfabeto original Σ en p subalfabetos $\Sigma_0, \Sigma_1, \Sigma_2, \dots, \Sigma_{p-1}$, donde $\bigcup_{i=0}^{p-1} \Sigma_i = \Sigma$ y $\Sigma_i \cap \Sigma_j = \emptyset \forall i, j$.

Para empezar, se construye la estructura encargada de asignarle a cada símbolo $c \in [0 \dots \sigma]$ su subalfabeto. Una forma de definir a qué subalfabeto debe ser asignado un símbolo es la siguiente:

$$m[c] = \left\lceil \log \left(\frac{n}{n_c} \right) \log(n) \right\rceil, \quad (12)$$

donde n_c corresponde a la cantidad de veces que el símbolo c ocurre en la cadena original s , y $m[c] = j$ quiere decir que el símbolo c fue asignado al subalfabeto Σ_j . Este particionamiento se conocerá como **sparse**. Dado que $\log \left(\frac{n}{n_c} \right) \leq \log(n)$, el alfabeto de $m \in [1 \dots \lceil \log^2 n \rceil]$ y $\sigma_m = p \leq \log^2 n$, lo que será relevante al considerar el espacio utilizado por la estructura en su totalidad. Esta estrategia tiende a agrupar en una misma partición los símbolos c que tienen una frecuencia n_c similar.

Cada Σ_j tiene un largo $|\Sigma_j| = n_j$. Los símbolos asignados a Σ_j son codificados de manera que el alfabeto de Σ_j es $[0 \dots \sigma_j]$. La recodificación para un símbolo c que fue asignado al subalfabeto Σ_j se hace de esta forma: si hay k símbolos que ya fueron asignados al subalfabeto Σ_j , entonces c será codificado como k . Formalmente: $m.rank_j(c) = k$.

Como segunda estructura se tiene el vector t de largo n , donde cada c es reemplazado por el subalfabeto $m[c]$ al que fue asignado. Al igual que m , t también tiene alfabeto $\in [1 \dots \lceil \log^2 n \rceil]$ y $\sigma_t = p \leq \log^2 n$, por la misma razón. Además, hay n_0 ocurrencias del símbolo 0 en t , n_1 ocurrencias del símbolo 1 en t , etc. Por lo tanto, definimos:

$$H_0(t) = \sum_{i=0}^{p-1} \frac{n_i}{n} \log \frac{n}{n_i}. \quad (13)$$

Finalmente cada subalfabeto es representado por su cadena s_l con $l \in m[c]$, $\forall c$. La construcción de estos está dada por:

$$s_l[t.rank_i(i)] = m.rank_l(s[i]), \text{ para todo } 1 \leq i \leq n \text{ tal que } t[i] = l. \quad (14)$$

Dadas estas estructuras, es posible definir *access* de la siguiente manera:

$$s.access(i) = m.select_l(s_l.access(t.rank_l(i))), \text{ con } l = t.access(i), \quad (15)$$

la operación *rank* de la siguiente manera:

$$s.rank_a(i) = s_l.rank_c(t.rank_l(i)), \text{ con } l = m.access(a) \text{ y } c = m.rank_l(a), \quad (16)$$

y por último, *select*:

$$s.select_a(i) = t.select_l(s_l.select_c(i)), \text{ con } l = m.access(a) \text{ y } c = m.rank_l(a). \quad (17)$$

2.4.1. Consideraciones Prácticas

En la práctica, el particionamiento **sparse** definido en la Ecuación (12) es reemplazado por un esquema en el que para cada $c \in \Sigma$, $m[c] = \lfloor \log r(c) \rfloor$. Aquí $r(c)$ denota el ranking del símbolo c de acuerdo a su frecuencia (esto es, el símbolo más frecuente

tiene ranking 1, mientras que el menos frecuente tiene ranking σ). Por lo tanto, la primera partición contiene solo un símbolo (el más frecuente), la segunda partición tiene dos símbolos, la tercera partición contiene cuatro símbolos y así sucesivamente. Por consiguiente, existen $p = \lceil \log \sigma \rceil$ particiones. Esta estrategia de particionamiento se llamará **dense** [4]. Otra consideración práctica consiste en tener un parámetro l_{min} para el particionamiento **dense**, tal que los $2^{l_{min}}$ símbolos más frecuentes sean representados directamente en t . Esto quiere decir que no son representados en ninguna partición. Notar que el particionamiento **dense** puede se obtiene con $l_{min} = 1$.

2.4.2. Ejemplo

Para visualizar la estructura se considerará el string:

"to_be_or_not_to_be_that_is_the_question",

el cual tiene largo $n = 39$, su alfabeto es $\Sigma = \{_, t, o, b, e, r, n, h, a, i, s, q, u\}$ con lo que $\sigma = 13$. Por lo tanto, $p = \lceil \log \sigma \rceil = 4$. La Tabla 1 muestra la partición que le fué asignada a cada símbolo c , $m[c] = \lfloor \log r(c) \rfloor$, junto con la frecuencia n_c y ranking $r(c)$ usados para obtenerla.

Tabla 1: Símbolo c , su correspondiente frecuencia n_c y su ranking $r(c)$ dentro del string de ejemplo, y la partición asignada $\lfloor \log r(c) \rfloor$.

c	n_c	$r(c)$	$\lfloor \log r(c) \rfloor$
_	9	1	0
t	5	2	1
o	4	3	1
e	2	4	2
b	2	5	2
h	2	6	2
i	2	7	2
n	2	8	3
s	2	9	3
a	1	10	3
q	1	11	3
r	1	12	3
u	1	13	3

Con esto se tiene que los subalfabetos son $\Sigma_0 = \{_ \}$, $\Sigma_1 = \{t, o\}$, $\Sigma_2 = \{e, b, h, i\}$ y $\Sigma_3 = \{n, s, a, q, r, u\}$, con $\sigma_0 = 1$, $\sigma_1 = 2$, $\sigma_2 = 4$ y $\sigma_3 = 6$. Notar que todas las particiones excepto la última tienen tamaño 2^l .

En la Figura 1 se puede ver la representación de m , y en la Figura 2 la representación de t para el string "to_be_or_not_to_be_that_is_the_question". Finalmente tenemos los s_l , que se muestran en las Figuras 3, 4 y 5. Recordar que s_0 no existe porque $\sigma_0 = 1$, por lo que el único símbolo '_' en $\Sigma_0 = \{_ \}$ queda representado directamente

$$\begin{array}{c} _ t o e b h i n s a q r u \\ \boxed{0} \boxed{1} \boxed{1} \boxed{2} \boxed{2} \boxed{2} \boxed{2} \boxed{3} \boxed{3} \boxed{3} \boxed{3} \boxed{3} \end{array}$$

Figura 1: Representación de m para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

$$\begin{array}{c} t o _ b e _ o r _ n o t _ t o _ b e _ t h a t _ i s _ t h e _ q u e s t i o n \\ \boxed{1} \boxed{1} \boxed{0} \boxed{2} \boxed{2} \boxed{0} \boxed{1} \boxed{3} \boxed{0} \boxed{3} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{2} \boxed{2} \boxed{0} \boxed{1} \boxed{2} \boxed{3} \boxed{1} \boxed{0} \boxed{2} \boxed{3} \boxed{0} \boxed{1} \boxed{2} \boxed{2} \boxed{0} \boxed{3} \boxed{3} \boxed{2} \boxed{3} \boxed{1} \boxed{2} \boxed{1} \boxed{3} \end{array}$$

Figura 2: Representación de t para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

en t .

$$\begin{array}{c} t o o o t t o t t t t o \\ \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \end{array}$$

Figura 3: Representación de s_1 para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

$$\begin{array}{c} b e b e h i h e e i \\ \boxed{1} \boxed{0} \boxed{1} \boxed{0} \boxed{2} \boxed{3} \boxed{2} \boxed{0} \boxed{0} \boxed{3} \end{array}$$

Figura 4: Representación de s_2 para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

$$\begin{array}{c} r n a s q u s n \\ \boxed{4} \boxed{0} \boxed{2} \boxed{1} \boxed{3} \boxed{5} \boxed{1} \boxed{0} \end{array}$$

Figura 5: Representación de s_3 para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

2.4.3. Uso de espacio

Barbay et al. [4] mostraron que $nH_0(t) + \sum_{i=0}^{p-1} n_i \log \sigma_i \leq nH_0(s) + o(n)$. Esto significa que si para representar t se usa una estructura que soporte las operaciones *rank/select*

con compresión de orden cero, y se representan los s_l sin comprimir, se obtiene compresión de orden cero para el string s . Recordar que $p \leq \lceil \log^2 n \rceil$, por lo tanto los alfabetos de t y m son polilogarítmicos. Por lo tanto se usa un wavelet tree multi-ario [9] para representar t y m , obteniendo tiempos de respuesta $O(1)$ para *rank*, *select* y *access*. El uso de espacio es $nH_0(t) + o(n)$ bits para t , $O\left(\frac{n \log \log n}{\log n}\right) H_0(s) = o(n)H_0(s)$ bits para m . Para s_l , usando la estructura de Golynski et al. se obtiene un uso de espacio de $n_l \log \sigma_l + O\left(\frac{n_l \log \sigma_l}{\log \log \log n}\right)$ bits por partición, soportando la operación *select* dentro de los s_l en tiempo $O(1)$, mientras que *rank* y *access* en tiempo $O(\log \log \sigma)$ también dentro de los s_l .

Considerándolo todo, el espacio usado es $nH_0(s) + o(n)(H_0(s) + 1)$ bits, con tiempo de respuesta para *select* de $O(1)$, mientras que para *rank* y *access* sobre el string original son soportadas en tiempo $O(\log \log(\sigma))$.

3. Una implementación práctica para *rank* y *select* basada en Particionamiento del alfabeto

La estrategia de particionamiento del alfabeto fue pensada originalmente para mejorar la velocidad de descompresión [18]. Barbay et al. [4] demostraron que también puede ser usada como estructura de soporte para consultas *rank* y *select* en cadenas de texto arbitrarias, logrando ser uno de los acercamientos más competitivos en la práctica. A continuación se presentará una alternativa a la estrategia de particionamiento del alfabeto que sacrifica eficiencia en consultas *access* para ganar eficiencia en consultas *rank* y *select*, las que suelen ser más importantes que la primera en muchas aplicaciones.

3.1. Definición de la Estructura de Datos

En la propuesta original, descrita en la Sección 2.4, la estructura t es representada con un wavelet tree multi-ario, capaz de responder consultas *rank*, *select* y *access* en tiempo $O(1)$, dado que t tiene alfabeto de tamaño $O(\text{polylog}(n))$. Sin embargo, no hay una implementación eficiente de wavelet trees multi-arios en la práctica. Esto se hace evidente al comprobar que Barbay et al. [4] usan un wavelet tree en sus experimentos, y en la biblioteca *sds1* se usa por defecto un *Huffman-shaped wavelet tree* para t . La solución que será presentada a continuación elimina la necesidad de un wavelet tree multi-ario global para todas las particiones, por una representación que se distribuye entre las particiones.

Sea $s[1..n]$ la secuencia de símbolos a codificar, con un alfabeto Σ de tamaño σ . Las estructuras a utilizar para representarlo de manera eficiente se presentan a continuación:

1. Una estructura de mapeo de símbolos $m[1..\sigma]$, dada por una función f cuya función es particionar el alfabeto y distribuir la secuencia original s en subsecuencias s_l , donde s_l contiene la subsecuencia conformada por los símbolos a de la secuencia original que cumplan $f(a) = l$.
2. Vectores de bit $B_l[1..n]$, uno para cada partición, donde $B_l[i] = 1$ si $m[s[i]] = l$.

3. Las subsecuencias s_l de símbolos de cada subalfabeto. Éstas están dadas por:

$$s_\ell[B_\ell.rank_1(i)] = m.rank_\ell(s[i]) \quad (18)$$

De esta forma entonces podemos implementar las operaciones rank y select de la siguiente forma:

$$s.rank_a(i) \equiv s_l.rank_c(k), \text{ con } l = m.access(a), c = m.rank_l(a), k = B_l.rank(i). \quad (19)$$

$$s.select_a(i) \equiv B_l.select(k), \text{ con } l = m.access(a), c = m.rank_l(a), k = s_l.select_c(i). \quad (20)$$

Desafortunadamente no es posible implementar eficientemente $s.access(i)$ debido a que no sabemos que símbolo está en $s[i]$ ni la partición ℓ tal que $B_\ell[i] = 1$. La alternativa que existe es realizar una búsqueda lineal entre todos los B_j hasta encontrar el que cumpla $B_j[i] = 1$. Una vez encontrada la partición a la que corresponde el símbolo en $s[i]$, entonces podemos computar $s.access(i)$ de la siguiente manera:

$$s.access(i) \equiv m.select_\ell(s_\ell.access(k)), \text{ con } \ell \text{ tal que } B_\ell[i] = 1, k = B_\ell.rank(i). \quad (21)$$

Es importante recordar que existen aplicaciones relevantes que no requieren la consulta *access* para operar, como por ejemplo para calcular la intersección de listas invertidas, calcular las posiciones de términos para búsqueda de frases y funciones de ranking posicional. Además, muchas aplicaciones necesitan la consulta *access* no solo para extraer un símbolo, sino que un fragmento de largo L desde una posición i como $s[i..i + L - 1]$. En la Sección 4.7 se mostrará como se puede implementar esto de manera eficiente.

Llamaremos a esta implementación *Arroyuelo-Sepúlveda Alphabet Partitioning* (ASAP).

Estudiaremos cómo afecta la elección de estructuras para m , B_l y s_l , además del algo-

ritmo usado para particionar el alfabeto.

Como alternativas para m y s_l tenemos los wavelet trees presentados en el Estado del Arte, mientras que para B_l se determinó usar *SDArray* por sobre *RRR* debido a que, pese a que en teoría *RRR* parecía ser un buen candidato como alternativa, en la práctica tuvo que ser descartado porque la cantidad de estructuras de soporte anexas que requiere lo hicieron inviable para su uso con *ASAP* si lo comparamos con *SDArray*. Recordando, *SDArray* usa espacio $m \log \frac{n}{m} + 2m + o(m)$, con m = número de $1s$ y *RRR* usa espacio $nH_0(B) + o(n)$. En ambos casos el término relevante es el sublineal $o(x)$. Considerando p particiones, para *SDArray* se tiene $\sum_{i=0}^p o(m_i) = o(n)$ dado que $\sum_{i=0}^p m_i = n$, mientras que para *RRR* se tiene $p \sum_{i=0}^p o(n)$; éste término incurre en p veces más espacio en *RRR* que en *SDArray*.

Como algoritmo de particionamiento se considerarán tres opciones. La primera consiste en la propuesta por Barbay, Claude y Gagie [4] en su *Alphabet Partitioning*, que como se dijo anteriormente llamaremos **sparse** y que está descrita en la Ecuación (12). Como segunda alternativa se tomará **dense**, descrita en la Sección 2.4.1. Por último se considerará **dense** usando el parámetro $l_{min} = 23$ (l_{min} también fué descrito en la Sección 2.4.1), que llamaremos **dense-23**. El efecto de l_{min} en *ASAP* es que a cada uno de los l_{min} símbolos más frecuentes se les asigna un subalfabeto propio, siendo representados directamente en su propio vector de bits B_l , eliminándose la necesidad de tener el correspondiente s_l .

3.2. Ejemplo

Para hacer la analogía con el ejemplo presentado en la Sección 2.4.2 ahora se presentará el mismo ejemplo,

"to_be_or_not_to_be_that_is_the_question",

en *ASAP*. Tanto m como los s_l se mantienen y son los mismos que en *AP*. El cambio a realizar es sustituir t por los respectivos B_l por partición. En las Figuras 6, 7, 8 y 9 se muestran los B_l resultantes.

CAPÍTULO 3 : UNA IMPLEMENTACIÓN PRÁCTICA PARA RANK Y SELECT BASADA EN PARTICIONAMIENTO DEL ALFABETO

t o _ b e _ o r _ n o t _ t o _ b e _ t h a t _ i s _ t h e _ q u e s t i o n

0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 6: Representación de B_0 para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

t o _ b e _ o r _ n o t _ t o _ b e _ t h a t _ i s _ t h e _ q u e s t i o n

1	1	0	0	0	0	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 7: Representación de B_1 para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

t o _ b e _ o r _ n o t _ t o _ b e _ t h a t _ i s _ t h e _ q u e s t i o n

0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0	1	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 8: Representación de B_2 para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

t o _ b e _ o r _ n o t _ t o _ b e _ t h a t _ i s _ t h e _ q u e s t i o n

0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 9: Representación de B_3 para el string de ejemplo "to_be_or_not_to_be_that_is_the_question".

3.3. Uso de espacio y tiempos de respuesta

Usando *SDArray*, el espacio total ocupado por los vectores de bit B_l es $\sum_{i=1}^{p-1} \left(n_i \log \frac{n}{n_i} + 2n_i + o(n_i) \right)$. Notar que $\sum_{i=1}^{p-1} n_i \log \frac{n}{n_i} = nH_0(t)$, de acuerdo a las Ecuaciones (1) y (13). Además tenemos que $2 \sum_{i=1}^{p-1} n_i = 2n$. Finalmente, para $\sum_{i=1}^{p-1} o(n_i)$ tenemos que cada término en la suma es $O(n_i / \log n_i)$ [14]. En el peor caso se tiene que cada partición tiene $n_i = n/p$ símbolos. Por lo tanto $n_i / \log n_i = n / (p \log \frac{n}{p})$, lo que para p particiones suma un total de espacio de $O(n / \log \frac{n}{p})$ bits. Esto es $o(n)$ siempre que $\log \frac{n}{p} \in \omega(1)$. En este caso, $p \leq \log^2 n$, por lo tanto $\sum_{i=1}^{p-1} o(n_i) \in o(n)$.

Resumiendo, los vectores de bits B_l requieren $n(H_0(t) + 2 + o(1))$ bits de espacio. Son 2 bits extras de espacio por símbolo al compararlo con el vector t de la estrategia usada por Barbay et al. [4]. La estructura de datos completa usa $nH_0(s) + 2n + o(n)(H_0(s) + 1)$ bits.

En cuanto a los tiempos de respuesta, *s.select* es soportada en tiempo $O(1)$ (usando *SDArray*). La consulta *s.rank* es soportada en tiempo $O(\log n)$ en el peor caso: si $n_i = O(\sqrt{n})$, la consulta $B_i.rank$ toma tiempo $O(\log \frac{n}{n_i}) = O(\log n)$.

Respecto al tiempo de construcción, los B_j pueden ser construidos en tiempo lineal, iterando sobre el string original de izquierda a derecha. Para cada símbolo $s[i]$, se determina su partición j , añadiéndose al correspondiente s_j el símbolo $m.rank_j(s[i])$. A su vez se añade a un vector auxiliar aux_j la posición i . Luego los *SDArray* B_j son construidos a partir de aux_j .

4. Resultados Experimentales

En esta sección se darán primeramente todas las consideraciones que se tuvieron en cuenta antes y durante la ejecución de los experimentos, desde el hardware hasta la cadena de texto que fue representada, y a su vez se caracterizarán las consultas que fueron utilizadas en los experimentos. Luego se presentarán los resultados de los experimentos.

4.1. Hardware empleado

Los experimentos fueron ejecutados en un servidor HP Proliant con Intel(R) Xeon(R) CPU E5-2630 2.30GHz de 6 núcleos, 15 MB de caché y 48 GB de RAM.

4.2. Código

El código fuente fue compilado usando los flags `-std=c++11` y `-O3`. El mismo puede ser descargado desde:

<https://github.com/ericksepulveda/asap>.

La implementación se basa en la biblioteca `sdsl-lite` de Simon Gog, que puede ser encontrada en:

<https://github.com/simongog/sdsl-lite>.

4.3. Texto representado

El texto s que se usó para realizar los experimentos es un extracto de la Wikipedia obtenido de un dump de Agosto de 2016. Los tags XML fueron removidos dejando sólo el texto, resultando 3.0GB.

Tiene $\sigma = 8468328$ palabras distintas, las que fueron representadas con enteros de 32 bit (`uint32_t`) usando en total 1.9 GB. La entropía de orden cero para este texto es

12.45 bits y considerando que $\log \sigma \approx 23$, podemos decir que el texto es relativamente compresible.

4.4. Resultados

Los experimentos, como se dijo, se basan en la biblioteca `sdsl-lite`. A continuación se presenta una lista con la forma:

<Etiqueta usada> [<clase_sdsl_usada>]: <Descripción>

- AP [clase `wt_ap<>` en `sdsl`]: *Alphabet Partitioning* usando *Huffman shaped wavelet tree* tanto en m como en t , y *Wavelet Matrix* en los s_l .
- BLCD-WT [clase `wt_blcd<>` en `sdsl`]: *Wavelet Tree balanceado*.
- GMR [clase `wt_gmr<>` en `sdsl`]: Representación de *Golynski, Munro, Rao*.
- HUFF-WT [clase `wt_huff_int<>` en `sdsl`]: *Huffman shaped wavelet tree* con vectores de bits planos (sin compresión).
- HUFF-RRR-WT [clase `wt_huff_int<rrr_vector>` en `sdsl`]: *Huffman-shaped wavelet tree* con vectores de bits de *Raman, Raman y Rao*.
- RRR WT [clase `wt_int<rrr_vector>` en `sdsl`]: *Wavelet Tree* con vectores de bits de *Raman, Raman y Rao*.
- WM [clase `wm_int<>` en `sdsl`]: *Wavelet Matrix*.
- ASAP gmr-wm (D): ASAP con *Golynski, Munro, Rao* en los s_l y *Wavelet Matrix* en m usando particionamiento **dense**.
- ASAP gmr-wm (D 23): ASAP con *Golynski, Munro, Rao* en los s_l y *Wavelet Matrix* en m usando particionamiento **dense-23**.
- ASAP wm-ap (S): ASAP con *Wavelet Matrix* en los s_l y *Alphabet partitioning* en m usando particionamiento **sparse**.

- ASAP wm-huff_int (D 23): *ASAP* con *Wavelet Matrix* en los s_l y *Huffman shaped wavelet tree* en m usando particionamiento **dense-23**.
- ASAP wm-wm (S): *ASAP* con *Wavelet Matrix* en los s_l y en m usando particionamiento **sparse**.

Como se dijo anteriormente, en todas las implementaciones de *ASAP* se usó *SDArray* para representar los B_l .

4.5. Experimentos con Rank y Select

Tanto el particionamiento **dense** como los *Huffman-shaped wavelet trees* tienden a favorecer las consultas que están relacionadas con los símbolos más frecuentes, que en ambos casos tienen representación más corta, y por lo tanto las operaciones sobre ellos son más rápidas que para el resto de los símbolos.

Sin embargo, las consultas con los símbolos más frecuentes suelen ser las de menor interés. Los símbolos menos frecuentes son más discriminantes en consultas a motores de búsqueda, por ejemplo, por lo que entregan mejores resultados y son más importantes. Teniendo esto presente, los experimentos que envuelven a las consultas *Rank* y *Select* fueron llevados a cabo en dos modalidades: con consultas aleatorias y con consultas generadas a partir de un *query log*, con el fin de esclarecer si existe alguna diferencia notable.

Para ejecutar consultas a partir de un *query log* se utilizó el set de consultas TREC 2007 Million Query Track¹. A este archivo se le realizó una limpieza que consistió en remover las *stopwords* y también las palabras que no se encontraban en el alfabeto del string s . Se obtuvieron **29.711** palabras no necesariamente únicas, las que se usaron como consultas.

Para *rank*, al utilizar el *query log* se tomó cada palabra de éste y se generó un valor i aleatorio que se encontrara en los rangos válidos $[0..n - 1]$. Para consultas aleatorias,

¹<http://trec.nist.gov/data/million.query/07/07-million-query-topics.1-10000.gz>

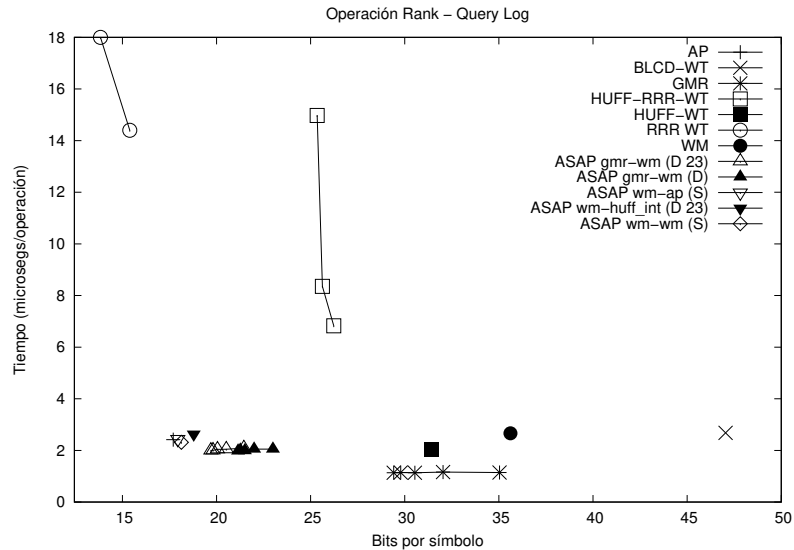


Figura 10: Resultados experimentales para la operacion rank con consultas desde un query log. El eje x empieza en $H_0(s) = 12,45$ bits.

ambos c e i fueron generados aleatoriamente con la misma consideración de validez. Para *select* se utilizó el mismo procedimiento, cambiando la condición de validez a que $i \leq n_c$.

Los resultados se muestran en las Figuras 10, 11, 12 y 13.

Como puede observarse, la estructura que usa menor espacio es *RRR WT*, pero con un costo en tiempo considerable. Vale la pena destacar, por ejemplo, que en *select* con consultas aleatorias, *ASAP gmr-wm (D 23)* usa 1.11 veces el espacio usado por *AP*, y reduce el tiempo promedio tomado por operación en un 79,50 % (de 9.37 a 1.92 microsegundos por operación). *ASAP wm-ap (S)* se vuelve más competitivo en *select* con consultas desde el *query log* usando solo 1.01 veces el espacio de *AP*, reduciendo el tiempo promedio por consulta en un 38,80 %.

Para *rank* las mejoras en tiempos de respuesta son algo menores, encontrándose en los rangos que van desde un 4,78 % y un 17,34 %. Esto puede deberse a que las operaciones de *rank* en los B_l no son tan eficientes como para *select*.

CAPÍTULO 4 : RESULTADOS EXPERIMENTALES

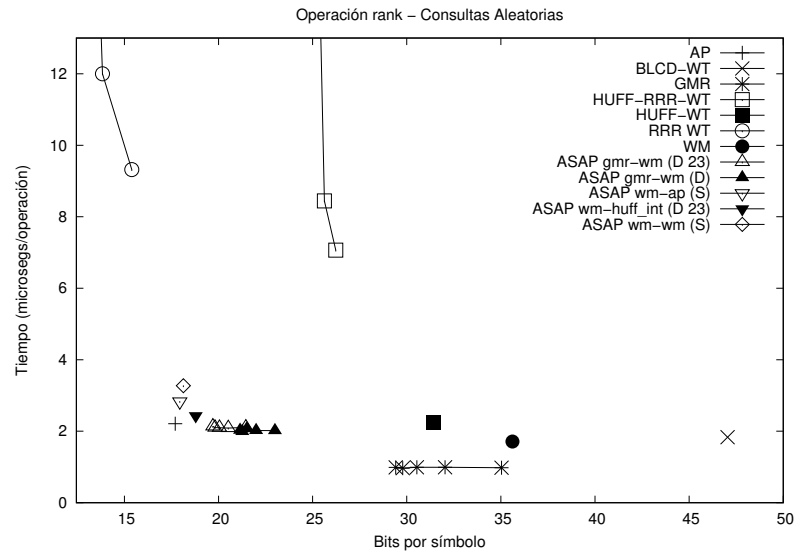


Figura 11: Resultados experimentales para la operacion rank con consultas aleatorias. El eje x empieza en $H_0(s) = 12,45$ bits.

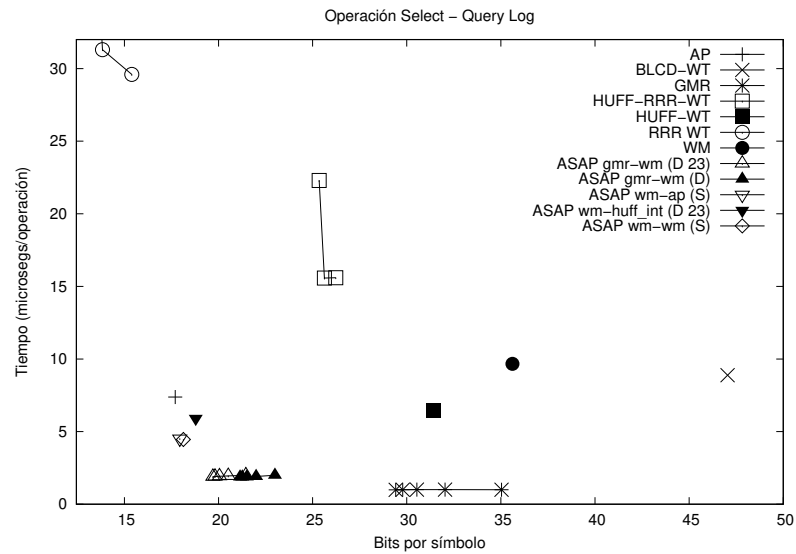


Figura 12: Resultados experimentales para la operacion select con consultas desde un querylog. El eje x empieza en $H_0(s) = 12,45$ bits.

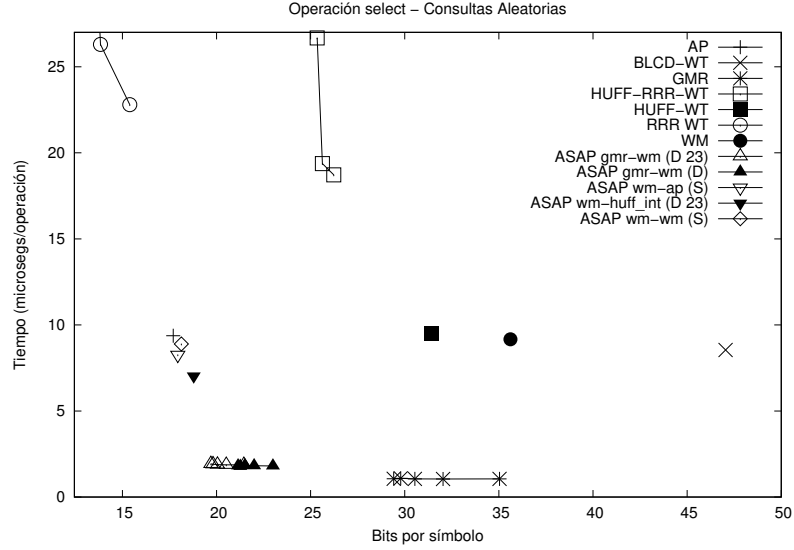


Figura 13: Resultados experimentales para la operación select con consultas aleatorias. El eje x empieza en $H_0(s) = 12,45$ bits.

4.6. Experimentos con Access

Las mejoras en *rank* y *select* tienen un costo. En la Figura 14 se presentan los resultados para el tiempo promedio de respuesta a consultas *access* en posiciones aleatorias dentro de s .

Como se esperaba, *ASAP* no es competitivo para *access*, comparado con *AP*. Si recordamos la Ecuación (21), primero es necesario obtener la partición ℓ , tal que $B_\ell[i] = 1$. Para obtener el bit vector que cumple esta condición, dadas las estructuras que componen *ASAP*, no queda otra alternativa más que recorrer los vectores de bits hasta encontrar el que cumpla dicha condición. Este proceso es particularmente ineficiente, y es un factor que no puede ser olvidado al considerar el rendimiento de *ASAP* durante consultas de tipo *access*. De todas maneras, *ASAP* se comporta mejor que *RRR WT* y ofrece trade-offs que pueden ser de interés contra *GMR*.

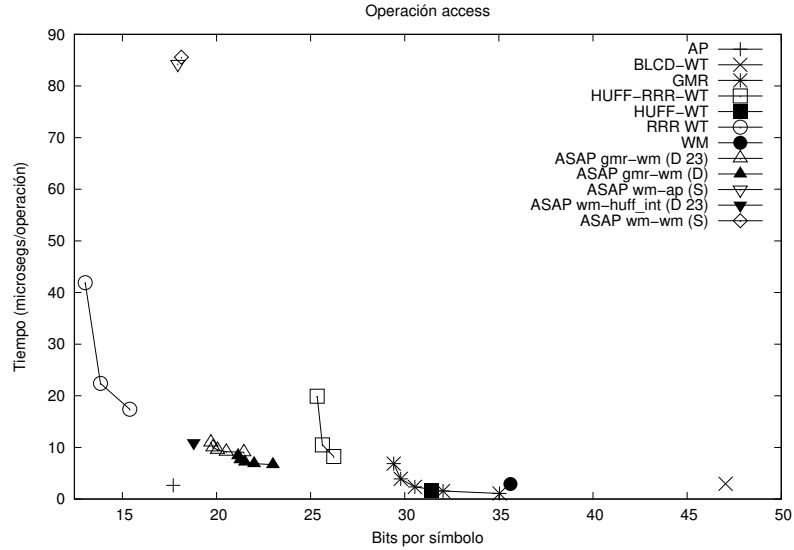


Figura 14: Resultados experimentales para la operacion access. El eje x empieza en $H_0(s) = 12,45$ bits.

4.7. Aplicación 1: Extracción de fragmentos en bases de datos textuales

Como primera aplicación práctica para *ASAP* se comprobó su rendimiento en la extracción de fragmentos desde un texto. Esta operación será definida como *snippet*. Así, para denotar que se extraerá un fragmento de largo L a partir de la posición i desde el texto s se escribirá $s.snippet(i, L)$.

Si bien es cierto que sería posible realizar esta operación realizando L operaciones *access* consecutivos a partir de la posición i , no se estaría aprovechando la estructura inherente de *ASAP*. Recordando el por qué la operación *access* obtiene mal rendimiento, una de las razones es el hecho de que requiere realizar chequeos en cada uno de los B_l hasta encontrar el que tenga un 1 en la posición i (para $s.access(i)$). Repetir esto para extraer todos los símbolos de un fragmento es ineficiente y redundante en esfuerzo.

Si se considerara hacer esto para un fragmento, se realizarían muchas consultas en los

B_l solo para obtener los l tal que $B_l = 1$.

Como alternativa a esto se propone el Algoritmo 1.

Algorithm 1 snippet(i, L)

```

1: Sea  $S[1..L]$  un arreglo de símbolos en  $\Sigma$ 
2: for  $j = 0$  to  $p - 1$  do
3:    $cur \leftarrow B_j.rank(i - 1)$ 
4:    $count \leftarrow B_j.rank(i + L - 1) - cur$ 
5:   for  $k = 1$  to  $count$  do
6:      $cur \leftarrow cur + 1$ 
7:      $S[B_j.select(cur) - i + 1] \leftarrow m.select_j(s_j.access(cur))$ 
8:   end for
9: end for

```

El algoritmo de extracción de fragmentos toma tiempo $O\left(\sum_{i=0}^{p-1} \log \frac{n}{n_i} + L \log \log \sigma\right)$. La suma $\sum_{i=0}^{p-1} \log \frac{n}{n_i}$ se maximiza cuando $n_i = n/p = n/\log^2 n$. Por lo tanto, $\sum_{i=0}^{p-1} \log \frac{n}{n_i} = \log^2 n \cdot \log \log n$, lo que es $O(\log^2 n \cdot \log \log n + L \log \log \sigma)$. Por lo tanto, para fragmentos lo suficientemente largos, este algoritmo es más rápido que usar *access*.

En las Figuras 15, 16, 17, 18 y 19 se muestran los resultados obtenidos para fragmentos de 100, 200, 300, 500 y 1000 símbolos de largo.

Los tiempos bajan desde alrededor de 10 microsegs por símbolo extraído para *access* a alrededor de 3 microsegs para la operación snippet.

4.8. Aplicación 2: Intersección de listas invertidas en Recuperación de la Información

En el Algoritmo 2 se muestra el pseudocódigo del algoritmo utilizado para calcular los textos en los que ocurre la intersección de un grupo de símbolos $Q[c_1..c_q]$.

En la Figura 20 se muestra el gráfico con los resultados de correr dicho algoritmo con distintas estructuras. El texto fue el mismo prefijo de la Wikipedia usado anteriormente. El *query log* también se mantuvo, pero fue interpretado con grupos de símbolos (separados por líneas) de manera de poder realizar intersecciones. De todas formas se repitió

CAPÍTULO 4 : RESULTADOS EXPERIMENTALES

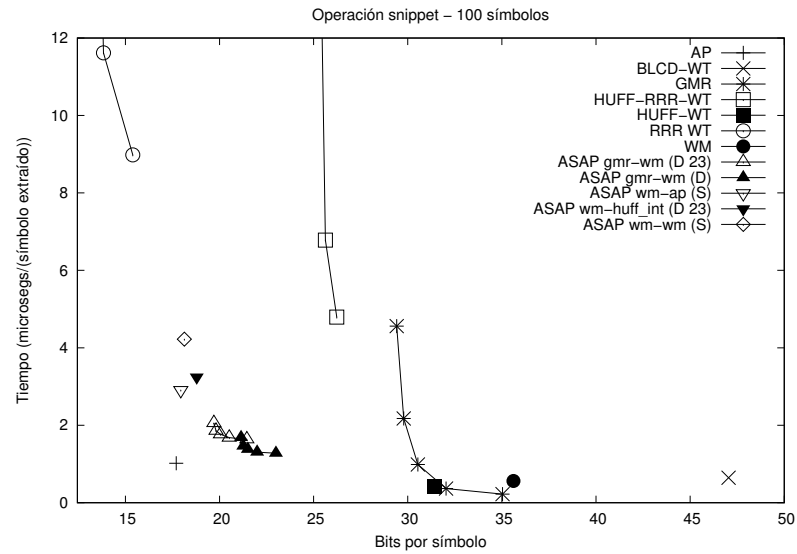


Figura 15: Resultados experimentales para extracción de fragmentos con largo 100. El eje x empieza en $H_0(s) = 12,45$ bits.

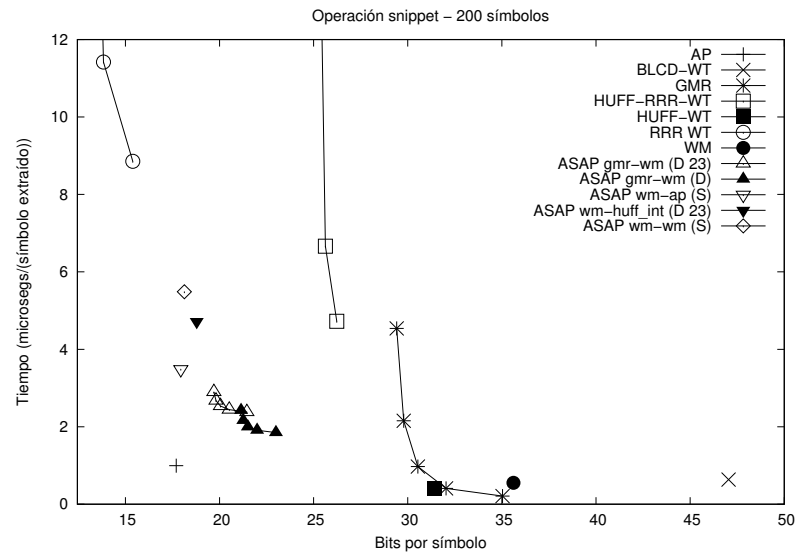


Figura 16: Resultados experimentales para extracción de fragmentos con largo 200. El eje x empieza en $H_0(s) = 12,45$ bits.

CAPÍTULO 4 : RESULTADOS EXPERIMENTALES

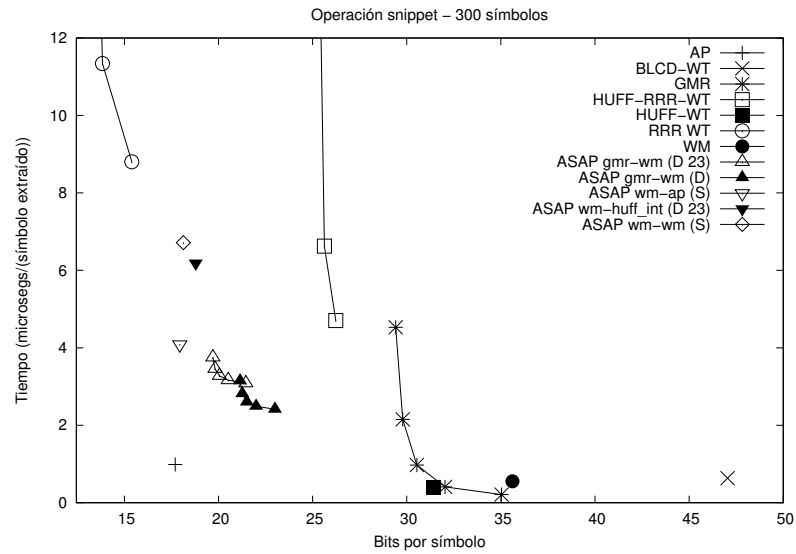


Figura 17: Resultados experimentales para extracción de fragmentos con largo 300. El eje x empieza en $H_0(s) = 12,45$ bits.

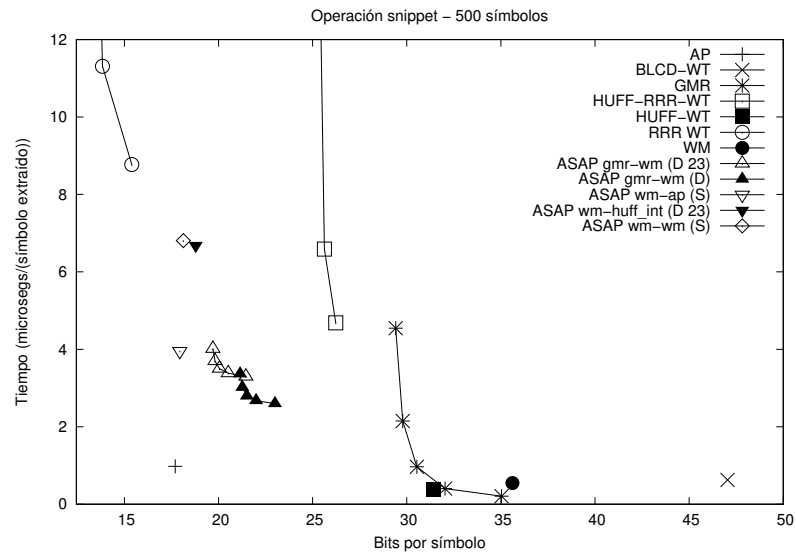


Figura 18: Resultados experimentales para extracción de fragmentos con largo 500. El eje x empieza en $H_0(s) = 12,45$ bits.

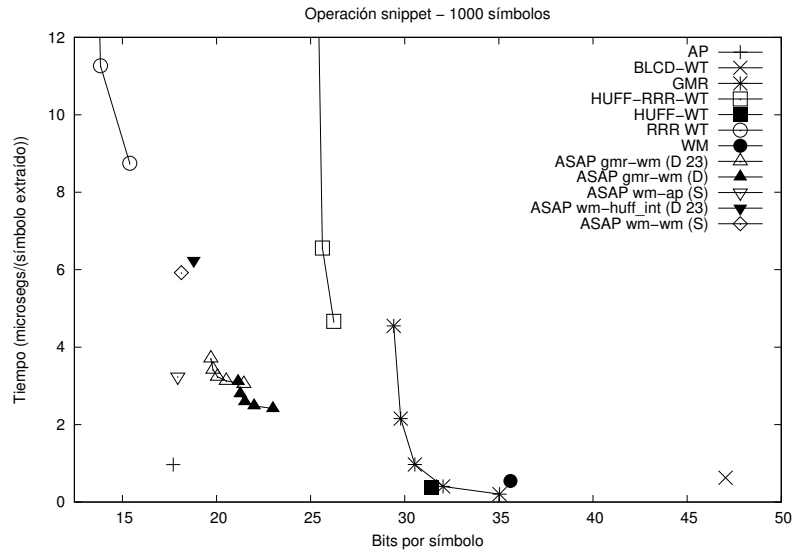


Figura 19: Resultados experimentales para extracción de fragmentos con largo 1000. El eje x empieza en $H_0(s) = 12,45$ bits.

el proceso de eliminado de *stopwords* y de símbolos no presentes en s .

Como puede verse, ASAP se muestra competitivo en esta aplicación: usando solo un 2 % más de espacio, ASAP *wm-wm (S)* es capaz de reducir el tiempo de intersección en un 60,67 %

4.9. Aplicación 3: Computación distribuida de rank y select

Las estrategias de particionamiento del alfabeto dan pie a la computación distribuida de lotes de consultas de *rank* y *select*. En esta sección se explicará el cómo, las consideraciones y los resultados obtenidos de los experimentos.

En sistemas distribuidos en general puede existir un nodo *broker* encargado de enrutar y distribuir las consultas entre los nodos *worker* (procesadores), los cuales realizan el cómputo de la consulta para entregar el resultado esperado. A continuación se estudiará cómo las estrategias de particionamiento del alfabeto pueden ser adaptadas a un ambiente distribuido.

Algorithm 2 intersect inverted list

```

1: Sea  $Q[c_1..c_q]$  un arreglo de símbolos en  $\Sigma$  ordenados ascendentemente por su
   frecuencia  $n_{c_i}$ 
2: Sea  $N$  el número de textos en la colección
3: Sea  $D$  el símbolo que marca el inicio de un texto
4: Sea  $p = s.rank_D(s.select_{Q[0]}(1))$ 
5: Sea  $Ans = []$ 
6: while There is a text left to check do
7:    $textStart \leftarrow s.select_D(p)$ 
8:    $textEnd \leftarrow s.select_D(p + 1)$ 
9:    $intersects \leftarrow \mathbf{true}$ 
10:  for  $c_q$  in  $Q$  do
11:     $beforeText \leftarrow s.rank_{c_q}(textStart)$ 
12:     $untilText \leftarrow s.rank_{c_q}(textEnd)$ 
13:    if  $beforeText = untilText$  then
14:       $intersects \leftarrow \mathbf{false}$ 
15:      break
16:    end if
17:  end for
18:  if  $intersects = \mathbf{true}$  then
19:     $Ans \leftarrow p$ 
20:  end if
21:  if There is still a text to check then
22:     $covered \leftarrow s.rank_{Q[0]}(s.select_D(p + 1))$ 
23:     $p \leftarrow s.rank_D(s.select_{Q[0]}(covered + 1))$ 
24:  else
25:    break
26:  end if
27: end while

```

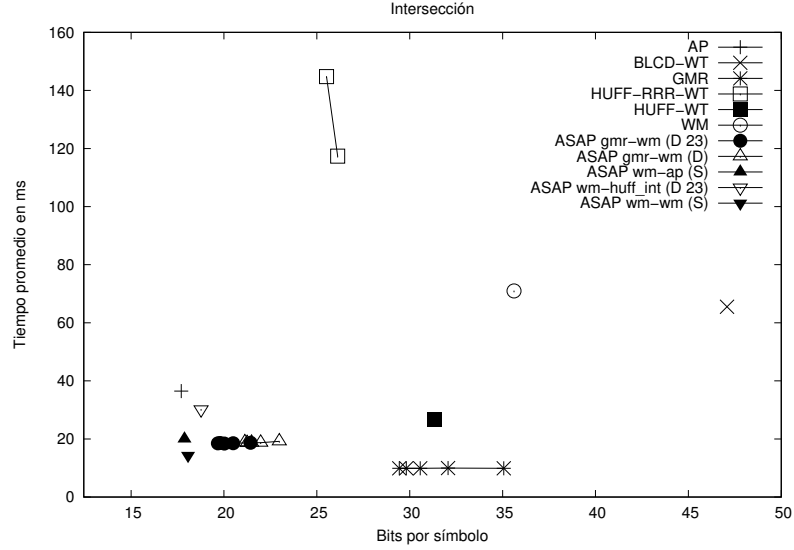


Figura 20: Resultados experimentales para la intersección de listas invertidas

Computación distribuida de consultas usando AP. Las secuencias que contienen a los subalfabetos s_l son distribuidas entre los nodos *worker*, por lo tanto tenemos p procesadores. El *broker* en este caso debe almacenar tanto a m como a t . Esto es una desventaja para este caso, dado que estas estructuras actuarán como cuello de botella al intentar distribuir las consultas.

Computación distribuida de consultas usando ASAP. Para este caso las secuencias que contienen los subalfabetos s_l y los vectores de bits B_l son distribuidos entre los nodos *worker*. A diferencia del acercamiento usado con AP, aquí cada nodo actúa como *broker*, necesitando sólo replicar m en cada partición. El espacio usado para esto es $O(p\sigma \log \log p)$ usando un wavelet tree sin comprimir para m . Esto, traduciéndolo a espacio por nodo de computación, son sólo $O(\sigma \log \log p) = o(n)H_0(s)bits$. De esta manera se evita tener un único *broker*, distribuyendo el trabajo de repartir la carga también entre los nodos *worker*, logrando así evitar la creación de cuellos de botella. Esto hace al sistema más resiliente a fallas e incluso le abre las puertas a su uso en un

ambiente *P2P*.

Las consultas llegan a un nodo *worker*, donde usando m se evalúa a qué nodo *worker* le corresponde ser distribuida para poder resolver su requerimiento. Para $s.access(i)$ se realiza un *broadcast* para determinar el nodo ℓ que cumple con $B_\ell[i] = 1$, siendo éste el que deba computar y responder la consulta. Para la extracción de fragmentos también se genera inicialmente un *broadcast*, de manera que colectivamente se construya la respuesta.

Comparación. La principal desventaja de *AP* es que para ser aplicado distribuidamente necesita un nodo *broker* donde estén m y t , provocando un cuello de botella y reduciendo la eficiencia del sistema. Con *ASAP* no existe dicho problema, dado que lo único que es necesario replicar en cada nodo es m , que en la práctica es pequeño. Para usar la misma estrategia con *AP*, sería necesario replicar en cada nodo no sólo m , sino que también t , incurriendo en un uso de memoria considerablemente mayor. Si consideramos un sistema en el que la memoria es un recurso escaso, *ASAP* será capaz de representar textos más grandes.

En la Tabla 2 se muestran los resultados obtenidos con la simulación de estos esquemas. Solo se consideró tiempo de procesamiento, dejando de lado el tiempo de comunicación y transferencia. Es posible observar que *ASAP* le saca mayor provecho al esquema distribuido. Los tiempos promedios por operación para *rank* y *select* se ven reducidos en 71 % y 76 % respectivamente, comparados con *AP*. Para la extracción de fragmentos (*snippet*) el tiempo usado por símbolo extraído se reduce en un 50 %.

*Tabla 2: Resultados experimentas para la implementación de AP y ASAP en ambiente distribuido Los tiempos están en promedio de microsegundos por operación (para snippet es promedio de microsegundos por símbolo extraído). En este caso ASAP usa wavelet matrix en los s_l y Huffman shaped wavelet tree en m . El particionamiento es **dense-23**, lo que generó 46 particiones (número de nodos)*

Operation	ASAP		AP		AP/ASAP
	Tiempo	Speedup	Tiempo	Speedup	
rank	0,373	8,03	1,310	1,91	3,51
select	0,706	8,41	2,970	2,55	4,21
access	1,390	8,11	2,130	1,45	1,53
snippet	0,466	6,96	0,939	1,25	2,02

5. Conclusiones y trabajo futuro

La implementación práctica propuesta para consultas *rank* y *select* basada en particionamiento del alfabeto (*ASAP*) ofrece *trade-offs* interesantes. Usando poco espacio más que la estructura original que implementa particionamiento del alfabeto (*AP*) presentada en [4], es capaz de reducir el tiempo utilizado por la consulta *select* en aproximadamente un 80 %, mientras que para la consulta *rank* las mejoras en tiempo de respuesta son entre un 4 % y un 17 %. Para el problema de intersección de listas invertidas, se mostraron mejoras de alrededor de un 60 % en el tiempo usado por consulta, usando solo 2 % de espacio adicional al compararse con *AP*. Esto hace a este tipo de estructura más atractiva para esta aplicación en particular.

Se estudió también como *ASAP* puede ser usado para el cómputo de las consultas *rank*, *select*, *access* y *snippet*. Hasta donde se sabe, este es el primer estudio sobre cómo soportar estas consultas en un ambiente distribuido. En los experimentos mostrados *ASAP* obtuvo *speedups* (ratio entre el tiempo utilizado por consulta en ambiente no-distribuido y tiempo utilizado por consulta en ambiente distribuido) entre 6,96 y 8,41, con 46 procesadores. Los *speedups* obtenidos por *AP* estuvieron en el rango 1,25 – 2,55 en el mismo escenario. Estos resultados fueron obtenidos simulando un ambiente de cómputo distribuido, sólo considerando el tiempo de computo y dejando de lado el tiempo de comunicación. Los resultados observados en estos experimentos permiten pensar en un ambiente distribuido real.

La implementación de *ASAP* se encuentra disponible para su uso libre en `GitHub`², con sus respectivas instrucciones de uso.

Respecto a la librería *sds1*, la herramienta fué usada extensiva e intensivamente durante la ejecución e implementación de los experimentos. El hecho de que estuviera documentada más que nada a través de papers y el mismo código puede ser una desventaja para principiantes en la lectura y entendimiento de papers, pero dado que el universo de las estructuras sucintas es técnico por naturaleza, parece una elección *ad-hoc* para

²<https://github.com/ericksepulveda/asap>

CAPÍTULO 5 : CONCLUSIONES Y TRABAJO FUTURO

usuarios frecuentes. Se espera que luego de la publicación de los resultados expuestos en este trabajo, la estructura *ASAP* pueda ser incluida en la librería *sds1* para que así pueda ser aprovechada y evaluada por más público. Ya existe un canal de comunicación con Simon Gog (autor de la *sds1*), y él se encuentra enterado del trabajo que se realizó. Como trabajo futuro se propone la implementar y probar esta estructura en un ambiente distribuido real, junto con un estudio más en profundidad que incluya aspectos como cómo balancear la carga, evaluar otras alternativas para los vectores de bits B_i , de manera de capturar regularidades y mejorar la compresión (usando *gap encoding*, por ejemplo [16]), y poner a prueba el rendimiento de esta estructura en otras aplicaciones como árboles cardinales, autoíndices comprimidos para texto, índices posicionales para recuperación de la información, compresión de símbolos en el parsing LZ78 de un texto, entre otras.

Referencias

- [1] Diego Arroyuelo, Senén González y Mauricio Oyarzún. “Compressed Self-indices Supporting Conjunctive Queries on Document Collections”. En: *Proceedings of the 17th International Conference on String Processing and Information Retrieval*. SPIRE’10. Los Cabos, Mexico: Springer-Verlag, 2010, págs. 43-54. ISBN: 3-642-16320-3, 978-3-642-16320-3. URL: <http://dl.acm.org/citation.cfm?id=1928328.1928335>.
- [2] Diego Arroyuelo y col. “Distributed Search Based on Self-indexed Compressed Text”. En: *Inf. Process. Manage.* 48.5 (sep. de 2012), págs. 819-827. ISSN: 0306-4573. DOI: 10.1016/j.ipm.2011.01.008. URL: <http://dx.doi.org/10.1016/j.ipm.2011.01.008>.
- [3] Diego Arroyuelo y col. “To Index or Not to Index: Time-space Trade-offs in Search Engines with Positional Ranking Functions”. En: *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’12. Portland, Oregon, USA: ACM, 2012, págs. 255-264. ISBN: 978-1-4503-1472-5. DOI: 10.1145/2348283.2348320. URL: <http://doi.acm.org/10.1145/2348283.2348320>.
- [4] Jérémy Barbay y col. “Alphabet Partitioning for Compressed Rank/Select and Applications”. En: *21st Annual International Symposium on Algorithms and Computation (ISAAC)*. Lecture Notes in Computer Science 6507 (part II). Jeju, South Korea: Springer-Verlag, dic. de 2010, págs. 315-326.
- [5] David Richard Clark. “Compact Pat Trees”. UMI Order No. GAXNQ-21335. Tesis doct. Waterloo, Ont., Canada, Canada, 1998.
- [6] Francisco Claude, Gonzalo Navarro y Alberto Ordóñez. “The Wavelet Matrix”. En: *Inf. Syst.* 47.C (ene. de 2015), págs. 15-32. ISSN: 0306-4379. DOI: 10.1016/j.is.2014.06.002. URL: <http://dx.doi.org/10.1016/j.is.2014.06.002>.

- [7] Simon Gog. *Succinct Data Structure Library*. 2017. URL: <https://github.com/simongog/sdsl-lite> (visitado 02-04-2017).
- [8] Alexander Golynski, J. Ian Munro y S. Srinivasa Rao. “Rank/Select Operations on Large Alphabets: A Tool for Text Indexing”. En: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*. SODA '06. Miami, Florida: Society for Industrial y Applied Mathematics, 2006, págs. 368-373. ISBN: 0-89871-605-5. URL: <http://dl.acm.org/citation.cfm?id=1109557.1109599>.
- [9] Alexander Golynski, Rajeev Raman y S. Srinivasa Rao. “On the Redundancy of Succinct Data Structures”. En: *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory*. SWAT '08. Gothenburg, Sweden: Springer-Verlag, 2008, págs. 148-159. ISBN: 978-3-540-69900-2. DOI: 10.1007/978-3-540-69903-3_15. URL: http://dx.doi.org/10.1007/978-3-540-69903-3_15.
- [10] Roberto Grossi, Alessio Orlandi y Rajeev Raman. “Optimal Trade-offs for Succinct String Indexes”. En: *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming*. ICALP'10. Bordeaux, France: Springer-Verlag, 2010, págs. 678-689. ISBN: 3-642-14164-1, 978-3-642-14164-5. URL: <http://dl.acm.org/citation.cfm?id=1880918.1880992>.
- [11] Guy Joseph Jacobson. “Succinct Static Data Structures”. AAI8918056. Tesis doct. Pittsburgh, PA, USA, 1988.
- [12] Veli Mäkinen y Gonzalo Navarro. “Succinct Suffix Arrays Based on Run-length Encoding”. En: *Nordic J. of Computing* 12.1 (mar. de 2005), págs. 40-66. ISSN: 1236-6064. URL: <http://dl.acm.org/citation.cfm?id=1195881.1195885>.
- [13] Gonzalo Navarro. “Wavelet Trees for All”. En: *Combinatorial Pattern Matching: 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*. Ed. por Juha Kärkkäinen y Jens Stoye. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 2-26. ISBN: 978-3-642-31265-6. DOI: 10.1007/978-3-

REFERENCIAS

- 642-31265-6_2. URL: http://dx.doi.org/10.1007/978-3-642-31265-6_2.
- [14] Daisuke Okanohara y Kunihiro Sadakane. “Practical Entropy-Compressed Rank/Select Dictionary”. En: *PROCEEDINGS OF ALENEX’07*, ACM. Press, 2007.
- [15] Daisuke Okanohara y Kunihiro Sadakane. “Practical Entropy-compressed Rank/Select Dictionary”. En: *Proceedings of the Meeting on Algorithm Engineering & Experiments*. New Orleans, Louisiana: Society for Industrial y Applied Mathematics, 2007, págs. 60-70. URL: <http://dl.acm.org/citation.cfm?id=2791188.2791194>.
- [16] Jorge Andrés Dinator Pardo. *Estructuras Comprimidas Para Rank, Select y Next basadas en Gap Encoding*. Memoria UTFSM. 2016.
- [17] Rajeev Raman, Venkatesh Raman y S. Srinivasa Rao. “Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees and Multisets”. En: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’02. San Francisco, California: Society for Industrial y Applied Mathematics, 2002, págs. 233-242. ISBN: 0-89871-513-X. URL: <http://dl.acm.org/citation.cfm?id=545381.545411>.
- [18] A. Said. “Efficient alphabet partitioning algorithms for low-complexity entropy coding”. En: *Data Compression Conference*. Mar. de 2005, págs. 183-192. DOI: 10.1109/DCC.2005.35.