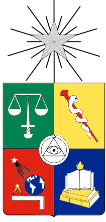


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SOLUCIONES EFICIENTES PARA RANK Y SELECT EN SECUENCIAS BINARIAS

TESIS PARA OPTAR AL GRADO DE GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

ELIANA PAZ PROVIDEL GODOY

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
DIEGO ARROYUELO BILLIARDI
JEREMY BARBAY
BENJAMÍN BUSTOS CÁRDENAS

Este trabajo ha sido parcialmente financiado por proyecto FONDECYT 1-110066 y por el Instituto de Dinámica Celular y Biotecnología

SANTIAGO DE CHILE
DICIEMBRE 2012

Resumen

Las estructuras de datos compactas ofrecen funcionalidad y acceso a los datos usando poco espacio. En una estructura de datos *plana* se conservan los datos en su forma original y se busca minimizar el espacio extra usado para proveer la funcionalidad, mientras que en una estructura *comprimida* además se recodifican los datos para comprimirlos. En esta tesis se estudian estructuras de datos compactas para secuencias de bits (bitmaps) que proveen las operaciones *rank* y *select*: $rank_b(B, i)$ cuenta el número de bits $b \in \{0, 1\}$ en $B[1..i]$ y $select_b(B, i)$ retorna la posición de la i-ésima ocurrencia de b en B.

En teoría ambas consultas se pueden responder en tiempo constante, pero la implementación práctica de estas soluciones no siempre es directa o con buenos resultados empíricos. Las estructuras de datos con un enfoque más práctico, usualmente no óptimas en teoría, pueden tener mejor desempeño que implementaciones directas de soluciones teóricamente óptimas. Esto es particularmente notorio para la operación *select*. Además, las implementaciones más eficientes para *rank* son deficientes para *select*, y viceversa.

En esta tesis se definen nuevas estructuras de datos prácticas para mejorar el desempeño de las operaciones de *rank* y *select*, basadas en dos ideas principales. La primera consiste en, a diferencia de las técnicas actuales, que usan estructuras separadas para *rank* y *select*, reutilizar cada estructura también para acelerar la otra operación. La segunda idea es simular en tiempo de consulta una tabla de resultados precomputados en vez de almacenarla, lo que permite utilizar tablas universales mucho mayores que las que sería posible almacenar.

Los resultados experimentales muestran que la primera idea, aplicada a estructuras planas, utiliza sólo 3 % de espacio sobre el bitmap y ofrece tiempos similares a estructuras que usan mucho más espacio, para ambas operaciones. En estructuras de datos comprimidas se pueden combinar ambas ideas, obteniendo un espacio extra de menos de 7 % sobre el bitmap comprimido y manteniendo, para ambas operaciones, tiempos similares o mejores que las estructuras actuales (que usan 27 % de espacio extra).

A mi amada hija, Amanda Paz

Agradecimientos

Cuando se finaliza una etapa es imposible no mirar hacia atrás y ver todos los obstáculos que se tuvieron que pasar para cumplir un sueño o una meta. Hoy, luego de un largo camino recorrido agradezco primeramente a Dios, por acompañarme y guiarme en en este largo proceso. Muchas veces vi todo oscuro, pero en su infinita sabiduría me mostró el camino a seguir. Agradezco además a muchas personas, y en especial entre ellas agradezco...

A mi hija Amanda, que fue mi principal motivo para cumplir esta meta. Su sonrisa y ternura me motivaron a seguir luchando.

A mi mamita, por apoyarme en las buenas y malas, por hacer que este camino fuese más fácil de caminar. Por incentivarme cuando pensé que nunca lo lograría.

A mi papá, por su amor y compañía. Por preguntarme una y otra vez cómo iba el avance.

A mis hermanas, cuñados y sobrinos, por confiar en mí, apoyarme cada día, y estar ahí siempre que las necesité. Me enseñaron la importancia de la amistad entre hermanas, que puedo confiar en ustedes cuando todo se está derrumbando. Al Isma, por su paciencia, apoyo incondicional y lo más importante por enseñarme a creer nuevamente en las cosas lindas de la vida. A mis tatas y tíos, que confiaron y que permitieron que cada día fuera un avance. A mi querida amiga Lolo, que en algún momento creyó que nunca terminaría la tesis lo que me incentivó a seguir dando la lucha. Gracias por acompañarme en todos esos momentos duros.

Y finalmente, a mi profe guía, que fue un verdadero guía. Admiro su paciencia para explicarme las cosas una y otra vez. Me escuchó cada vez que tuve problemas y alegrías, siempre me ayudó aunque no fuese algo académico, se preocupó por mí y mi hija, motivándome a seguir adelante. Me retó cuando las cosas no funcionaban y me incentivó a mirar hacia adelante cuando era necesario.

Creo que de alguna forma, todos los que he mencionado y muchas otras personas son cómplices de esta tesis, y quizás nunca entendieron lo que estaba haciendo por más que les explicaba la maravilla de investigar. Pero aún así me acompañaron, apoyaron y me ayudaron a ser una mejor persona.

Índice general

1. Introducción **1**
1.1. Organización de la Tesis 3

2. Conceptos Básicos **4**
2.1. Concepto de Entropía 4
2.2. Estructuras de Datos Planas y Comprimidas 5
2.3. Operaciones *rank* y *Select* 5
2.4. Codificación de Números 6
2.4.1. Codificación Gamma 6
2.4.2. Codificación Simple9 7
2.5. FM-index 8
2.5.1. Transformada de Burrows-Wheeler 8
2.5.2. Wavelet Tree 10
2.5.3. Construcción del FM-index 11

3. Trabajo Relacionado **12**
3.1. Estructuras de Datos Planas 12
3.1.1. Solución de Jacobson para *rank* 12
3.1.2. Implementación de González *et al.* 13
3.1.3. Solución de Clark para *Select* 15
3.1.4. Implementación *darray* de Okanohara y Sadakane 16
3.1.5. Implementaciones de Vigna 16
3.1.6. Solución de Golynski *et al.* para Bitmaps Densos 19
3.2. Estructuras de datos Comprimidas 21
3.2.1. Solución de Raman *et al.* 22
3.2.2. Implementación de Claude y Navarro 22
3.2.3. Implementación *sarray* de Okanohara y Sadakane 23
3.2.4. Solución de Golynski *et al.* para Bitmaps Dispersos 23
3.2.5. Compressed Dense Fully-Indexable Dictionaries 25
3.2.6. Solución de Pătraşcu 26

4. Nuevas Implementaciones **28**
4.1. Estructura de Muestreo Combinado 28
4.1.1. Respondiendo Consultas *rank* y *Select* 29
4.2. Select 1 Nivel 34
4.2.1. Creación de la Estructura 34

4.2.2. Consulta *Select* Utilizando la Estructura 34
4.2.3. Select 1 Nivel con Codificación 36
4.3. Estructura para Bitmaps Dispersos 37
4.3.1. Codificación en Tiempo de Construcción 38
4.3.2. Decodificación en Tiempo de Consulta 39
4.3.3. Respondiendo *rank* y *Select* 40
4.3.4. Combinando con el Esquema de Particionamiento Regular en 1s y Select de 1 Nivel 43

5. Resultados Experimentales **45**
5.1. Resultados de Estructura de Muestreo Combinado 45
5.1.1. Consultas Independientes 46
5.1.2. Consultas Mezcladas 48
5.2. Estructura Select 1 Nivel 54
5.3. Estructura Select 1 Nivel con Codificación 55
5.3.1. Utilizando Codificación Gamma 57
5.3.2. Utilizando Codificación Simple9 59
5.3.3. Comparación para Consultas *Select* entre Muestreo Combinado y Select 1 Nivel 59
5.4. Estructuras para Bitmaps Dispersos 62
5.4.1. Consultas Independientes 62
5.4.2. Resultados al Combinar con el Esquema de Particionamiento Regular en 1s para *Select* 65
5.4.3. Comparación Variantes Estructura de Raman *et. al* para Consultas *Select* 69
5.4.4. Integrando la Estructura al FM-index 71

6. Conclusión y Trabajos Futuros **74**

Índice de tablas

5.1. Combinación de largos de bloques y superbloques. 62

Índice de figuras

2.1. Entropía empírica de orden cero de acuerdo a la densidad de la secuencia binaria. 5
2.2. Ejemplo de la transformada de Burrows-Wheeler para el texto original $T\$ = S_1 =$ mississippi\$ y T^{bwt}= ipssm\$pissii. 9

3.1. Ejemplo para responder una consulta *rank* para bloques de tamaño $b = 4$ y superbloques de tamaño $s = 16$. 13
3.2. Algoritmo broadword para contar la cantidad de 1s en x. 17
3.3. Algoritmo broadword para calcular el índice del r-ésimo 1 en x. Si no existe tal bit, el resultado es 72. 18

4.1. Ejemplo de muestreo combinado para una secuencia de bits utilizando $S_s = S_r = 16$. 29
4.2. Ejemplo de consulta *rank* en la estructura de muestreo combinado. 30
4.3. Ejemplo de consulta *select* en la estructura de muestreo combinado. 31
4.4. Algoritmo para responder *rank* utilizando la estructura de muestreo combinado. B se considera como un arreglo de bytes. 32
4.5. Algoritmo para responder *select* utilizando la estructura de muestreo combinado. B se considera como un arreglo de bytes. 33
4.6. Ejemplo de estructura Select 1 Nivel para una secuencia de bits. Se consideran los parámetros $d = 0.4$ y $S = 4$. 35
4.7. Ejemplo de consulta *select* sobre la estructura Select 1 Nivel para una secuencia de bits. Se consideran los parámetros $d = 0.4$ y $S = 4$. 35
4.8. Algoritmo para responder *select* en la estructura con un nivel de directorio. La función *popcount_secuencial*(B, p, k) cuenta en B la cantidad de posiciones utilizando operaciones *popcount* desde la posición p hasta encontrar el k-ésimo 1. A es el arreglo con las respuestas *select* almacenadas en el directorio. *ObtenerBit*(C, b) retorna el valor del b-ésimo bit en C. 36
4.9. Algoritmo para responder *select* en la estructura para *select* con un nivel de directorio usando codificación. *DecodificarRespuesta*(b, k) retorna la k-ésima respuesta relativa almacenada en el bloque b, decodificando y sumando las k primeras diferencias. 37
4.10. Ejemplo de construcción de la estructura *select* de 1 nivel utilizando codificación. En a) se muestra la secuencia original donde solamente se han marcado los bloques densos y dispersos. En b) se muestra la secuencia modificada donde cada bloque disperso almacena sus respuestas codificadas por alguna función `f`. 37

4.11. Algoritmo para codificar en tiempo de construcción el identificador r asignado a un bloque. 39
4.12. Algoritmo para reconstruir el bloque identificado por r para encontrar cuántos 1s hay en ese bloque hasta la posición w relativa al bloque. 40
4.13. Algoritmo para reconstruir el bloque identificado por r para encontrar la posición del w-ésimo 1 relativo al bloque. 41
4.14. Algoritmo para responder una consulta *rank* en la estructura para bitmaps dispersos. $ObtenerIndice(S, ini, end)$ obtiene el identificador r del bloque donde está la posición i, leyendo la tabla S entre las posiciones *ini* y *end*. Recuerde que los valores $\lceil\log\binom{t}{k_c}\rceil$ están precalculados. 42
4.15. Algoritmo para responder una consulta *select* en la estructura para bitmaps dispersos. Recuerde que los valores $\lceil\log\binom{t}{k_c}\rceil$ están precalculados. 42

5.1. Consultas *rank* y *Select* para bitmaps planos con densidad de 10 %. El espacio corresponde al espacio extra sobre el bitmap. 47
5.2. Consultas *rank* y *Select* para bitmaps planos con densidad de 50 %. El espacio corresponde al espacio extra sobre el bitmap. 48
5.3. Consultas *rank* y *Select* para bitmaps planos con densidad de 90 %. El espacio corresponde al espacio extra sobre el bitmap. 49
5.4. Resultado de experimentos de consultas *select/rank* mezcladas para las proporciones *select/rank* de 10 %/90 % y 30 %/70 %. 50
5.5. Resultado de experimentos de consultas *select/rank* mezcladas para las proporciones *select/rank* de 50 %/50 % y 70 %/30 %. 51
5.6. Resultado de experimentos de consultas *select/rank* mezcladas para la proporción *select/rank* de 90 %/10 %. 52
5.7. Relación tiempo y espacio por consulta *select* utilizando el texto Inglés. . . . 53
5.8. Relación tiempo y espacio por consulta *select* utilizando el texto Proteínas. . 53
5.9. Relación tiempo y espacio por consulta *select* utilizando texto XML. 54
5.10. Resultados para bitmap Inglés, Select 1 Nivel utilizando codificación Gamma. 55
5.11. Resultados para bitmap Proteínas, Select 1 Nivel utilizando codificación Gamma. 56
5.12. Resultados para bitmap XML, Select 1 Nivel utilizando codificación Gamma. 56
5.13. Resultados para bitmap Inglés, Select 1 Nivel utilizando codificación Simple9. 57
5.14. Resultados para bitmap Proteínas, Select 1 Nivel utilizando codificación Simple9. 58
5.15. Resultados para bitmap XML, Select 1 Nivel utilizando codificación Simple9. 58
5.16. Resultados comparación consultas *select* para bitmap Inglés. 60
5.17. Resultados comparación consultas *select* para bitmap Proteínas. 61
5.18. Resultados comparación consultas *select* para bitmap XML. 61
5.19. Resultados para bitmaps comprimidos con densidad 5 %. La coordenada x parte desde la entropía de orden cero del bitmap, con lo que la redundancia se puede apreciar con mayor claridad. 63
5.20. Resultados para bitmaps comprimidos con densidad 10 %. La coordenada x parte desde la entropía de orden cero del bitmap, con lo que la redundancia se puede apreciar con mayor claridad. 64
5.21. Resultados para bitmaps comprimidos con densidad 20 %. La coordenada x parte desde la entropía de orden cero del bitmap, con lo que la redundancia se puede apreciar con mayor claridad. 65

5.22. Resultados para consultas *select* para un bitmap de densidad 2 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb. 67
5.23. Resultados para consultas *select* para un bitmap de densidad 3 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb. 67
5.24. Resultados para consultas *select* para un bitmap de densidad 5 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb. 68
5.25. Resultados para consultas *select* para un bitmap de densidad 10 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb. 68
5.26. Resultados para consultas *select* para un bitmap de densidad 20 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb. 69
5.27. Resultados para consultas *select*, para el primer nivel del wavelet tree de la transformada de Burrows-Wheeler del texto Inglés. Se utilizan diferentes valores de bloques, superbloques y S_s, y $d = 0.05$. 70
5.28. Resultados para consultas *select*, para el primer nivel del wavelet tree de la transformada de Burrows-Wheeler del texto Inglés. Se utilizan diferentes valores de bloques, superbloques y S_s, y $d = 0.1$. 70
5.29. Resultados comparación variantes estructura de Raman *et. al* para consultas *select* en el texto Inglés. 72
5.30. Resultados comparación variantes estructura de Raman *et. al* para consultas *select* en el texto Proteínas. 72
5.31. Resultados comparación variantes estructura de Raman *et. al* para consultas *select* en el texto XML. 73
5.32. Consultas *Count* en el FM-index sobre texto Inglés de 50 MB. 73

Capítulo 1

Introducción

Las estructuras de datos *compactas* son representaciones de estructuras de datos clásicas, pero que operan en espacio reducido. Se dividen en dos clases: *planas* y *comprimidas*. Las primeras no alteran los datos originales, sino que agregan poco espacio para soportar las operaciones. En cambio, las segundas recodifican los datos para soportar el acceso a los datos y la funcionalidad de la estructura, usando poco espacio total, que a veces es menor que el espacio de los datos planos. En cambio, en un esquema de compresión puro (en donde se usa un compresor tradicional sobre los datos), a pesar de utilizar menos espacio, se requiere la descompresión total de los datos para operar en ellos.

El desarrollo de estructuras de datos compactas ha sido tema de investigación relevante en los últimos años, dado que la información que se desea almacenar y consultar ha aumentado considerablemente. La importancia de dichas estructuras de datos radica en que existe una gran brecha en el desempeño de los distintos niveles de la jerarquía de memoria. Las estructuras de datos compactas permiten operar en memoria principal en casos que de otra manera requerirían memoria secundaria, logrando responder de forma eficiente las consultas sobre los datos en la estructura.

Un caso fundamental es la representación de una secuencia binaria B que soporte las operaciones *rank* y *select*, donde $rank_b(B, i)$ cuenta el número de ocurrencias del bit b hasta la posición i en B, y $select_b(B, i)$ encuentra la posición del i-ésimo bit b en B. Estas operaciones son esenciales en muchas otras estructuras de datos compactas, tales como árboles [2, 18, 22, 30], grafos [18, 22], permutaciones [4, 20], strings [14, 17], relaciones binarias [3, 5], y colecciones de texto [23], entre muchas otras. Dado lo fundamental de las operaciones *rank* y *select*, se han desarrollado diferentes estructuras de datos, las que ofrecen diferentes compromisos de espacio versus tiempo.

El problema principal de las estructuras de datos existentes es que las implementaciones que funcionan de forma más eficiente para *rank* no lo hacen con *select* y viceversa, por lo que lograr un balance en tiempo y espacio entre las estructuras para las dos operaciones es importante. Además, la mayoría de las soluciones a la consulta *select*, a pesar de ser atractivas en la teoría, en la práctica se ven afectadas por un significativo costo extra en espacio. Esto

desincentiva su uso en muchas aplicaciones.

Una secuencia binaria resulta compresible, en particular, cuando tiene pocos o muchos 1s, o éstos no están uniformemente distribuidos. Estos tipos de secuencias binarias aparecen frecuentemente, por lo que es de interés comprimirlos. Sin embargo, en las implementaciones actuales se presenta el problema de que, para dar respuesta a las operaciones básicas, se agrega una considerable redundancia en la forma de comprimir, es decir el espacio utilizado por la representación se aleja mucho del mínimo teórico requerido, lo que resulta en un mayor uso de espacio.

Las diferentes soluciones para responder a las consultas de *rank* y *select* pueden ser clasificadas en planas y comprimidas. Entre las estructuras de datos planas está la solución de Jacobson para *rank* [18], la solución de Clark para *select* [8], la solución de Munro [21], la implementación de González *et. al.* [16], la implementación *darray* de Okanohara y Sadakane [25], y las implementaciones de Vigna [31]. Entre las estructuras de datos comprimidas está la solución de Raman *et al.* [29], la implementación de Claude y Navarro [9], la implementación *sarray* de Okanohara y Sadakane [25], y otras más teóricas como las de Golynski *et al.* [13] y la de Pătraşcu [27].

En este trabajo se desarrollaron e implementaron nuevas técnicas para estructuras de datos compactas, las que fueron comparadas en tiempo y espacio con las soluciones implementadas existentes. Las técnicas propuestas permitieron diseñar dos estructuras de datos que son la contribución principal de este trabajo.

En primer lugar, se presenta una nueva estructura de datos plana que realiza un muestreo combinado para consultas *rank* y *select*, en vez de realizar muestreos independientes como en otras soluciones. Cada operación utiliza su propio muestreo, y además utiliza el muestreo de la otra operación si es posible. Esta estructura de datos permite resolver consultas *rank* y *select* en alrededor de 0.2 microsegundos, usando sólo un 3 % de espacio extra sobre el bitmap original. Este es un resultado sin precedentes, que finalmente posiciona a *rank* y *select* como operaciones que pueden usarse en la práctica sin restricciones.

La segunda contribución es una implementación de bitmaps comprimidos que mantiene el desempeño de las soluciones actuales, pero reduce drásticamente el uso de espacio extra en un 50 %-60 %. Esta estructura de datos utiliza un espacio extra de alrededor de un 10 % (del bitmap original), comparado con el 27 % usado por la mejor implementación actual [9], y responde consultas *rank* en alrededor de 0.4 microsegundos y consultas *select* en alrededor de 1 microsegundo. Esta mejora se logra reemplazando la *tabla universal* de respuestas precomputadas usada en la estructura de datos de Raman *et al.* [29] por la generación en tiempo de consulta de dichas respuestas. Esta solución se refuerza para el caso de bitmaps muy mal distribuidos, como los que aparecen en los FM-index [11] (un índice de texto que soporta búsqueda de patrones), logrando la misma reducción de espacio y tiempos mucho mejores. Finalmente esta estructura de datos se combina con las mejoras para *select* de la estructura de datos plana.

1.1. Organización de la Tesis

El presente documento está dividido en los siguientes capítulos:

- En el *Capítulo 2* se presentan y explican los conceptos básicos relevantes utilizados y necesarios para el desarrollo de esta tesis.
- En el *Capítulo 3* se explica el funcionamiento de las diferentes soluciones existentes para responder a las consultas *rank* y *select*. Las soluciones se dividen según su clasificación: planas y comprimidas.
- En el *Capítulo 4* se presentan las nuevas soluciones, explicando la creación de las estructuras de datos y cómo se logra responder a *rank* y *select* utilizando las estructuras de datos creadas. Estas nuevas soluciones corresponden principalmente a una estructura de datos para comprimir bitmaps basado en la idea propuesta por Raman *et al.* [29], una estructura de datos para bitmaps planos que utiliza un muestreo combinado para *rank* y *select*, y otras variantes y optimizaciones.
- En el *Capítulo 5* se presentan los resultados experimentales obtenidos al utilizar las nuevas estructuras de datos. Estos resultados corresponden al espacio utilizado y los tiempos obtenidos al responder a las consultas *rank* y *select*. Junto con esto, se realiza una comparación con las estructuras de datos existentes. Finalmente se presentan resultados al integrar una de las nuevas estructuras de datos al FM-index [11].
- Finalmente, en el *Capítulo 6* se presentan las conclusiones obtenidas en este trabajo y las perspectivas de trabajos futuros.

Los resultados preliminares relevantes de este trabajo fueron publicados en *11th International Symposium on Experimental Algorithms (SEA) 2012* [24].

Los códigos se incorporarán a la Librería de Estructuras de Datos Compactas `http://libcds.recoded.cl/`.

Capítulo 2

Conceptos Básicos

En este capítulo se presentan los conceptos básicos que se necesitan para comprender el trabajo desarrollado. Dentro de esto, se abarca la definición de entropía, clasificación de las estructuras de datos, definición de las operaciones de *rank* y *select*, tipos de codificación y, para finalizar, los conceptos asociados al FM-index.

2.1. Concepto de Entropía

Dada una secuencia binaria B de largo n compuesta por n_1 unos y n_0 ceros, la entropía empírica binaria de orden cero se define de la siguiente forma[1]:

$$H_0(B) = \frac{n_0}{n} \log \frac{n}{n_0} + \frac{n_1}{n} \log \frac{n}{n_1}.$$

Observe que por definición $n_0 \neq 0$ y $n_1 \neq 0$ pues eso indefine los argumentos del logaritmo correspondiente. En estos casos se considera que $H_0 = 0$. Las variaciones de la entropía de acuerdo a la densidad de 1s de una secuencia binaria se muestra en la Figura 2.1. Se observa que la entropía máxima ocurre cuando $n_0 = n_1$, y que a medida que un valor de bit se vuelve predominante la entropía disminuye.

En una secuencia de símbolos $S[1, n]$ sobre un alfabeto $\Sigma = \{1, ..., \sigma\}$, la entropía empírica de orden cero está definida como

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c},$$

donde n_c es el número de ocurrencias de c en S. La entropía empírica otorga una cota inferior, $nH_0(S)$, al número de bits necesarios para comprimir S, cuando siempre se codifica un símbolo de la misma manera. Algunas cotas comúnmente utilizadas, relacionadas con la

[1] En este documento se utiliza $\log = \log_2$, a menos que se mencione otra base.

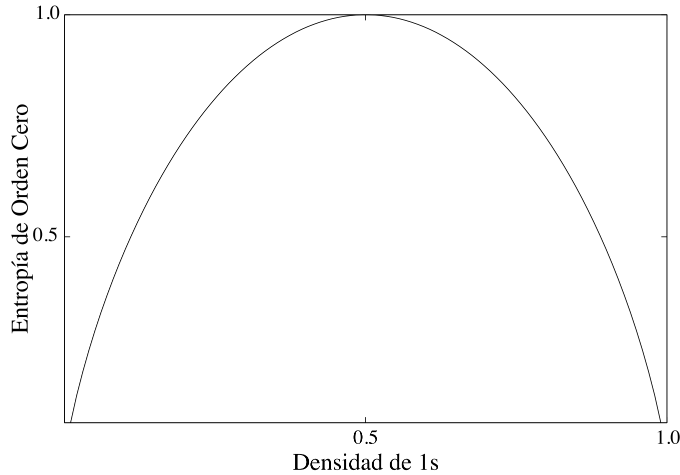


Figura 2.1: Entropía empírica de orden cero de acuerdo a la densidad de la secuencia binaria.

entropía de orden cero en una secuencia binaria $B[1,n]$ con $k = n_1$ 1s son:

$$\begin{aligned}\log \binom{n}{k} &\leq nH_0(B) \leq \log \binom{n}{k} + O(\log k),\\ k \log n/k &\leq nH_0(B) \leq k \log n/k + k \log e,\\ k \log n/k &\leq nH_0(B) \leq k \log n.\end{aligned}$$

Es posible lograr una mejor compresión mediante calcular las frecuencias según el contexto de cada símbolo. Utilizando este concepto se define la entropía empírica de orden k como

$$H_k(S) = \frac{1}{n} \sum_{A \in \Sigma^k} |S_A| H_0(S_A),$$

donde S_A es la subcadena formada por todos los caracteres que ocurren siguiendo las ocurrencias de A en S. Esta medida es una cota inferior para el número de bits que se necesitan para comprimir la secuencia S, con un compresor que codifique cada carácter de S usando un código que dependa del carácter en sí y de los k caracteres que lo preceden [12].

2.2. Estructuras de Datos Planas y Comprimidas

Las estructuras de datos compactas se clasifican en dos tipos: *planas* y *comprimidas*. Las primeras agregan información adicional al bitmap original, mientras que las segundas recodifican el bitmap original para ocupar poco espacio. Ambos tipos de estructura de datos mantienen la funcionalidad y proveen acceso directo a los datos. Las estructuras de datos compactas pueden permitir operar en memoria principal con datos que de otra manera sólo cabrían en memoria secundaria, mejorando así el rendimiento debido a la jerarquía de memoria.

2.3. Operaciones *rank* y *Select*

Dada una secuencia binaria $B[1,n]$, se definen las siguientes operaciones [18]:

- $rank_b(B, i)$ equivale al número de ocurrencias del bit b en la secuencia de bits $B[1, i]$.
- $select_b(B, i)$ equivale a la posición de ocurrencia del i-ésimo bit b en B.

Por ejemplo, si $B = 0110110101010110101110$, $select_1(B, 5) = 8$ y $rank_1(B, 5) = 3$. En este documento se utiliza $rank = rank_1$ y $select = select_1$.

Una secuencia binaria de n bits se caracteriza por su densidad, es decir, existen bitmaps *densos* y *dispersos*. Un vector es denso cuando la cantidad de 1s es cercana a $n/2$, y es disperso en otro caso. La entropía de orden cero, H_0, asociada a un bitmap está directamente relacionada con la densidad de éste[2]. El valor de H_0 disminuye si la densidad de 0s ó 1s es muy alta, y alcanza su valor máximo si la densidad es 50 %. Por lo tanto, al momento de representar el bitmap es importante considerar su densidad.

2.4. Codificación de Números

Un valor numérico n puede ser representado utilizando un entero (es decir una palabra de 32 o 64 bits). Sin embargo en ocasiones se necesita utilizar menos espacio, lo que es factible mediante el uso de otra codificación. Un número puede codificarse usando códigos de largo fijo o códigos de largo variable. Los primeros se caracterizan por usar la misma cantidad de bits para la codificación de cada número, mientras que los segundos destinan una cantidad distinta de bits según el valor a codificar. Esto permite utilizar menos espacio en la representación de los valores numéricos.

En esta sección se explican dos codificaciones: Gamma y Simple9 [1, 10]. La primera corresponde a un esquema de largo variable mientras que la segunda combina ambos esquemas para la representación de los valores. Otras codificaciones para enteros son: códigos *Elias-Delta*, códigos *Elias-Omega*, y *codificación de Rice*. En el capítulo 6 de [7] se presenta un resumen de estas y otras codificaciones.

2.4.1. Codificación Gamma

La codificación *Gamma* [10] es un código de largo variable para secuencias de enteros positivos. Es generalmente utilizado cuando no se conoce una cota superior para los valores en la secuencia, y cuando los valores pequeños son mucho más frecuentes que los grandes. La codificación de un entero n utiliza $2l - 1$ bits, donde $l = \lfloor \log n \rfloor$.

Para codificar un número n se realizan los siguientes pasos:

1. Escribir n en binario, utilizando l bits.
2. Anteponer $(l - 1)$ 0s a la representación binaria de n.

Por ejemplo, para codificar el número 5, se tiene que $(5)_2 = 101$, por lo tanto $l = 3$ y $Gamma(5) = 00101$.

[2] A menos que los 1s predominen, en cuyo caso H_0 se asocia a la cantidad de 0s.

Para decodificar un número codificado con este esquema:

1. Contar la cantidad de 0s hasta encontrar el primer 1; sea p la cantidad de posiciones contadas, incluyendo la posición de este primer 1.
2. El valor codificado es $n = 2^{p-1} + r$, donde r es el número representado por los $p - 1$ bits siguientes al primer 1 encontrado en el punto anterior.

Para decodificar el ejemplo anterior, 00101, se tiene que $p = 3$ y $r = (01)_2$. Por lo tanto $n = 2^{3-1} + (01)_2 = 4 + 1 = 5$.

Con este esquema no es posible codificar el valor 0. Una solución usual es sumar 1 a los números antes de codificarlos y restarles el mismo valor luego de decodificarlos.

2.4.2. Codificación Simple9

La codificación *Simple9* [1] es un esquema de compresión que posee un componente de largo fijo y otro de largo variable, y que ofrece mejor velocidad de descompresión que los esquemas de largo variable como la codificación Gamma.

Este esquema permite codificar una secuencia de enteros. La idea general para la codificación es almacenar la mayor cantidad posible de enteros codificados en una palabra de 32 bits, utilizando la cantidad mínima de bits necesarios para representar cada entero.

Para ello, de los 32 bits se usan 4 bits de control y 28 bits para datos. Para lograr un tiempo de acceso constante a los números codificados, se utiliza una máscara binaria sobre los 28 bits de datos. Para determinar qué máscara utilizar se consultan los bits de control, que indican cuál de los siguientes esquemas se está utilizando:

1. Bits de control: 0001. Hay 28 enteros representados con 1 bit.
2. Bits de control: 0010. Hay 14 enteros representados con 2 bits.
3. Bits de control: 0011. Hay 9 enteros representados con 3 bits (1 bit no utilizado).
4. Bits de control: 0100. Hay 7 enteros representados con 4 bits.
5. Bits de control: 0101. Hay 5 enteros representados con 5 bits (3 bits no utilizados).
6. Bits de control: 0110. Hay 4 enteros representados con 7 bits.
7. Bits de control: 0111. Hay 3 enteros representados con 9 bits (1 bit no utilizado).
8. Bits de control: 1000. Hay 2 enteros representados con 14 bits.
9. Bits de control: 1001. Hay 1 entero representado con 28 bits.

Por ejemplo, dada la secuencia de enteros: 1, 1, 3, 4, 1, 7, 5, 6, 7, una representación no codificada utiliza $9 \cdot size(int)$ bits, lo que en arquitecturas de 32 bits significa que se utilizan 288 bits. Sin embargo, cada entero puede ser representado utilizando a lo más 3 bits. Por lo tanto, utilizando el esquema 3, se puede codificar esta secuencia utilizando 32 bits en vez de 288. La palabra codificada es: 0011 001 001 011 100 001 111 101 110 111. De izquierda a derecha, los primeros cuatro bits son los bits de control: 0011, que indican el esquema 3. Los siguientes 28 bits corresponden a los datos. Cada 3 bits se tiene una representación de

un entero: 001 001 011 100 001 111 101 110 111, lo que corresponde a la secuencia original 1, 1, 3, 4, 1, 7, 5, 6, 7.

Por un lado, este esquema se considera como codificación de largo fijo porque siempre se usan palabras de 32 bits. Y, por otro lado, se considera como codificación de largo variable porque la cantidad de bits utilizada para cada entero de la secuencia depende de cuál de los esquemas anteriores se usó para la codificación (si la secuencia es codificada en varias palabras, cada una puede usar un esquema distinto). Esto influye en la cantidad (variable) de palabras que se usan para codificar una secuencia de números. Es importante destacar que la codificación depende de la permutación de los elementos en la secuencia de números a codificar.

2.5. FM-index

Con el objetivo de responder de forma eficiente a la búsqueda de un patrón sobre un texto, es necesario mantener un índice completo del texto. Un índice comprimido conocido es el *FM-index* [11], que está basado en la transformada de *Burrows-Wheeler (BWT)* [6] y tiene la característica de ser un índice que no almacena el texto indexado de forma explícita, sin embargo el texto original puede ser derivado del índice.

En las siguientes secciones se explicará la transformada de Burrows-Wheeler, la estructura de datos wavelet tree, y finalmente cómo se construye un FM-index, sus operaciones y su relación con las consultas *rank* sobre secuencias binarias.

2.5.1. Transformada de Burrows-Wheeler

La Transformada de Burrows-Wheeler (BWT) [6, 23] sobre un texto T de tamaño n es una permutación reversible de T que puede ser utilizada en algoritmos de compresión. La transformada permite convertir el texto en una secuencia de caracteres, que generalmente es más compresible con métodos locales.

La transformada de Burrows-Wheeler se aplica sobre un bloque de texto $T = c_0, ..., c_{n-1}$ donde los c_i pertenecen a un alfabeto Σ. Además, para indicar el fin del texto se usa un carácter especial, por ejemplo \$, que debe ser lexicográficamente menor a todos los caracteres en Σ. Para transformar T se realizan n rotaciones consecutivas del texto, donde cada rotación define una secuencia S_i (comenzando con $S_1 = T\$$), definidas como:

$$
\begin{aligned}
S_1 &= c_0c_1c_2c_3c_4 \ldots c_{n-1}\$ \\
S_2 &= c_1c_2c_3c_4 \ldots c_{n-1}\$c_0 \\
S_3 &= c_2c_3c_4 \ldots c_{n-1}\$c_0c_1 \\
S_4 &= c_3c_4 \ldots c_{n-1}\$c_0c_1c_2 \\
&\vdots \\
S_n &= \$c_0c_1c_2 \ldots c_{n-2}c_{n-1}
\end{aligned}
$$

Luego se debe ordenar de forma lexicográfica las secuencias de caracteres $S_1, ..., S_n$ generadas. Con esto se obtiene una matriz M donde la primera columna, F, corresponde a los caracteres del texto ordenados de forma lexicográfica y la última columna, L, corresponde a la transformada de Burrows-Wheeler sobre T, es decir $L = T^{bwt}$.

El ejemplo de la Figura 2.2, para el texto T = *mississippi*, muestra todos los pasos que se siguieron para obtener la transformada de Burrows-Wheeler. La matriz (a) muestra todas las rotaciones consecutivas que se realizan sobre el texto, y la matriz (b) corresponde a la matriz (a) ordenada de forma lexicográfica por filas. La última columna de la matriz (b) corresponde a T^{bwt}.

(a)		F		$L = T^{bwt}$	
m i s s i s s i p p i $		$	m i s s i s s i p p	i	
i s s i s s i p p i $ m		i	$ m i s s i s s i p	p	
s s i s s i p p i $ m i		i	p p i $ m i s s i s	s	
s i s s i p p i $ m i s		i	s s i p p i $ m i s	s	
i s s i p p i $ m i s s		i	s s i s s i p p i $	m	
s s i p p i $ m i s s i		m	i s s i s s i p p i	$	
s i p p i $ m i s s i s	⇒	p	i $ m i s s i s s i	p	$= M$
i p p i $ m i s s i s s		p	p i $ m i s s i s s	i	
p p i $ m i s s i s s i		s	i p p i $ m i s s i	s	
p i $ m i s s i s s i p		s	i s s i p p i $ m i	s	
i $ m i s s i s s i p p		s	s i p p i $ m i s s	i	
$ m i s s i s s i p p i		s	s i s s i p p i $ m	i	

(a) (b)

Figura 2.2: Ejemplo de la transformada de Burrows-Wheeler para el texto original $T\$ = S_1$ = mississippi$ y T^{bwt}= ipssm$pissii.

Sobre la matriz M se tienen las siguientes definiciones:

- $C[c]$ corresponde al número de ocurrencias alfabéticamente menores a $c \in \Sigma$ en el texto T.

- $Occ(c, q)$ es el número de ocurrencias del carácter $c \in \Sigma$ en el prefijo $T^{bwt}[1, q]$.
- $LF(i) = C[T^{bwt}[i]] + Occ(T^{bwt}[i], i)$.

LF() es una función de mapeo que permite ubicar el carácter $T^{bwt}[i]$ de la última columna en la primera (*Last-to-First*), es decir, un carácter en la posición i de la última columna L se encuentra en la primera columna F en la posición $LF(i)$. Por ejemplo, $LF(10) = C[s] + Occ(s, 10) = 8 + 4 = 12$, por lo que $F[LF(10)] = F[12]$ que es $T^{bwt}[10]$, que en ambos casos corresponde a la primera s en el texto *mississippi*.

2.5.2. Wavelet Tree

Un *wavelet tree* es una estructura de datos para representar una secuencia $S = s_1 \ldots s_n$, con $s_i \in \Sigma$, como un árbol binario balanceado en base a sub-alfabetos de Σ.

Dado un sub-alfabeto $[a..b] \subseteq \Sigma$, un wavelet tree es un árbol binario balanceado con $b - a + 1$ hojas. Si $a = b$, entonces el árbol es una hoja etiquetada como a. En otro caso, el árbol posee un nodo raíz R que representa $S[1, n]$. Este nodo raíz almacena un bitmap $B_R[1, n]$ tal que $B_R[i] = 0$ si $S[i] \leq (a + b)/2$, y $B_R[i] = 1$ en otro caso. Esto corresponde a codificar la mitad del alfabeto como 0 y la otra mitad como 1. Sea $S_0[1, n_0]$ la subsecuencia de $S[1, n]$ formada por los símbolos $c \leq (a + b)/2$, y $S_1[1, n_1]$ la subsecuencia formada por los símbolos $c > (a + b)/2$. Luego, el hijo izquierdo de R es el wavelet tree para S_0 y el hijo derecho de R es el wavelet tree para S_1.

Es posible reconstruir la secuencia S dada su representación como wavelet tree. Para acceder al elemento $S[i]$ se explora recursivamente el árbol partiendo desde la raíz $V = R$: si en un nodo V el bitmap B_V en la posición i es 0, entonces se continúa en el sub-árbol izquierdo, y en otro caso con el sub-árbol derecho. Al realizar el paso recursivo es necesario conocer qué posición del sub-árbol izquierdo o derecho corresponde a la posición i en la raíz respectiva. En caso de continuar por la izquierda, la posición i corresponde a $rank_0(B_V, i)$. Si se continúa por la derecha, la posición i corresponde a $rank_1(B_V, i)$. Se continúa recursivamente hasta alcanzar una hoja, cuya etiqueta corresponde al símbolo $S[i]$.

Además de representar en forma sucinta la secuencia S, el wavelet tree permite realizar operaciones $rank_c$ y $select_c$ en S sobre símbolos arbitrarios $c \in \Sigma$. Para responder $rank_c(i)$ se considera el bit, 0 ó 1, asociado al caracter c en el bitmap B_R de la raíz del wavelet tree. Similarmente al caso anterior, los valores 0 y 1 indican si se prosigue en el sub-árbol izquierdo o derecho, respectivamente. En cada paso recursivo se continúa con la posición $i \leftarrow rank_0(B_R, i)$ o $i \leftarrow rank_1(B_R, i)$ según corresponda. La operación continúa hasta alcanzar una hoja, y la respuesta final es la posición i calculada al alcanzar la hoja.

Para responder $select_c(i)$ se procede desde las hojas hacia la raíz. Se comienza desde la hoja donde c está representado, y se utiliza $i \leftarrow select_0(B_V, i)$ o $i \leftarrow select_1(B_V, i)$, según el nodo sea hijo izquierdo o derecho de su padre V, para obtener la posición de la i-ésima ocurrencia de c. Se continúa recursivamente hasta la raíz del wavelet tree, donde i resulta ser la posición buscada de la ocurrencia de c.

2.5.3. Construcción del FM-index

El FM-index [11] es un *auto-índice* que permite buscar de forma eficiente las ocurrencias de un determinado patrón $P = p_0 \dots p_{m-1}$, de largo m, como un substring del texto T. El índice está basado en una representación compacta de la transformada de Burrows-Wheeler (BWT) [6] sobre T, por lo que el texto T es preprocesado y el patrón P es entregado de forma on-line. El concepto de *auto-índice* indica el hecho de que el texto T no es almacenado directamente pero puede ser derivado del FM-index [12].

La matriz M, descrita en la Sección 2.5.1 anterior, que se obtiene al aplicar la BWT, puede ser vista como el arreglo de sufijos [23] A, donde $A[i] = j$ si la fila *i-ésima* de la matriz contiene el string $c_j c_{j+1} \dots c_{n-1} \$ c_0 c_1 \dots c_{j-1}$. El arreglo de sufijos permite búsqueda eficiente de un patrón P sobre el texto T. Como cada patrón P es un prefijo de algún sufijo en A y los sufijos en A están ordenados lexicográficamente, las ocurrencias de P forman un intervalo $[sp, ep]$ en A donde se ubican todos los sufijos que comienzan con el patrón. Tradicionalmente, la búsqueda del segmento se realiza utilizando dos búsqueda binarias, para la posición inicial y final del segmento.

Sin embargo, mediante la transformada del texto, T^{bwt}, es posible representar de forma implícita el arreglo A. Es así como la idea del FM-index es almacenar T^{bwt} de forma comprimida y simular la búsqueda en el arreglo de sufijos.

Para la búsqueda, se necesita implementar la operación $Occ(c, i)$ para el caso general. Esto puede realizarse utilizando operaciones *rank* sobre los caracteres del texto, donde $Occ(c, i) = rank_c(T^{bwt}, i)$ y se usa $C[c] + Occ(c, i)$. Para ello, se puede representar T^{bwt} utilizando un wavelet tree, que soporta operaciones *rank* y *select* sobre un alfabeto arbitrario. Entonces, para encontrar el intervalo $[sp, ep]$ en A con las ocurrencias de un patrón $P = p_0 \dots p_{m-1}$ se procede de la siguiente forma[3]

- Sea $sp \leftarrow 1$, $ep \leftarrow n$, $i \leftarrow m - 1$.
- Mientras $sp \leq ep$ y $i > 0$, calcular $sp = C[P[i]] + Occ(P[i], sp - 1) + 1$ y $ep = C[P[i]] + Occ(P[i], ep)$, y asignar $i \leftarrow i - 1$.

El desempeño del FM-index está directamente relacionado con la operación *rank* sobre un alfabeto arbitrario que, a su vez, debido a la implementación de los wavelet tree, depende del desempeño de *rank* para secuencias binarias.

[3]Si el patrón no existe en el texto, eventualmente ocurrirá que $ep > sp$, y se termina la búsqueda.

Capítulo 3

Trabajo Relacionado

En el presente capítulo se explican las soluciones prácticas existentes para responder a consultas *rank* y *select*. Estas soluciones están agrupadas de acuerdo a si las estructuras de datos son planas o comprimidas.

3.1. Estructuras de Datos Planas

Las *estructuras de datos planas* son aquellas que mantienen el bitmap original y que agregan estructuras auxiliares para responder *rank* y *select*. Dentro de este tipo de soluciones existen varias implementaciones.

3.1.1. Solución de Jacobson para *rank*

Jacobson [18] propuso la primera solución para *rank* en tiempo constante, utilizando $n+o(n)$ bits. La estructura de datos está formada por dos niveles de directorios, conocidos como bloques y superbloques, donde un directorio se refiere a una estructura de datos para encontrar información. Además se tiene una tabla precalculada de respuestas *rank* para todas las secuencias de bits pequeñas.

Para crear la estructura de datos se divide el bitmap B de largo n en bloques de largo fijo $b = \lfloor (\log n)/2 \rfloor$. Estos bloques son agrupados en superbloques de largo $s = b\lfloor \log n \rfloor$. Luego se almacenan en un arreglo R_s las respuestas de *rank* al comienzo de cada superbloque, es decir $R_s[i] = rank(B, i \cdot s)$, con $i \in [0, n/s]$. Con esto se obtienen $n/s = O(n/\log^2 n)$ superbloques, donde cada uno utiliza $\log n$ bits, por lo que se necesitan $O(n/\log n)$ bits para el primer directorio. Luego, para cada bloque k perteneciente a un superbloque $i = (k \text{ div } \lfloor \log n \rfloor)$, con $k \in [0, \lfloor n/b \rfloor]$, se almacena similarmente el valor de *rank* contando desde el comienzo del superbloque al que pertenece. Estos valores son almacenados en un arreglo R_b, donde $R_b[k] = rank(B, k \cdot b) - rank(B, i \cdot s)$. Para almacenar estos valores el arreglo R_b necesita $(n/b)\log s = O(n \log\log n/\log n)$ bits, dado que tiene n/b entradas y cada una requiere $\log s$ bits. Finalmente, para todas las subsecuencias S de largo b y para cada posición i dentro de S, se precalcula la tabla $R_p[S, i] = rank(S, i)$. Es-

to requiere $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log\log n)$ bits. Las estructura completa necesita $O(n/\log n + n \log\log n/\log n + \sqrt{n} \log n \log\log n) = o(n)$ bits, además de almacenar el mismo bitmap B.

Con esta solución se logra responder *rank* en tiempo constante de la siguiente forma:

$$rank(B,i) = R_s[i \text{ div } s] + R_b[i \text{ div } b] + R_p[B[(i \text{ div } b) \cdot b + 1 \ldots (i \text{ div } b) \cdot b + b], i \bmod b]$$

La Figura 3.1 ilustra un ejemplo.

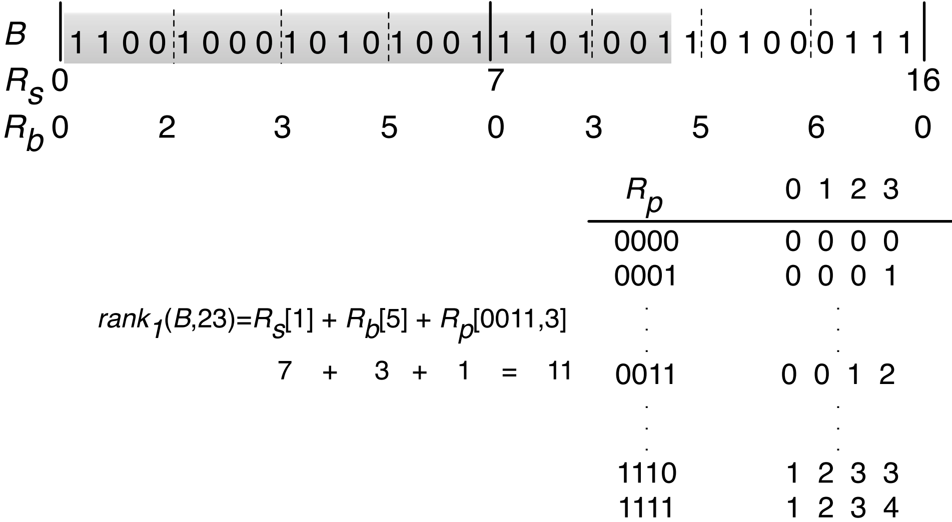


Figura 3.1: Ejemplo para responder una consulta *rank* para bloques de tamaño $b = 4$ y superbloques de tamaño $s = 16$.

3.1.2. Implementación de González *et al.*

González *et al.* [16], basados en la solución de Jacobson, realizaron las siguientes implementaciones:

Implementación de Dos Niveles para *rank*

La diferencia principal de esta implementación con la solución de Jacobson es que se reemplaza la tabla R_p por el uso de operaciones *popcount*, que consiste en contar cuántos bits están en 1 en un pequeño arreglo de bits. Se calcula $R_p[S,i] = popcount(S \ \& \ 1^i)$, donde 1^i es una secuencia de i 1s, y & es el operador *and* para bits. Esto permite eliminar el segundo argumento de R_p, lo que hace la tabla más pequeña.

La operación *popcount* está implementada de la siguiente manera:

1: popc = {0, 1, 1, 2, 1, 2, 2, 3, 1, ...}
2: popcount = popc[x & 0xFF] + popc[(x ≫ 8) & 0xFF] + popc[(x ≫ 16) & 0xFF] + popc[x ≫ 24]

---

donde popc es una tabla de *popcount* precomputada, indexada por bytes. Esto hace mejor uso del caché y es más rápido que una tabla indexada de a 16 bits, aunque requiera más accesos. Además, esta opción resultó ser más rápida que otras soluciones basadas en manipulación de bits. De todos modos, existe un tradeoff tiempo/espacio ocasionado por el tamaño de los bloques y el uso de *popcount*. Mientras mayor es el tamaño del bloque, mayor es la cantidad de accesos a memoria realizados por *popcount*. Experimentalmente González *et al.* obtuvieron la mejor combinación con bloques de $b = 32$ bits y superbloques de $s = 256$ bits lo cual además permite alinear los valores de R_b a bytes. Esto requiere un espacio extra al bitmap de entrada de 37,5 %.

Implementación de Un Nivel para *rank*

Con el propósito de reducir el espacio que requiere el esquema de Jacobson al usar R_s y R_b, González *et al.* consideran la alternativa de usar sólo la tabla R_s. Este directorio considera una entrada cada $s = 32 \cdot k$ bits, y utiliza un entero de 32 bits para almacenar los valores de *rank* para cada entrada. Para responder una consulta $rank(B, i)$, primero se busca la entrada en R_s que anteceda y sea más cercana a i. Luego, se escanea secuencialmente el arreglo, realizando *popcount* en segmentos de $w = 32$ bits, hasta alcanzar la posición deseada, de la siguiente manera:

$$
\begin{aligned}
rank(B, i) = {} & R_s[i \text{ div } s] \\
& + \sum_{j=(((i \text{ div } s)\cdot s) \text{ div } w)+1}^{(i \text{ div } w)-1} popcount(B[j \cdot w + 1 \ldots j \cdot w + w]) \\
& + popcount(B[(i \text{ div } w) \cdot w + 1 \ldots (i \text{ div } w) \cdot w + w] \; \& \; 1^{i \bmod w}).
\end{aligned}
$$

El escaneo secuencial accede a lo más a k palabras de memoria, y el espacio extra es $1/k$, lo que implica un tradeoff espacio/tiempo. Por ejemplo, con $k = 3$, se tiene aproximadamente el mismo costo de espacio que en la implementación de dos niveles [16]. Experimentalmente, con $k = 20$ se logra un espacio extra de un 5 % al bitmap de entrada, respondiendo consultas más lentamente que otras implementaciones, pero sin dejar de ser competitivo.

Respondiendo *Select* con Búsqueda Binaria

La estructura de González *et al.* permite responder consultas *select* aprovechando la información almacenada para responder *rank*, ya sea la implementación de uno o dos niveles. Para responder a una consulta $select(B, i)$, se realiza búsqueda binaria sobre el primer nivel R_s

de la estructura de *rank* (superbloques), obteniendo el superbloque en donde se encuentra la respuesta. Luego se busca en el superbloque el bloque correspondiente al i-ésimo bit. Finalmente, se busca la posición relativa del i-ésimo bit en el bloque.

González *et al.* consideraron varias opciones para la búsqueda del bloque en el superbloque: (1) búsqueda binaria usando R_b, (2) búsqueda secuencial usando R_b, y (3) búsqueda secuencial usando *popcount* (sin usar R_b). Para la búsqueda en el bloque se consideran dos opciones: (a) contar por byte usando *popcount*, y usar búsqueda bit por bit en el último byte; y (b) usar búsqueda bit por bit en el bloque completo. Experimentalmente se estableció que la mejor combinación es (3,a) que usa búsqueda secuencial con *popcount* para encontrar el bloque y búsqueda con *popcount* para buscar dentro del bloque.

3.1.3. Solución de Clark para *Select*

Clark [8] propuso una estructura de datos que utiliza tres niveles de directorios con el objetivo de precalcular algunos valores, que serán posteriormente accedidos de forma directa al momento de dar respuesta a una consulta *select*.

La estructura de datos almacena, en un primer directorio, la posición de cada $\lceil \log n \rceil \lceil \log \log n \rceil$-ésimo bit en 1. Obtiene bloques de diferentes tamaños r, que representan el subrango entre dos valores del primer directorio. El valor de r no necesita ser guardado ya que puede ser calculado en tiempo de consulta. En un segundo nivel se verifica el tamaño r de cada bloque y se almacenan todas las respuestas *select* para los bloques *dispersos*. Un bloque, en este nivel, es considerado disperso si cumple que $r \geq \lceil \log n \rceil^2 \lceil \log \log n \rceil^2$. Para los otros bloques, llamados *densos*, sólo se almacenan las posiciones donde comienzan.

Los bloques densos se subdividen, almacenando la posición de cada $\lceil \log r \rceil \lceil \log \log n \rceil$-ésimo bit en 1 en un segundo directorio, creando sub-bloques de tamaño r'. Un sub-bloque es disperso si $r' \geq \lceil \log r' \rceil \lceil \log r \rceil \lceil \log \log n \rceil^2$, y en este caso se guardan las respuestas *select* para los 1s del sub-bloque en un tercer directorio. En caso contrario el sub-bloque es denso, y la respuesta tendrá que ser buscada en tiempo de consulta. Las respuestas *select* en este nivel sólo necesitan $\log(\lceil \log n \rceil^2 \lceil \log \log n \rceil^2) \in O(\log \log n)$ bits, y las búsquedas secuenciales sólo abarcan $O(\log n)$ bits, lo que permite usar tablas de respuestas precomputadas.

Cada directorio utiliza a lo más $\lfloor n / \lceil \log \log n \rceil \rfloor$ bits, y las tablas de respuestas precomputadas utilizan $O(\sqrt{n} \log n \log \log n)$ bits. Por lo tanto, se utiliza un espacio extra de $3n / \lceil \log \log n \rceil + O(\sqrt{n} \log n \log \log n) = o(n)$ bits. A cambio, se consigue dar respuesta en tiempo constante. Para responder a una consulta *select*(B, i) es necesario conocer a qué bloque pertenece i en el primer directorio, calculando el valor de r. Si $r > (\lceil \log n \rceil \lceil \log \log n \rceil)^2$ la respuesta está almacenada y se consigue en un acceso directo. En caso contrario se repiten los pasos sobre el segundo nivel de directorio, considerando que la respuesta puede estar almacenada o se tendrá que realizar un escaneo sobre una secuencia pequeña de bits (utilizando las tablas de respuestas precomputadas) cuando los valores de sus respuestas no han sido almacenadas en ninguno de los directorios.

Una variante de la solución de Clark fue desarrollada por Kim *et al.* [19], la cual se en-

foca en mejorar el desempeño para los peores casos donde se necesitan muchos accesos a tabla, que empeoran el tiempo de respuesta. Esta implementación es entre 1.5 y 2.5 veces más rápida que la solución de Clark, pero ocupa más espacio. En nuestra investigación experimental utilizamos la implementación basada en la idea original de Clark, implementada por R. González (ver inicio de Sección 5), ya que uno de los objetivos de las estructuras desarrolladas en este trabajo es reducir el espacio de las estructuras de datos para *rank* y *select*, obteniendo tiempos competitivos.

3.1.4. Implementación *darray* de Okanohara y Sadakane

Okanohara y Sadakane [25] implementaron una estructura de datos llamada *darray* para resolver eficientemente consultas *select*, siguiendo la línea de Clark (Sección 3.1.3). En cambio, para responder *rank* en *darray* se usa una estructura de dos directorios similar a la de González *et al.* (Sección 3.1.2). El funcionamiento de *darray* se basa en crear dos niveles de directorios, lo que se realiza particionando el bitmap B en bloques tal que cada bloque contiene L 1s.

Esta estructura de datos trabaja con un arreglo $P_l[0...n/L-1]$ donde $P_l[i] = select(B, i \cdot L)$. Estos bloques se clasifican en dos grupos según su largo $(P_l[i] - P_l[i-1])$. Si el tamaño de un bloque es más largo que un parámetro L_2, entonces se almacenan todas las posiciones de 1s explícitamente en un primer directorio S_l. Si el largo del bloque es más pequeño que L_2 se almacenan las posiciones de los L_3-ésimos 1s ($i = 0...L_2/L_3$) en un segundo directorio S_s, donde L_3 también es un parámetro. Para esto utiliza $\log L_2$ bits, ya que almacena los valores relativos a los almacenados en el directorio P_l.

Para responder $select(i, B)$ se verifica si el bloque donde se encuentra i, $P_l[\lceil i/L \rceil]$, es más largo o no que L_2. Si el bloque es más largo que L_2, se obtiene el valor que está almacenado explícitamente en S_l. Si no se cumple esta condición, se utilizan los valores de S_s. Cuando la respuesta no se encuentra en estos directorios se realiza una búsqueda secuencial en el bloque en el cual está la respuesta. Esto requiere tiempo $O(L_2/\log n)$ considerando que se puede leer $O(\log n)$ bits en tiempo constante en el modelo RAM.

El tamaño de P_l es $O((n/L) \cdot \log n)$ bits, S_l require a lo más $((n/L_2) \cdot L \log n)$ bits y S_s requiere a lo más $((n/L_3) \cdot \log L_2)$ bits. Si se fijan los valores en $L = \log^2 n$, $L_2 = \log^4 n$ y $L_3 = \log n$, el tamaño de la estructura de datos es $n + O((n \log\log n)/\log n) = n + o(n)$ y el tiempo de respuesta para *select* es $O(\log^3 n)$.

3.1.5. Implementaciones de Vigna

Otras soluciones basadas en dos niveles de directorios son las propuestas por Vigna [31]. Estas soluciones están orientadas al uso de procesadores de 64 bits, aunque hay versiones para 32 bits. Este tipo de algoritmos se denomina *broadword computing*. Vigna presenta las siguientes soluciones: *Rank9* para resolver consultas *rank* de forma eficiente, *Select9* que se basa en la estructura de *Rank9* para responder *select* en tiempo casi constante, y sólo realiza accesos a datos alineados, y *Simple Select*, que es independiente de las estructuras de *Rank9* y *Select9*, que responde sólo consultas *select*.

Rank9

Esta estructura de datos utiliza el enfoque tradicional de dos directorios, pero a diferencia de implementaciones como la de González *et al.* [16], que utilizan una tabla de *popcount* para computar las respuestas una vez localizada la palabra correspondiente, utiliza un algoritmo broadword para contar dentro de dicha palabra de memoria. Supone que la secuencia de bits B está dividida en un arreglo de palabras de 64 bits cada una. El bit en posición p se encuentra en la palabra $\lfloor p/64 \rfloor$.

Para cada subsecuencia de ocho palabras que comienzan en el bit de cada posición p, lo que se denomina un *bloque básico*, se generan dos palabras asociadas: la primera, en el primer directorio, contiene $rank(p)$; la segunda contiene siete valores de 9 bits calculados por $rank(p+64k) - rank(p)$, para $1 \leq k \leq 7$. Cada valor es desplazado a la izquierda en $9(k-1)$ bits.

Para calcular una respuesta $rank(p)$ donde p reside en la palabra w, se realiza lo siguiente:

- Sumar las palabras del primer directorio de la secuencia que comienza en $\lfloor w/8 \rfloor$.
- Sumar las palabras del segundo directorio si $p \bmod w \neq 0$.
- Invocar el algoritmo broadword en la palabra que contiene a p, con una máscara adecuada, para contar los 1s en esa palabra.

El algoritmo [31] se muestra en la Figura 3.2, donde L_k denota la menor constante cuyos 1s se encuentran en las posiciones $0, k, 2k, \ldots$, en este caso $L_8 = $ 0x0101010101010101.
Estos valores son usados para realizar sumas parciales en subpalabras de tamaño k.

1: $x = x - ((x \,\&\, \text{0xAAAAAAAAAAAAAAAA}) \gg 1)$
2: $x = (x \,\&\, \text{0x3333333333333333}) + ((x \gg 2) \,\&\, \text{0x3333333333333333})$
3: $x = (x + (x \gg 4)) \,\&\, \text{0x0F0F0F0F0F0F0F0F0}$
4: **return** $x \cdot L_8 \gg 56$

Figura 3.2: Algoritmo broadword para contar la cantidad de 1s en x.

En el primer paso (línea 1), en cada par de bits se almacena la cantidad de 1s originalmente contenida en dicho par. Los siguientes pasos (líneas 2 a 3) realizan sumas parciales para, finalmente (línea 4), realizar la última suma mediante la multiplicación por L_8.

Select9

Esta solución se apoya en la implementación de *Rank9* para responder consultas *select* en tiempo casi constante. Una vez ubicada la palabra x necesaria para computar el resultado final, se usa el algoritmo de la Figura 3.3 para calcular el índice del r-ésimo 1 en ella.

Para responder una consulta $select(r)$ se comienza al igual que el caso anterior, es decir, pri-

1: $s = x - ((x \,\&\, \text{0xAAAAAAAAAAAAAAAA}) \gg 1)$
2: $s = (s \,\&\, \text{0x3333333333333333}) + ((x \gg 2) \,\&\, \text{0x3333333333333333})$
3: $s = ((s + (s \gg 4)) \,\&\, \text{0x0F0F0F0F0F0F0F0F0}) * L_8$
4: $b = ((s \leq_8 r * L_8) \gg 7) * L_8 \gg 53 \,\&\, \bar{7}$
5: $l = r - (((s \ll 8) \gg \text{b})\&\text{0xFF})$
6: $s = (((x \gg \text{b} \,\&\, \text{0xFF}) * L_8 \,\&\, \text{0x8040201008040201} >_8 0) \gg 7) * L_8$
7: **return** $b + (((s \leq_8 l * L_8) \gg 7) * L_8 \gg 56)$

Figura 3.3: Algoritmo broadword para calcular el índice del r-ésimo 1 en x. Si no existe tal bit, el resultado es 72.

mero se calculan en s las sumas acumulativas de los 1s de x (líneas 1 a 3), sin sobre-escribir x como en el algoritmo anterior. Luego, en paralelo se compara cada suma parcial[1] con r (línea 4); la cantidad de respuestas positivas es exactamente el índice m del byte que contiene el bit cuyo rank es r. Por lo tanto la posición hasta el inicio del byte m es $b = m \cdot 8$. Luego (línea 5) se define l como los bits restantes a ser buscados, restando los 1s del byte siguiente a r. Se genera una palabra z que contiene ocho copias del byte l, pero cada j-ésima copia mantiene sólo el bit j (línea 6). Sobre z se compara en paralelo cada copia del byte contra 0. Esto permite calcular, multiplicando por L_8, el rank para cada bit de m (todo esto ocurre en la línea 6). Finalmente, se comparan las sumas acumuladas con ocho copias (completas) del byte m. La cantidad k de resultados positivos coincide con el l-ésimo 1. La respuesta entonces es $b + k$ (línea 7).

El algoritmo de la Figura 3.3 sólo describe el caso particular de procesar una palabra de memoria. Queda establecer cómo encontrar el bloque que contiene dicha palabra, y el superbloque que contiene dicho bloque. Vigna señala que se puede utilizar la misma técnica de búsqueda binaria utilizada por González *et al.* [16], pero además propone una técnica denominada *hinted bsearch*. En esta técnica se usa un directorio de un nivel para localizar más rápidamente la región sobre la cual realizar la búsqueda binaria para encontrar el bloque. Luego realiza una búsqueda broadword de la palabra con el bit buscado sobre el bloque encontrado, para finalmente aplicar el algoritmo descrito en la Figura 3.3.

Simple Select

Además de *Select9*, Vigna propone una estructura especializada para consultas *select* que no depende de *Rank9*. Esta estructura se llama *Simple Select*. Se basa en la observación de que el algoritmo de la Figura 3.3 puede ser extendido a un algoritmo de búsqueda de bits. Si se quiere ubicar el bit cuyo *rank* es r en una secuencia de palabras, se carga la primera palabra en x y se ejecuta un ciclo en las primeras 3 líneas del algoritmo. Luego, si $r < (s \gg 6)$, se termina el ciclo y se prosigue normalmente. En otro caso, se carga x con el contenido de la siguiente palabra, se disminuye r en $s \gg 56$ y se itera nuevamente.

[1]Sean x e y dos secuencias binarias de 64 bits. $x \leq_k y$ crea una nueva palabra de 64 bits, dividida en sub-palabras de k bits. El bit más significativo de cada nueva sub-palabra es 1 si y solamente si la sub-palabra respectiva en x es menor o igual que la sub-palabra respectiva en y. Similarmente se define $>_8$. Además, $\bar{x}$ denota el complemento binario de x.

La implementación de *Simple Select* utiliza una estructura de dos directorios, similar al *darray* de Okanohara y Sadakane (Sección 3.1.4). En el primer directorio se almacenan los 1s en posiciones múltiplos de $\lceil Lm/n \rceil$, donde m indica cada cuántos 1s se realiza el muestreo y L es una constante que limita el tamaño del directorio. Luego para cada bit en el directorio se asigna una cantidad de palabras, acotadas por una constante M, dependiendo de la densidad del bitmap. En estas palabras se almacena un subdirectorio de 16 bits. Se utilizan ambos directorios para encontrar una posición cercana al bit que se desea seleccionar, y luego se realiza una búsqueda lineal broadword, descrita en el párrafo anterior. En los resultados experimentales de Vigna esta solución resultó ser la más rápida y con menor uso de memoria. Sin embargo está limitada a responder sólo consultas *select*.

3.1.6. Solución de Golynski *et al.* para Bitmaps Densos

Golynski [15] propone una estructura de datos plana para responder *rank* y *select* en tiempo constante, y que utiliza un espacio extra sobre el bitmap de entrada de $(1+o(1))(n \log \log n / \log n) + O(n/\log n)$ bits. La solución consiste en una estructura base, denominada *count index*, e índices adicionales para cada tipo de consulta. En la estructura count index se particiona el bitmap B de tamaño n en *chunks* de tamaño $\log n - 3 \log \log n$, y se almacena la cardinalidad (cantidad de 1s) de cada chunk.

Índice para *Select*

El índice para soportar consultas *select* primero almacena las ocurrencias de los $(k \cdot S_1)$-ésimos bits en 1 (con $1 \leq k \leq n/S_1$) en B, donde $S_1 = (\log n)^2$. Los segmentos de B entre las posiciones $select(k \cdot S_1)$ y $select((k+1) \cdot S_1)$ se denominan *upper blocks*. Para ejecutar $select(B, i)$, primero se calcula $j = \lfloor i/S_1 \rfloor$ que corresponde al *upper block* que contiene el i-ésimo 1. Luego, la consulta *select* se expresa como:

$$select(i) = select(B, j \cdot S_1) + select(UB_j, i \ mod \ S_1)$$

donde UB_j corresponde al j-ésimo upper block (es decir al segmento entre las posiciones $select(j \cdot S_1)$ y $select((j+1) \cdot S_1)$). Como la respuesta para la posición $j \cdot S_1$ en B está almacenada en el índice, el problema se reduce a responder *select* en UB_j.

Dependiendo de la densidad del upper block es cómo se responde una consulta *select* sobre él. En la construcción del índice, un upper block se considera disperso si su largo es al menos $(\log n)^4$. En ese caso se almacenan las respuestas explícitamente. En caso contrario, el bloque se considera denso y se particiona en *middle blocks* que contienen $S_2 = \log n \log \log n$ 1s, de manera similar a como se construyen los upper blocks, pero almacenando posiciones relativas al upper block y no al bitmap original.

Al igual que para los upper blocks, los middle blocks se clasifican como dispersos o densos. Un middle block es disperso si su largo es al menos $(\log n \log \log n)^2$, y en este caso también se almacenan las respuestas explícitamente. El middle block se considera denso si

su largo es a lo más $(\log n)^2/(4 \log\log n)$, y no se almacenan respuestas. Ahora, es posible que un bloque no sea clasificado como denso ni como disperso. Estos bloques se particionan cada $S_3 = (\log\log n)^3$ 1s, y son llamados *lower blocks*. Cuando el largo de un lower block es al menos $\log n(\log\log n)^4$, se considera disperso y se almacenan explícitamente sus respuestas.

Finalmente, para responder a *select* en un bloque denso (ya sea en un middle block o un lower block) se utiliza la estructura count index. Sea MB un middle block denso y $P = C_l \ldots C_m$ los chunks en el count index que corresponden a MB. Supongamos por simplicidad que MB está alineado a C_l y a C_m. Se tiene una tabla universal T que toma como entrada una secuencia de valores del count index, y tiene precalculada la posición (el chunk) donde la suma excede cada posible valor. Otra tabla universal, Q, completa el *select* dentro de cualquier posible chunk en tiempo constante.

Entonces, para responder $select(MB, i)$ se consulta T para encontrar el chunk C_h $(l \leq h \leq m)$ donde la cantidad de 1s acumulada excede i, y finalmente se consulta Q para obtener la posición del i-ésimo bit en 1, relativo a C_h. El procedimiento es análogo para los *lower blocks* densos. Esto permite calcular *select* dentro de MB en tiempo constante.

Índice para *rank*

El índice para *rank* particiona el bitmap en *upper blocks* de tamaño fijo $S_1 = (\log n)^2$. Para cada upper block se almacena la respuesta *rank* de la posición anterior al inicio de dicho bloque. Sea $j = \lfloor i/S_1 \rfloor$ el número del upper block que contiene el i-ésimo bit de B, entonces

$$rank(B, i) = rank(B, j \cdot S_1 - 1) + rank(UB_j, i \ mod \ S_1)$$

donde UB_j representa el upper block que falta por examinar y $rank(B, j \cdot S_1 - 1)$ se responde con las respuestas almacenadas para los upper blocks.

Cada upper block se particiona en middle blocks de largo fijo $S_2 = \log n \log\log n$, y sus respuestas *rank* se almacenan de manera similar a los upper blocks. Además, en esta estructura se utiliza la tabla Q, descrita en el índice para *select*, y una nueva tabla T' parametrizada por segmentos P (que coinciden en tamaño con los middle blocks) y que contiene respuestas *rank* almacenadas para todas las posiciones de P.

Entonces, para responder $rank(i)$ en un middle block MB, sea $j = \lfloor i/S_3 \rfloor$ el número del chunk C que contiene el i-ésimo bit de B, donde $S_3 = \log n - 3 \log\log n$ (observe que S_3 coincide con el tamaño de los chunks). La respuesta está dada por:

$$rank(MB, i) = rank(MB, j \cdot S_3 - 1) + rank(C, j \ mod \ S_3).$$

Donde para calcular $rank(MB, j \cdot S_3 - 1)$ se consulta la tabla T, y para responder $rank(C, j \ mod \ S_3)$ se consulta la tabla Q. Hay que notar que se asume que los middle blocks están alineados a los chunks.

Es importante destacar que esta solución no ha sido implementada aún. Sin embargo, parece ser poco promisoria ya que resulta ser más complicada que la solución de Clark (Sección 3.1.3), que como se verá en los resultados experimentales de la Sección 5 está lejos de ser la más rápida. La complejidad adicional de esta estructura podría impactar negativamente el tiempo de respuesta, llegando a ser peor que la solución de Clark.

Si bien esta estructura es asintóticamente óptima en el uso de espacio extra, un análisis detallado en tamaños típicos de bitmaps muestra que en la práctica este sobrecosto no es tan bajo. Si se estima el sobrecosto de la estructura siguiendo fielmente la descripción [15] obtenemos que, para un bitmap de tamaño $n = 2^{30}$ con un 50 % de 1s, se tiene un sobrecosto aproximado de 44 % del espacio del bitmap. Como veremos, este sobrecosto es muy alto comparado con soluciones existentes. Incluso para bitmaps muy grandes, $n = 2^{40}$, el sobrecosto aún no baja de 32 %.

Cotas Inferiores

Además de la estructura de datos descrita en los puntos anteriores, Golynski establece cotas inferiores óptimas sobre el espacio requerido para indexar un bitmap de manera de resolver consultas *rank* y *select*, si se mantiene el bitmap original.

Estos resultados se obtienen en el denominado *indexing model*, que se resume en lo siguiente: se asume que B está almacenado en memoria externa y no tiene costo de espacio y que el índice I está almacenado en memoria principal y sí tiene costo de espacio. Por otro lado, acceder a un bit de I no tiene costo de tiempo, mientras que acceder a un bit de B tiene como costo una unidad de tiempo.

Entonces, en este modelo se demostró que cualquier algoritmo que realice consultas *rank* o *select* con costo de tiempo $O(\log n)$ debe tener un costo de espacio de $\Omega(n \log \log n / \log n)$. Además, en el caso de bitmaps dispersos, con cardinalidad (cantidad de 1s) m, cualquier algoritmo para responder *rank* o *select* accediendo a t bits de B tiene costo de espacio $\Omega((m/t) \log t)$.

Lo anterior implica que en el modelo RAM, donde se pueden aceder $O(\log n)$ bits en una operación, todo algoritmo de tiempo $O(1)$ requiere espacio $\Omega(n \log \log n / \log n)$.

3.2. Estructuras de datos Comprimidas

Las estructuras de datos comprimidas recodifican el bitmap B para utilizar menos espacio, soportando acceso directo y eficiente a los datos. Por lo tanto estas estructuras de datos son útiles si B es compresible, en particular, si tiene muchos o muy pocos 1s o si éstos no se distribuyen uniformemente.

3.2.1. Solución de Raman *et al.*

La solución de Raman *et al.* [29] utiliza $nH_0(B) + o(n)$ bits de espacio, donde $o(n)$ es la redundancia sobre la entropía. Con esto logra responder a *rank* y *select* en tiempo constante. Su funcionamiento se basa en particionar el bitmap de entrada B en bloques de largo $t{=}(\log n)/2$ que formarán diferentes clases; cada clase de largo t tiene k bits fijados en 1 y $t-k$ ceros. Una clase con k 1s tiene $\binom{t}{k}$ elementos y por lo tanto se requieren $\lceil \log \binom{t}{k} \rceil$ bits para nombrar un elemento de ella. Para identificar un bloque se utiliza el par $(k{,}r)$ donde k es el identificador de la clase, que está en el rango $0 \le k \le t$, usando $\lceil \log(t+1) \rceil$ bits, y r corresponde al índice dentro de la clase k y utiliza $\lceil \log \binom{t}{k} \rceil$ bits. En una tabla universal llamada $smallrank_k[r,i]$ se almacena $rank(x,i)$ para el bloque x identificado por el par $(k{,}r)$. El espacio total requerido para la representación de las clases k es $\sum_{i=0}^{\lceil n/t \rceil} \lceil \log(t+1) \rceil = O(n \log(t)/t) = O(n \log \log n / \log n)$ bits. Por otro lado, sea k_i la clase del i-ésimo bloque, entonces el espacio requerido por todos los bloques está dado por la siguiente fórmula [26], donde m es la cantidad total de 1s en B.

$$\begin{aligned} \lceil \log \tbinom{t}{k_1} \rceil + \cdots + \lceil \log \tbinom{t}{k_{\lceil n/t \rceil}} \rceil &\le \log \left(\binom{t}{k_1} \times \cdots \times \binom{t}{k_{\lceil n/t \rceil}} \right) + n/t \\ &\le \log \binom{n}{k_1 + \cdots + k_{\lceil n/t \rceil}} + n/t \\ &= \log \binom{n}{m} + n/t \\ &\le nH_0(B) + O(n/\log n). \end{aligned}$$

Además de la tabla *smallrank*, se almacena en la secuencia R la concatenación de todos los identificadores de clase k_i; y la secuencia S con la concatenación de todos los índices r_i correspondientes a los bloques clasificados en R. También se requieren estructuras para sumas parciales, una para R y otra para almacenar los largos de los r_i en S, denominada *posS*.

Para responder $rank(B,i)$ se calcula $sum(R, \lfloor i/t \rfloor) = \sum_{j=0}^{\lfloor i/t \rfloor} R_j$, que es la cantidad de 1s hasta antes del comienzo del bloque que contiene el i-ésimo bit, y luego se calcula *rank* dentro de ese bloque hasta la posición i, usando la tabla *smallrank*. Para esto se necesita el valor de r_i, el cual se obtiene usando $sum(posS, \lfloor i/t \rfloor)$ para determinar la posición inicial de r_i en S, y como los valores k_i y t son conocidos entonces se sabe cuántos bits es necesario leer. Para responder consultas *select*, se puede almacenar la misma información extra que en la solución de Clark (Sección 3.1.3), por ejemplo.

3.2.2. Implementación de Claude y Navarro

Claude y Navarro [9] presentaron una implementación práctica de la solución de Raman *et al.*. En la implementación fijan el tamaño de los bloques en $t = 15$, lo que implica que los identificadores de clase k requieren 4 bits para representar las clases entre 0 y 15. Se implementa la tabla *smallrank* usando enteros de 16 bits para las entradas, y para los punteros a los inicios de cada clase en la tabla. Las respuestas *rank* no se almacenan sino que se calculan en tiempo de consulta usando las entradas de *smallrank*, por lo que *smallrank* utiliza sólo 64 KB.

La tabla R se representa usando un arreglo compacto que usa 4 bits por campo, permi-

tiendo una extracción rápida de los valores. La tabla S almacena cada valor usando $\lceil \log \binom{t}{k} \rceil$ bits. Las sumas parciales se representan usando un muestreo de un nivel. Para la tabla R se muestrea la suma cada l valores, y se almacenan estos valores en una nueva tabla $sumR$, usando $\lceil \log m \rceil$ bits, donde m es la cantidad de 1s. Para obtener la suma parcial hasta la posición i se calcula $sumR[j] + \sum_{p=jl}^{i} k_p$, donde $j = \lfloor i/l \rfloor$, y la suma sobre los valores k_p se realiza sobre las entradas de la tabla R. Las posiciones en S se representan de la misma manera, usando una tabla $posS$ que almacena el muestreo de sumas usando $\lceil \log(\sum_{i=1}^{n/t} \lceil \log \binom{t}{k_i} \rceil) \rceil$ bits por campo. La posición del bloque i se calcula como $posS[j] + \sum_{p=jl}^{i} \lceil \log \binom{t}{k_p} \rceil$. Se precalculan los 16 valores posibles de $\lceil \log \binom{t}{k_p} \rceil$ para acelerar el último paso del cálculo.

Con estas estructuras las consultas *rank* se responden de la misma manera que en la solución de Raman *et al.*. Sin embargo, las respuestas a *select* están implementadas de manera más práctica y eficiente. Para responder $select(B, i)$ se realiza una búsqueda binaria sobre $sumR$, con el objetivo de encontrar el bloque más cercano tal que $sumR[l] \leq i$. Luego se recorre R buscando el bloque donde se espera encontrar el i-ésimo bit en 1, sumando k_p hasta que se exceda i. Finalmente se accede al bloque en la tabla *smallrank* y se recorre bit por bit hasta hallar el i-ésimo 1.

3.2.3. Implementación *sarray* de Okanohara y Sadakane

La solución propuesta por Sadakane [25] para trabajar con bloques dispersos tiene como entrada un bitmap $B[0, n-1]$ que tiene m 1s, donde $m \ll n$, es decir sólo para bitmaps con muy pocos 1s. Para comenzar la construcción de la estructura se define un directorio auxiliar $x[0...m-1]$ tal que $x[i] = select(B, i+1)$. Luego, dado un parámetro t, los valores de x se dividen en los bits superiores e inferiores, donde $z = \lceil \log t \rceil$ representa los bits superiores y $w = \lfloor \log(n/t) \rfloor$ los bits inferiores. Según la construcción definida, los bits inferiores se almacenan explícitamente en $L[0...m-1]$ usando $m \cdot w$ bits, mientras que los bits superiores se representan como un arreglo de bits $H[0...m+t-1]$, tal que $H[\lfloor x_i/2^w \rfloor + i] = 1$ y los otros valores son 0. Esto se puede ver como una codificación unaria de las diferencias entre los valores de los bits superiores, donde hay m 1s y $2^z = t$ 0s en H.

Para responder a una consulta *select*, se utiliza H y L, y la respuesta está dada por $select(B, i) = (select(H, i) - i) \cdot 2^w + L[i]$. El tamaño total de la estructura es $m + t + m(\lg(n/t))$ bits, que es una función convexa que se minimiza cuando $t = m \lg e \simeq 1.44m$, por lo que el tamaño está dado por $1.92m + m \log(n/m)$ bits cuando $t = 1.44m$. H es denso; ya que existen m 1s y $t = 1.44m$ 0s en H, y por ello se usa un *darray* para calcular *select* en H.

3.2.4. Solución de Golynski *et al.* para Bitmaps Dispersos

Una solución teórica para responder *rank* y *select* en tiempo constante es la solución de Golynski *et al.* [13]. Uno de los resultados principales de esta solución es que es eficiente en cuanto a espacio para bitmaps dispersos. Si $n = \Theta(m \log^c m)$, con constante $c > 0$, la estructura utiliza $\lceil \log \binom{n}{m} \rceil + O(m(\log \log m)^2 / \log m)$ bits, donde n es el largo del bitmap y m es la cantidad de 1s.

La estructura de datos se construye particionando el bitmap en bloques de largo $b = n(\log m)/m$, y luego particionando cada bloque en *chunks* de tamaño máximo $s = \log m/(2(c+1) \log\log m)$. Los chunks se asignan de izquierda a derecha tratando de asignar la mayor cantidad posible de ellos hasta el final del bloque. Es decir, un chunk siempre es parte de un único bloque.

Usando este particionamiento se definen algunas estructuras auxiliares que se describen a continuación: sea g la cantidad de chunks definidos en el bitmap, entonces se define el bitmap C como la concatenación de las representaciones codificadas de los chunks $c_1, c_2, \ldots, c_g$ (se usa la misma codificación que en la Sección 3.2.1). Otra estructura, CO, consiste en dos arreglos, donde el primero almacena el largo de la codificación de cada chunk c_j en C, y el segundo almacena los offsets entre cada $(\log m/(2 \log b))$ chunks. Además se define una tabla de búsqueda para cada secuencia posible de $(\log m/(2 \log b))$ offsets. La estructura CO permite calcular la suma de prefijos de los offsets de los primeros k chunks, con $1 \leq k \leq (\log m/(2 \log b))$ y decodificar el offset de un chunk individual.

Para dar soporte a consultas *rank* se definen dos nuevas estructuras auxiliares, CW y CC. Cada una corresponde a un arreglo con entradas de tamaño fijo $\log b$. En CW se almacenan los largos $n_1, n_2, \ldots, n_g$ de cada chunk. Para ello se utiliza el esquema clásico de dos directorios para encontrar el bloque correcto en el bitmap original. Luego, en CC se almacenan las cardinalidades (cantidad de 1s) $m_1, m_2, \ldots, m_g$ de cada chunk. En cada estructura, para calcular las sumas de prefijos dentro de un bloque se agrega una estructura de árbol multiario a cada bloque, lo que permite realizar la búsqueda en tiempo constante pues el tamaño de los bloques es polilogarítmico. Los árboles se representan implícitamente en una secuencia binaria.

Para responder $rank(B, i)$ se realiza lo siguiente:

- Se busca en CW para encontrar el chunk c_j que contiene la i-ésima posición del bitmap.
- Se busca en CC para encontrar la cantidad x de 1s contenidos en los primeros $j - 1$ chunks.
- Se calcula y como la cantidad de 1s en el chunk c_j hasta la posición i usando una búsqueda de tabla en c_j (usando C y CO).
- Se retorna $x + y$.

Para responder *select*, se usan las estructuras descritas para codificar implícitamente dos nuevos bitmaps, R_0 y R_1, que representan los runs de 0s y 1s en el bitmap original, respectivamente (agregando $10^{\ell-1}$ por cada run de largo ℓ). La respuesta se calcula realizando consultas *rank* y *select* sobre R_1 y R_0. En particular, $select_1(B, i) = select_1(R_0, rank_1(R_1, i - 1)) + i$ y $select_0(B, i) = select_1(R_1, rank_1(R_0, i - 1) + 1) + i$.

De este modo sólo se necesita responder $select_1$. Para calcularlo se crean estructuras auxiliares análogas a CW y CC para soportar *rank* en R_0 y R_1, y una estructura con la distribución de 1s en R_1 para soportar $select_1$ en R_1.

3.2.5. Compressed Dense Fully-Indexable Dictionaries

Otra solución de Golynski *et al.* [13] para el caso donde el largo del bitmap n no está condicionado a la cantidad de m de 1s, como en la sección anterior, es la estructura denominada *compressed dense fully-indexable dictionary*. Esta estructura de datos permite responder *rank* y *select* en tiempo constante utilizando $\lceil \log \binom{n}{m} \rceil + O(n \log \log n / (\log n)^2)$ bits de espacio.

Es importante destacar que esta solución es la primera en bajar la redundancia de $\lceil \log \binom{n}{m} \rceil + O(n \log \log n / (\log n))$ bits (resultado de Raman *et al.* [29]) a $\lceil \log \binom{n}{m} \rceil + O(n \log \log n / (\log n)^2)$ bits, gracias a la recodificación del bitmap.

La estructura se basa en la construcción de *códigos informativos*, que son una codificación en la cual se pueden deducir ciertas características del elemento codificado tan solo observando una pequeña cantidad de bits de la codificación.

Esta estructura se construye particionando el bitmap en bloques, superbloques y megabloques. Los bloques tienen tamaño $k = \lceil (\log n)/2 \rceil$ bits, los superbloques contienen $t = \lceil \alpha \log n / \log \log n \rceil$ bloques cada uno (con $\alpha > 0$), y los megabloques contienen t superbloques cada uno.

La codificación informativa de un superbloque se representa como la concatenación de la codificación de la cantidad de 1s del superbloque, con la codificación de las posiciones de los 1s en el superbloque. Para la representación de los megabloques, considere R_i como el código de la cantidad de 1s del superbloque i. Entonces el megabloque está compuesto por un *overview* que consiste en la concatenación de los R_i de los superbloques que lo componen, seguido de las codificaciones de cada superbloque, utilizando un tamaño fijo (los superbloques más pequeños son rellenados hasta alcanzar el tamaño adecuado). Lo anterior permite elaborar una tabla de búsqueda sobre los superbloques de un megabloque.

Para responder *rank* o *select*, se realiza una búsqueda de tabla en el megabloque para así reducir la consulta a una operación *rank* o *select* sobre un superbloque en particular. Además, para cada bloque se almacena su codificación y se crea una tabla de búsqueda para decodificarlo en tiempo constante. Además, para mapear una consulta *rank* o *select* sobre el bitmap original a una consulta sobre los megabloques, se definen los bitmaps D_0 y D_1 (usando la estructura de datos de la sección anterior) como una codificación de la cantidad de 0s y 1s, respectivamente, de los megabloques. Se define $D_{x \in \{0,1\}} = 0^{N_1} 1 0^{N_2} 1 \ldots$, donde N_i es la cantidad de bits x en el i-ésimo megabloque. Cada megabloque está separado por un 1, por lo que tanto D_0 como D_1 tienen una cantidad de 1s igual a la cantidad de megabloques.

Finalmente, y considerando lo anterior, una consulta *rank* o *select* sobre el bitmap original se reduce a una consulta sobre los bitmaps D_0 y D_1, para así realizar una consulta en los megabloques, que a su vez se reduce a una consulta sobre los superbloques, que finalmente es contestada usando las tablas de búsqueda y la información codificada de los bloques.

Como en el caso de la solución de Golynski para bitmaps densos, no existe una implementación de esta estructura. Para un bitmap de tamaño 2^{30}, una estimación muy gruesa y

optimista del sobrecosto de espacio, basada en la descripción [13], arroja un valor de aproximadamente 13 % del bitmap original por sobre la entropía. Esto se reduce a 10 % para $n = 2^{40}$. En cambio la implementación que se presenta en la Sección 4.3, que es más sencilla que la presentada en esta sección, muestra un sobrecosto alrededor del 6 % sobre la entropía para $n = 2^{30}$, con buenos tiempos de respuesta (situación que se mantiene para todas las densidades de bitmap consideradas). Por lo tanto esta solución de Golynski no parece ser promisoria, pues su sobrecosto estimado es peor, y es probable que su tiempo de respuesta también lo sea, dada la complejidad de la estructura.

3.2.6. Solución de Pătraşcu

Otra solución teórica es la propuesta por Pătraşcu [27], que se basa en la utilización de recursión con el objetivo de eliminar la redundancia. Se puede aplicar a muchos problemas, en particular a bitmaps compresibles con soporte para operaciones *rank* y *select*.

El ejemplo mostrado por Pătraşcu para explicar esta técnica es el problema de almacenar una secuencia de *trits*, donde un trit $x \in \{0, 1, 2\}$. Se desea poder codificar la secuencia utilizando M bits más un *spill* que está en el rango $\{1 \ldots K\}$.

Supongamos que se desea codificar, por ejemplo, w trits, que tienen entropía $w \cdot \log 3$. De estos trits se desea extraer M bits de información (ej. $M = \lfloor w \cdot \log 3 \rfloor$) que se almacenarán, luego los restantes bits $\delta = w \cdot \log 3 - M$ son pasados al siguiente nivel de recursión. En el segundo nivel, se agregan w bloques, para los que se debe almacenar $w \cdot \delta$ bits de información. Luego se almacenan M' bits (ej. $M' = \lfloor w \cdot \delta \rfloor$) y se pasan $\delta' = w \cdot \delta - M'$ bits al siguiente nivel, etc.

Dado que se desea disminuir la redundancia, no se puede gastar un bit extra por bloque, por lo que δ puede no ser un entero. En ese caso δ se aproxima a $\log K$, donde K es un entero. Esto significa que pasar δ bits de información (el spill) es lo mismo que pasar un número en $\{0 \ldots K - 1\}$. Esto produce redundancia de $\log K - \delta$ bits. Notar que si δ es la mejor aproximación, δ está entre $\log(K - 1)$ y $\log K$.

En el segundo nivel de recursión, el problema es representar un arreglo de n/w valores de $\{0 \ldots K - 1\}$, y lo mismo ocurre para los diferentes niveles de recursión. En el último nivel se permite gastar una cantidad entera de bits.

Augmented B-Trees

En el caso general de aplicación a las estructuras de datos compactas, Pătraşcu define una estructura de datos denominada *augmented B-tree*, o simplemente *aB-tree*, que utiliza la técnica de recursión ya descrita. A continuación se describe el aB-tree para el caso particular de una secuencia binaria con soporte para *rank* y *select*, para algún $B \geq 2$ que representa la aridad del árbol.

- El árbol representa una secuencia binaria S de tamaño n, potencia de B. Los elementos de S se almacenan en las hojas.
- A cada nodo interno se le agrega la suma de las hojas de su subárbol (lo que equivale a

la suma de sus hijos), y la cantidad de 1s en su subárbol. En general se puede guardar cualquier resumen de la información del subárbol en cada nodo.

- El algoritmo de consulta examina los hijos de la raíz, decide en cuál hijo continuar la recursión, examina todos los valores de los hijos de ese nodo, y continúa recursivamente. Las consultas se responden al alcanzar una hoja, y se logra responder en tiempo constante si los valores de todos los hijos están en una misma palabra de memoria.

Dada la recursión presente en la estructura es posible comprimir el aB-tree con sólo 2 bits de redundancia.

Aplicación a *rank* y *select*

Para responder *rank* y *select* en tiempo $O(t)$ para un arreglo de tamaño n con m 1s se realiza lo siguiente:

- Sea $r = (\log n/t)^{\Theta(t)}$. Se particiona el arreglo en bloques de tamaño r.
- Cada bloque se almacena en un aB-tree. El tiempo para responder *rank* y *select* dentro de cada aB-tree es $O(t)$.
- Para cada bloque se almacenan sus posiciones iniciales en el bitmap original, y se almacenan las respuestas *rank* precomputadas hasta el inicio de cada bloque.
- Se crea una estructura auxiliar de predecesores, basada en el muestreo para *rank*, que ayuda a responder consultas *select*.

Para responder *rank* o *select*, se examinan las estructuras auxiliares para encontrar el bloque donde se encuentra la respuesta y luego se realiza una consulta en el aB-tree correspondiente. Para responder en tiempo $O(t)$ se requiere una redundancia de $n/(\log n/t)^t + O(n^{3/4})$ bits.

La solución de Pătraşcu puede considerarse una extensión de la solución de Golynski (Sección 3.2.4), pero que permite cualquier polilogaritmo en el denominador de la redundancia, con costo de tiempo $O(t)$, donde t es el grado del polilogaritmo. En trabajo posterior Pătraşcu y Viola [28] demuestran que esta redundancia es óptima, incluso cuando se recodifica el bitmap.

Desde un punto de vista práctico, la relación entre la redundancia y el tiempo de consulta hace improbable que la solución sea competitiva, pues una disminución cada vez menos notoria de la redundancia tiene un perceptible costo en tiempo.

Capítulo 4

Nuevas Implementaciones

En el presente capítulo se exponen las nuevas soluciones de esta tesis, describiendo la creación de las estructuras de datos y cómo logran responder a consultas *rank* y/o *select*. Primero se presenta una estructura de datos que combina un particionamiento regular en posiciones (conveniente para *rank*) y un particionamiento regular en 1s (conveniente para *select*), con lo que se logra responder a las consultas *rank* y *select*. Luego se presenta una estructura de datos plana de un nivel de directorio para responder consultas de tipo *select* para bloques densos y dispersos, junto con una variación de la estructura de datos que utiliza codificación `Gamma` y `Simple9`. Posteriormente se explica una nueva estructura de datos comprimida basada en la propuesta de Raman *et al.* [29] para trabajar con bitmaps dispersos, y finalmente se presenta una modificación a esta estructura de datos para responder de forma eficiente consultas de tipo *select*.

4.1. Estructura de Muestreo Combinado

Esta estructura de datos combina dos esquemas de partición: el primero es un particionamiento regular en posiciones, esto es, dividir el bitmap de entrada B en bloques de tamaño fijo S_r. Luego, se almacena un sampling de respuestas $rank(B, i)$ desde el comienzo de B, hasta el final de cada bloque. Así, para el i-ésimo bloque, se tiene $sampleRank[i] = rank(B, i \cdot S_r)$. En el segundo esquema, se crean bloques con S_s 1s cada uno (excepto el último bloque que podría tener menos 1s). Estos bloques son de largo variable, dependiendo de la densidad del bitmap. Los bloques parten con un 1, con esto se logra evitar procesar 0s innecesarios cuando se escanea un bloque desde el inicio. En *sampleSelect*, en vez de guardar la posición, p, del bit correspondiente en el muestreo, se almacena la posición del byte (en el bitmap original) que contiene dicho bit (usando 29 bits), y la cantidad de 1s desde el comienzo de ese byte hasta la posición p (usando 3 bits). Tanto S_r como S_s deben ser potencias de dos. Por consistencia, se ha definido $sampleRank[0] = sampleSelect[0] = 0$. Ambos muestreos son similares a los de las de las soluciones tradicionales para *rank* y *select*. La novedad de esta estructura es cómo combinarlos para resolver cada tipo de consulta.

En la Figura 4.1, se presenta un ejemplo de construcción de la estructura de muestreo combinado con parámetros $S_s = S_r = 16$. Por legibilidad, se marcan los bytes de la secuencia

usando las líneas verticales continuas pequeñas. El muestreo en posiciones, basado en S_r, está marcado con las líneas contínuas más largas que las usadas para bytes, y las líneas punteadas indican el muestreo regular en 1s, basado en S_s.

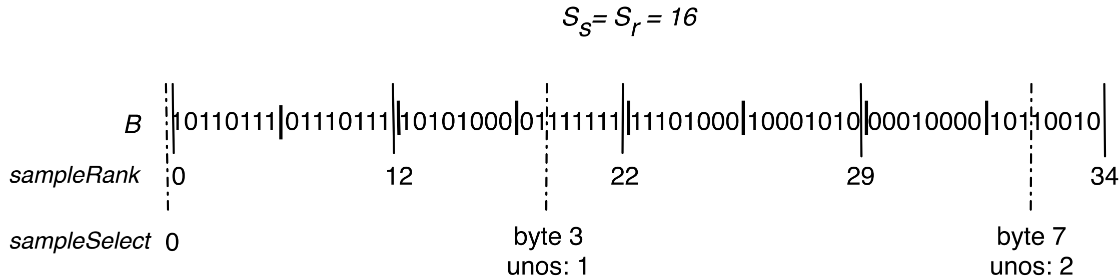


Figura 4.1: Ejemplo de muestreo combinado para una secuencia de bits utilizando $S_s = S_r = 16$.

La tabla *sampleRank* está compuesta por los valores $\{0,\ 12,\ 22,\ 29,\ 34\}$. Por su lado, *sampleSelect* en la primera posición almacena un 0, en la segunda posición usa 29 bits para almacenar el valor 3 (indicando el tercer byte) y los otros tres bits para el valor 1 (unos existentes desde el comienzo del byte). De la misma manera, para la tercera posición se almacena la posición del byte y la cantidad de unos desde el comienzo del byte.

4.1.1. Respondiendo Consultas *rank* y *Select*

Para responder la consulta $rank(B, i)$ se realizan los siguientes pasos, la Figura 4.4 muestra el pseudocódigo asociado:

1. Sea $j \leftarrow sampleRank[\lfloor i/S_r \rfloor]$ la cantidad de 1s hasta la respuesta precomputada del bloque anterior más cercana a i en *sampleRank*.
2. Se debe escanear el bitmap para incluir las posiciones restantes en la respuesta. Se trata de explotar *sampleSelect* para obtener una posición de partida más cercana a i.
3. Sea $r \leftarrow \lfloor j/S_s \rfloor$, el bloque en *sampleSelect* que contiene el j-ésimo 1.
4. Se ejecuta $while((sampleSelect[r+1] \gg 3) < (i/8))\ r \leftarrow r+1$ para obtener el valor precomputado *sampleSelect* más cercano que anteceda a i. Notar que se compara el byte que tiene la posición del i-ésimo 1, con la posición del byte almacenado en *sampleSelect*.
5. Ahora se considera $\max(j, r \cdot S_s)$, y sea *ini* la posición inicial desde donde se debe comenzar a escanear el bitmap original para incluir las posiciones restantes en la respuesta.
   (a) Si j es mayor que $r \cdot S_s$, se fija $ini \leftarrow \lfloor i/S_r \rfloor \cdot S_r$. Esto implica que se han contado j 1s.

(b) De otra forma, se fija $ini \leftarrow (sampleRank[r] \gg 3) \cdot 8$. Esto implica que se han contado $r \cdot S_s - (sampleSelect[r] \; \& \; 7)$ 1s. Observar que el valor de $sampleSelect[r] \; \& \; 7$ corresponde a la cantidad de bits 1s existentes desde el comienzo del byte.

6. Es importante notar que en ambos casos del punto anterior, se está alineado a los bytes de B. Ahora, se tienen $\lfloor (i - ini + 1)/8 \rfloor$ bytes restantes para ser escaneados. Dado que se está alineado, simplemente se escanea utilizando la operación *popcount*[1] en el byte. En el caso que el último byte no necesite ser escaneado por completo, se escanean las correspondientes posiciones bit a bit.

En la Figura 4.2 se muestra cómo responder la consulta $rank(B, 59)$ para la estructura de datos construida en la Figura 4.1. Siguiendo los pasos descritos se obtiene la respuesta a la consulta, aprovechando el muestreo para *select*. La flecha negra indica la posición inicial considerando sólo el muestreo para *rank*. Sin embargo, al usar el muestreo para *select* se obtiene una mejor posición inicial, indicada por la flecha blanca. La respuesta está dada por la expresión $(2 \cdot 16 - sampleSelect[2]\&3) + popcount(1011)$. Como la posición inicial obtenida es 2, se conoce que hay $2 \cdot \mathcal{S}_s = 2 \cdot 16$ 1s hasta la posición indicada por $sampleSelect[2]$, menos los 1s que sobran en el último byte, ya que se comienza a hacer *popcount* desde ahí, escaneando sólo los bits indicados con la zona sombreada.

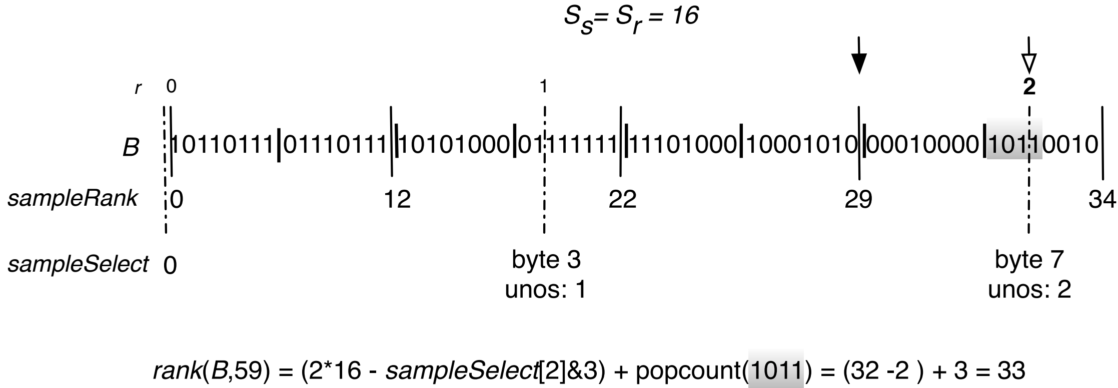


Figura 4.2: Ejemplo de consulta *rank* en la estructura de muestreo combinado.

Para responder una consulta $select(B, i)$, se realizan los siguientes pasos, cuyo pseudocódigo se muestra en la Figura 4.5:

1. Sea $pos \leftarrow \lfloor i/S_s \rfloor$, el índice del bloque en *sampleSelect* más cercano a i.
2. Sea $j \leftarrow (sampleSelect[pos] \gg 3) \cdot 8$ la posición inicial (en bits) del byte donde comienza el bloque *pos*.
3. Para obtener una mejor posición de partida y evitar un escaneo innecesario en el bitmap, se trata de explotar los valores almacenados en *sampleRank*.
4. Sea $r \leftarrow \lfloor j/S_r \rfloor$ el bloque en *sampleRank* que contiene la j-ésima posición.
5. Se ejecuta $while(sampleRank[r+1] < i) \; r \leftarrow r + 1$ para obtener el índice del bloque en *sampleRank* que contiene el i-ésimo 1.

[1] *Popcount* es una tabla de valores precomputados que entrega la cantidad de 1s en un byte, similar a la descrita en la Sección 3.1.2.

6. Ahora se considera el $\text{máx}(j, r \cdot S_r)$, y sea *ini* la posición más cercana a i según el muestreo combinado.
   (a) Si j es mayor que $r \cdot S_r$, se fija $ini \leftarrow j$. No se gana información adicional de *sampleRank*, y quedan $i - pos \cdot S_s + (sampleSelect[pos]\ \&\ 7)$ bits en 1 por encontrar.
   (b) De otra forma, se fija $ini \leftarrow r \cdot S_r$, que indica que se han contado $sampleRank[r]$ 1s hasta la posición *ini*, por lo que faltan $i - sampleRank[r]$ 1s por encontrar.
7. Notar que nuevamente se está alineado a los bytes de B ya que en ambos casos del punto anterior *ini* apunta al comienzo de un byte. Ahora, se deben encontrar los 1s restantes. Para hacer esto, se escanean los bytes del bitmap, comenzando por el byte que comienza en *ini* y aplicando la operación *popcount* hasta el byte con el i-ésimo bit. En caso que este último byte no necesite ser escaneado completo, se escanea bit por bit.

La Figura 4.3 ilustra la respuesta a la consulta $select(B, 30)$, también sobre la misma estructura que para *rank*. En este caso, la posición inicial utilizando *sampleSelect*, indicada con la flecha blanca, no es la más conveniente como punto de partida. Al considerar *sampleRank* se obtiene una mejor posición inicial, indicada con la flecha negra, que minimiza la cantidad de bits, indicados por la zona sombreada, que deben ser escaneados individualmente. Note que el escaneo bit por bit para la secuencia 0001 retorna 3 porque indica la posición, indexada desde 0, del último bit escaneado.

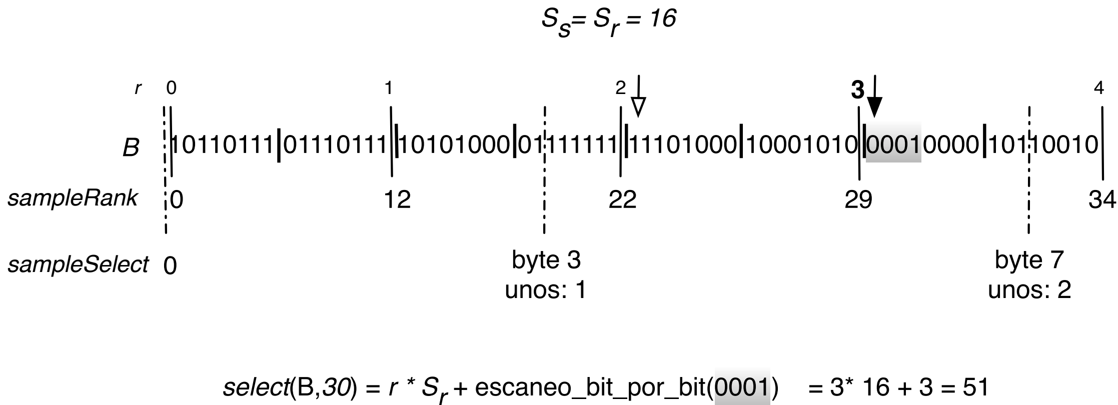


Figura 4.3: Ejemplo de consulta *select* en la estructura de muestreo combinado.

```
1: function RANK(i)
2:     j ← sampleRank[⌊i/S_r⌋]
3:     r ← ⌊j/S_s⌋
4:     while (sampleSelect[r + 1] ≫ 3) < (i/8) do
5:         r ← r + 1
6:     end while
7:     ini, m ← 0
8:     if j > r ∗ S_s then
9:         m ← j
10:        ini ← ⌊i/S_r⌋ ∗ S_r
11:    else
12:        m ← r ∗ S_s − (sampleSelect[r] & 7)
13:        ini ← (sampleSelect[r] ≫ 3) ∗ 8
14:    end if
15:    posicionesRestantes ← (i − ini + 1)
16:    bytesRestantes ← ⌊posicionesRestantes/8⌋
17:    posIni ← ⌊ini/8⌋
18:    posFin ← posIni + bytesRestantes
19:    while posIni < posFin do
20:        m ← m + popcount(B[posIni])
21:        posIni ← posIni + 1
22:    end while
23:    bitsRestantes ← posicionesRestantes & 7
24:    if bitsRestantes > 0 then
25:        m ← m + popcount(B[posIni]) ≪ (8 − bitsRestantes)
26:    end if
27:    return m
28: end function
```

Figura 4.4: Algoritmo para responder *rank* utilizando la estructura de muestreo combinado. B se considera como un arreglo de bytes.

```
1: function SELECT(i)
2:     pos ← ⌊i/S_s⌋
3:     j ← (sampleSelect[pos] ≫ 3) * 8
4:     r ← ⌊j/S_r⌋
5:     while sampleRank[r + 1] < i do
6:         r ← r + 1
7:     end while
8:     if j ≥ r * S_r then
9:         ini ← j
10:        bitsRestantes ← i − pos * S_s + (sampleSelect[pos] & 7)
11:    else
12:        ini ← r * S_r
13:        bitsRestantes ← i − sampleRank[r]
14:    end if
15:    posIni ← ⌊ini/8⌋
16:    repeat
17:        bitsRestantes ← bitsRestantes − popcount(B[posIni])
18:        posIni ← posIni + 1
19:        ini ← ini + 8
20:    until bitsRestantes ≤ 0
21:    ultimoByte ← B[posIni − 1]
22:    posBit ← 7
23:    while bitsRestantes ≤ 0 do
24:        if (ultimoByte ≫ posBit)&1 then
25:            bitsRestantes ← bitsRestantes + 1
26:        end if
27:        posBit ← posBit − 1
28:        ini ← ini − 1
29:    end while
30:    return ini + 1
31: end function
```

Figura 4.5: Algoritmo para responder *select* utilizando la estructura de muestreo combinado. B se considera como un arreglo de bytes.

4.2. Select 1 Nivel

En esta estructura de datos el bitmap se particiona en bloques de S 1s. Los bloques se clasifican como densos o dispersos de acuerdo a un parámetro d, y se utiliza un nivel de directorio para almacenar respuestas *select* precomputadas para bloques dispersos. En el caso de los bloques densos, las respuestas a *select* se buscan directamente en el bitmap de entrada.

4.2.1. Creación de la Estructura

Para un bitmap de n bits $B[1, n]$, con m bits en 1, se realizan los siguientes pasos:

1. El bitmap se divide en bloques con S bits en 1. Con esto se consigue tener $b = m/S$ bloques de tamaño r, con r variable considerando que depende de la distribución de los bits 1 del bitmap. Para estos bloques se almacena en una tabla R las posiciones en donde termina cada bloque, con respecto al bitmap de entrada, utilizando $m/S \log n$ bits. Con ello, en tiempo de consulta se puede conocer además el tamaño de un bloque, por lo que r no necesita ser almacenado.
2. Para los b bloques interesa conocer cuáles son densos y cuáles dispersos. Si el tamaño r de un bloque es tal que $S/r \leq d$, con d un parámetro entre 0 y 1, entonces el bloque se considera disperso, y en otro caso se considera denso. Para marcar cuáles son los bloques densos y cuáles son los bloques dispersos se utiliza un nuevo bitmap C utilizando m/S bits, tal que el i-ésimo bit de C está asociado al i-ésimo bloque. Si dicho bit es 0, el bloque es denso y si el bit es 1 el bloque es disperso.
3. Para cada bloque disperso se almacenan las S respuestas *select* relativas al inicio del bloque en una nueva tabla A, que a lo más tendrá $\frac{n \cdot d}{S} S \log n = n \cdot d \cdot \log n$ bits. Con esto se logra optimizar la consulta *select* para los bloques dispersos. En otro caso, sólo se conoce a qué bloque pertenece el bit buscado, y se requiere un escaneo de a lo más S/d bits.

En resumen, el espacio total utilizado por la estructura de datos está dado por $(m/S + n \cdot d) \log n$ bits obteniendo un tiempo de respuesta de $O(S/d)$. Esto permite elegir el compromiso deseado entre espacio extra y tiempo de consulta. En la Figura 4.6 se muestra una estructura de ejemplo.

4.2.2. Consulta *Select* Utilizando la Estructura

Al momento de responder la consulta *select*(B, i) sobre la secuencia de n bits en $B[1, n]$ se realizan los siguientes pasos:

1. Obtener el bloque $b = \lfloor i/S \rfloor$ al que pertenece el i-ésimo bit. La posición inicial de este bloque es $p = R[b-1] + 1$, pues corresponde a un bit más que el fin del bloque anterior.
2. Consultar en C el bit en la posición b, para saber si el bloque es denso o disperso.

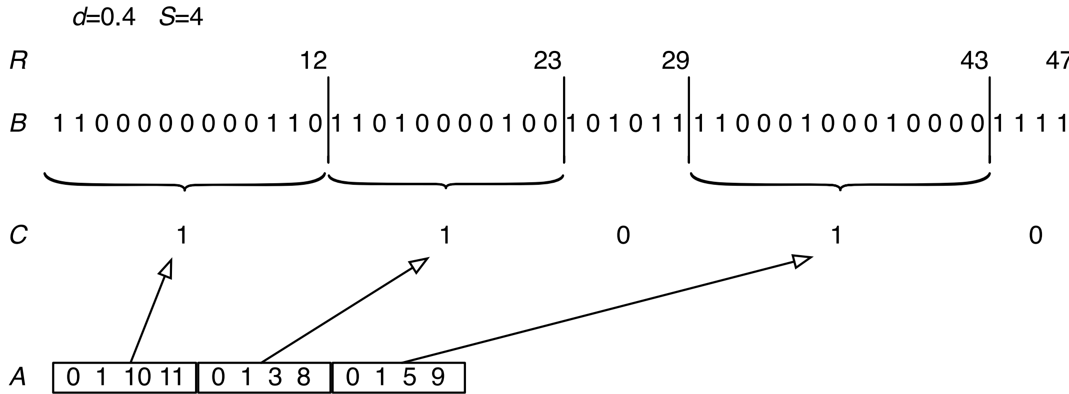
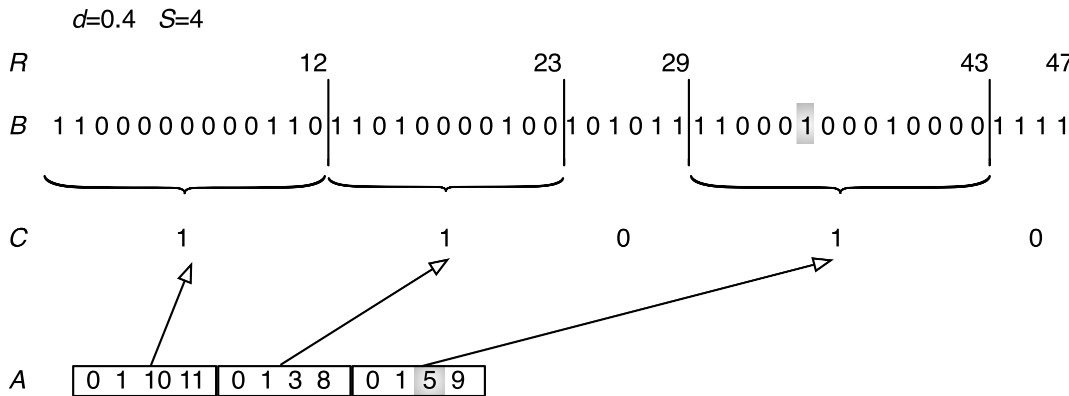


Figura 4.6: Ejemplo de estructura Select 1 Nivel para una secuencia de bits. Se consideran los parámetros $d = 0.4$ y $S = 4$.

3. En el caso de los bloques densos, se escanea el bloque b en B, recorriéndose S/d bits. Estos se procesan de a bytes con la operación *popcount*.
4. En el caso de ser un bloque disperso se realiza $rank(C, b)$ (para lo cual se utiliza la estructura de datos clásica para responder *rank* y *select* propuesta por González *et al.* [16]), con el objetivo de obtener la posición en el directorio A que tiene almacenada la posición directa de $select(B, i)$, aprovechando la estructura de datos bajo la cual se dividió.

La Figura 4.7 muestra un ejemplo de consulta $select(B, 15)$ para la estructura de ejemplo del punto anterior. En el ejemplo se utilizan los directorios ya que la respuesta se encuentra en un bloque disperso.



select(B,15) = R[2] + A[(rank(C,3) - 1) * S + (15 mod S) - 1] + 1
= 29 + A[(2 * 4) + 3-1] + 1
= 29 + A[10] + 1
= 29 + 5 + 1
= 35

Figura 4.7: Ejemplo de consulta *select* sobre la estructura Select 1 Nivel para una secuencia de bits. Se consideran los parámetros $d = 0.4$ y $S = 4$.

```
1: function SELECT(i)
2:     b ← ⌊i/S⌋
3:     p ← R[b − 1] + 1
4:     bitDensidad ← ObtenerBit(C, b)
5:     if bitDensidad = 0 then
6:         return popcount_secuencial(B, p, i − (b ∗ S))
7:     else
8:         index ← rank(C, b) − 1
9:         return p − 1 + A[index ∗ S + (i mod S) − 1] + 1
10:    end if
11: end function
```

Figura 4.8: Algoritmo para responder *select* en la estructura con un nivel de directorio. La función $popcount_secuencial(B, p, k)$ cuenta en B la cantidad de posiciones utilizando operaciones *popcount* desde la posición p hasta encontrar el k-ésimo 1. A es el arreglo con las respuestas *select* almacenadas en el directorio. $ObtenerBit(C, b)$ retorna el valor del b-ésimo bit en C.

4.2.3. Select 1 Nivel con Codificación

Esta estructura de datos es una variación de *Select 1 Nivel* que, en vez de utilizar A para acelerar las respuestas en bloques dispersos, reutiliza el mismo espacio utilizado por el bloque en el bitmap de entrada para almacenar las respuestas *select* precomputadas. Con esto se logra disminuir el espacio utilizado por la estructura de datos.

La construcción de la estructura de datos es idéntica a la versión descrita en la Sección 4.2.1, salvo para los bloques dispersos. En este caso, se sobreescribe el bloque disperso con una codificación de las respuestas *select* dentro del bloque. Un requisito adicional para clasificar un bloque como disperso es que tenga espacio suficiente para almacenar las respuestas codificadas. La idea es minimizar la búsqueda lineal dentro del bloque, con las respuestas precalculadas, y no incurrir en gasto extra de espacio, reutilizando la misma memoria del bloque. Para hacer un uso óptimo de este espacio se codifican las respuestas a almacenar.

Se implementaron dos esquemas de codificación, uno utiliza codificación *Gamma* (Sección 2.4.1), y el otro utiliza codificación *Simple9* (Sección 2.4.2). En ambos esquemas, en vez de almacenar el valor completo de las respuestas $select(i)$, se codifica la primera respuesta del bloque y luego se almacenan las diferencias acumulativas respecto a la respuesta anterior. Para obtener la respuesta asociada al l-ésimo 1 del bloque, se decodifican y suman las l primeras diferencias. El pseudocódigo es idéntico al de la Figura 4.8, salvo por la obtención de la respuesta *select* relativa al bloque disperso. La Figura 4.9 muestra este algoritmo.

En la Figura 4.10a se presenta la primera etapa de la construcción de la estructura, donde se crean los bloques cada S 1s, y se verifica si los bloques son densos o dispersos. Luego, en la Figura 4.10b, para los bloques dispersos se almacenan las respuestas codificadas de *select* del bloque, en el segmento correspondiente de la secuencia original. Aquí `f` se refiere

ya sea a la función de codificación Gamma o Simple9, descritas anteriormente. Notar que en esta estructura de datos ya no existe la tabla A, como en la estructura de la Figura 4.6.

1: **function** SELECT(i)
2: $b \leftarrow \lfloor i/S \rfloor$
3: $p \leftarrow R[b-1]+1$
4: $bitDensidad \leftarrow ObtenerBit(C,b)$
5: **if** $bitDensidad = 0$ **then**
6: **return** $popcount_secuencial(B,p,i-(b*S))$
7: **else**
8: $index \leftarrow rank(C,p)$
9: **return** $p-1+DecodificarRespuesta(b, i \bmod S)$
10: **end if**
11: **end function**

Figura 4.9: Algoritmo para responder *select* en la estructura para *select* con un nivel de directorio usando codificación. $DecodificarRespuesta(b,k)$ retorna la k-ésima respuesta relativa almacenada en el bloque b, decodificando y sumando las k primeras diferencias.

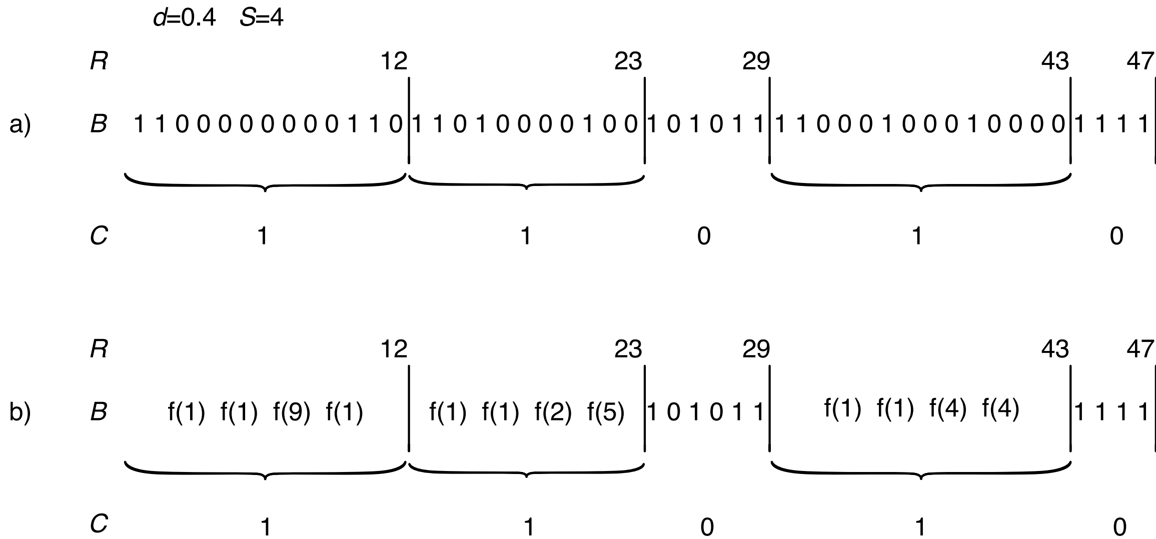


Figura 4.10: Ejemplo de construcción de la estructura *select* de 1 nivel utilizando codificación. En a) se muestra la secuencia original donde solamente se han marcado los bloques densos y dispersos. En b) se muestra la secuencia modificada donde cada bloque disperso almacena sus respuestas codificadas por alguna función `f`.

4.3. Estructura para Bitmaps Dispersos

En la propuesta de Raman *et al.* [29] hay dos factores que influyen en el costo extra de espacio. En primer lugar, la tabla R que almacena los identificadores de clase k de cada

bloque, utiliza $n \cdot \lceil \log(t+1)/t \rceil$ bits, por lo que al usar valores grandes de t se minimiza este costo de espacio. Sin embargo, el segundo factor es la tabla *smallrank*, que utiliza $2^t t$ bits, por lo que se debe seleccionar cuidadosamente un t que considere estos factores. Por ejemplo, en la implementación de Claude y Navarro (Sección 3.2.2) se utiliza $t = 15$, obteniendo un costo extra de $\lceil \log(15+1)/15 \rceil = \lceil 4/15 \rceil \approx 27\,\%$, y una tabla universal de 128 KB.

Esta nueva estructura de datos para bitmaps dispersos se basa en la idea propuesta por Raman *et al.* [29]. La idea es almacenar las respuestas $rank(B, i)$ para los superbloques y los valores (k, r) correspondientes a cada clase, al igual que en la estructura de datos original, pero sin almacenar la tabla *smallrank*. En lugar de esto, se genera el valor que se requiere de ella en tiempo de consulta. Como en esta solución no se almacena la tabla *smallrank*, es posible generar clases con largo $t = 31$ y $t = 63$, obteniendo un espacio extra de tan sólo 16 % y 9.5 %, respectivamente[2].

De manera similar a la implementación de Claude y Navarro, en la tabla S se almacenan los identificadores r de cada bloque con k 1s utilizando $\lceil \log \binom{t}{k} \rceil$ bits para cada valor almacenado. Como los valores de k son variables en cada bloque, S almacena una codificación de largo variable de los valores de r. Para acelerar el acceso al identificador r en el último bloque consultado al responder las consultas *rank* o *select*, se mantiene la tabla *PosS* donde se almacena la posición del bit en S donde comienzan los identificadores para cada superbloque.

A continuación se explica el proceso de asignación de los valores (k, r) a un bloque y la reconstrucción de un bloque utilizando estos valores.

4.3.1. Codificación en Tiempo de Construcción

La tabla *smallrank* almacena las respuestas *rank* parciales para cada clase. La expresión $smallrank_k[r, j]$ identifica la respuesta *rank* para el r-ésimo bloque de la clase k hasta la posición j (relativa al bloque). Dado que en esta nueva solución no se tiene la tabla *smallrank*, para responder las consultas *rank*/*select*, es necesario reconstruir el bloque representado por el par (k, r). Durante la construcción de la estructura se ordenan los bloques de una clase considerando la presencia de 1s en las posiciones menos significativas. Sea $x = x_0, ..., x_{t-1}$ un bloque de largo t cuyo bit menos significativo está en la posición $(t-1)$ y que tiene k 1s. Luego, para asignar el identificador r al bloque, se ejecuta el algoritmo de la Figura 4.11, que se explica a continuación:

1. Sea $j \leftarrow t - 1$.
2. Dado un x que tiene largo t, con k 1s, $r \in [rmin, rmax]$, donde $rmin \leftarrow 0$ y $rmax \leftarrow \binom{t}{k} - 1$.
3. Si $x_j = 1$ se decrementa el valor de k, y se tienen k 1s para las siguientes j posiciones. Esto lleva a que $rmax$ tome el valor $rmax \leftarrow rmin + \left(\binom{j}{k} - 1\right)$.
4. En otro caso, se fija $rmin \leftarrow rmax - \left(\binom{j}{k} - 1\right)$
5. Si $rmax = rmin$, entonces $r \leftarrow rmin$ es el identificador asignado al bloque y se termina.

[2] No se usan clases de largo mayor a 63, pues en procesadores actuales de 32 y 64 bits se requiere aritmética especial, con mucho peor desempeño, para procesar múltiples palabras de memoria.

En otro caso, se decrementa j en uno, y se continúa desde el paso 3.

```
1: function AsignarIdBloque(t, k, x)
2:     j ← t − 1
3:     rmin ← 0, rmax ← (t k) − 1
4:     while rmin < rmax do
5:         if x_j = 1 then
6:             k ← k − 1
7:             rmax ← rmin + ((j k) − 1)
8:         else
9:             rmin ← rmax − ((j k) − 1)
10:        end if
11:        j ← j − 1
12:    end while
13:    return rmin
14: end function
```

Figura 4.11: Algoritmo para codificar en tiempo de construcción el identificador r asignado a un bloque.

Es importante observar que en esta codificación los identificadores se asignan primero a las secuencias de bits que tienen los 1s en las posiciones menos significativas. Por ejemplo, si $t = 4$ y $k = 2$, se tienen las combinaciones $\{0011, 0101, 1001, 0110, 1010, 1100\}$ a las que se asignan los identificadores $\{0, 1, 2, 3, 4, 5\}$ respectivamente. En particular, si $\binom{t}{k} = 1$ la clase tiene sólo una combinación con identificador $r = 0$.

4.3.2. Decodificación en Tiempo de Consulta

A continuación se explica cómo se reconstruye un bloque con $x = x_0, \ldots, x_{t-1}$ bits, conociendo solamente el par (k, r). La reconstrucción se realiza bit por bit comenzando por el bit menos significativo. Para este proceso se asume que r fue asignado con el algoritmo se la sección anterior. El procedimiento general de reconstrucción del bloque se explica en los siguientes pasos:

1. Sea $j \leftarrow t$, $rmin \leftarrow 0$, y $rmax \leftarrow \binom{t}{k} - 1$.
2. Se decrementa j en 1.
3. Se asume que $x_j = 1$. Luego, debido a la suposición se define $rmax' \leftarrow rmin + \left(\binom{j}{k-1} - 1\right)$ como el nuevo límite posible.
4. Para verificar la suposición previa, se compara r con $rmax'$. La suposición es verdadera si y sólo si $r \leq rmax'$.
5. Si $r \leq rmax'$ entonces se decrementa $k \leftarrow k - 1$, y se asigna $rmax \leftarrow rmax'$.
6. En otro caso, $x_j = 0$ y se fija $rmin \leftarrow rmax - \left(\binom{j}{k} - 1\right)$.
7. Si $k = 0$ entonces las posiciones restantes sólo contienen 0s. Si no, se continúa desde el paso 2.

La reconstrucción del bloque se realiza sólo según sea necesario para responder a una consulta *rank* o *select*. En el primer caso se reconstruye hasta la posición w relativa al bloque, y en el segundo caso se reconstruye hasta encontrar el w-ésimo 1. El pseudocódigo para cada caso está dado en las Figuras 4.12 y 4.13 respectivamente.

Require: $w < t$
1: **function** ReconstruirRank(k, r, w)
2: $j \leftarrow t$
3: $rmin \leftarrow 0$, $rmax \leftarrow \binom{t}{k} - 1$, $rmax' \leftarrow rmax$
4: $i \leftarrow 0$, $c \leftarrow 0$
5: **while** $k > 0$ **do**
6: $j \leftarrow j - 1$
7: $rmax' \leftarrow rmin + \left(\binom{j}{k-1} - 1\right)$
8: **if** $r \leq rmax'$ **then**
9: $rmax \leftarrow rmax'$
10: $k \leftarrow k - 1$
11: $c \leftarrow c + 1$
12: **else**
13: $rmin \leftarrow rmax - \left(\binom{j}{k} - 1\right)$
14: **end if**
15: **if** $i = w$ **then**
16: **return** c
17: **end if**
18: $i \leftarrow i + 1$
19: **end while**
20: **end function**

Figura 4.12: Algoritmo para reconstruir el bloque identificado por r para encontrar cuántos 1s hay en ese bloque hasta la posición w relativa al bloque.

4.3.3. Respondiendo *rank* y *Select*

Para esta sección se considera una secuencia binaria B de largo n, particionada en bloques de tamaño t, con s bloques en cada superbloque, y la tabla de sumas parciales $sumR$ (como se definió en la Sección 3.2.1). Para responder la consulta $rank(B, i)$, se realiza lo siguiente:

1. Sea $sb = \lfloor i/(s \cdot t) \rfloor$ el superbloque que contiene el i-ésimo bit. La cantidad de 1s acumulada hasta este superbloque es $rankB = sumR[sb]$.
2. Se conoce que $b = sb \cdot s$ es el bloque de inicio del superbloque sb, y que $q = \lfloor i/t \rfloor$ es el bloque que contiene el i-ésimo bit.
3. La respuesta está dada por la expresión:

$$rankB + \left(\sum_{c=b}^{q-1} k_c \right) + ReconstruirRank(k_q, r_q, i \bmod t)$$

donde k_c es la cantidad de 1s en el bloque c-ésimo (valores almacenados en la tabla R).

```
Require: w < t
 1: function ReconstruirSelect(k, r, w)
 2:     j ← t
 3:     rmin ← 0, rmax ← (t choose k) − 1, rmax′ ← rmax
 4:     i ← 0, c ← 0
 5:     while k > 0 do
 6:         j ← j − 1
 7:         rmax′ ← rmin + ((j choose k−1) − 1)
 8:         if r ≤ rmax′ then
 9:             c ← c + 1
10:             if c = w then
11:                 return i
12:             end if
13:             rmax ← rmax′
14:             k ← k − 1
15:         else
16:             rmin ← rmax − ((j choose k) − 1)
17:         end if
18:         i ← i + 1
19:     end while
20: end function
```

Figura 4.13: Algoritmo para reconstruir el bloque identificado por r para encontrar la posición del w-ésimo 1 relativo al bloque.

Para responder a una consulta $select(B, i)$, se realiza lo siguiente:

1. Se ejecuta una búsqueda binaria sobre las respuestas *rank* almacenadas para los superbloques, para obtener el índice sb del superbloque que contiene el i-ésimo 1.
2. Al igual que en el algoritmo previo, conociendo sb se obtiene $b = sb \cdot s$, el bloque de inicio del superbloque sb.
3. Se define $m = sumR[sb]$ como los 1s encontrados hasta la posición de inicio de sb.
4. Se incrementa m agregando los k_c 1s de cada bloque c en sb, comenzando por b, hasta que se cumple $m > i$. Sea q este último bloque y p la cantidad de bloques recorridos sin incluir el bloque q.
5. La posición del i-ésimo 1 está dada por:

$$t \cdot (sb \cdot s + p) + ReconstruirSelect(k_q, r_q, i - (m - k_q))$$

.

```
1: function RANK(i)
2:     sb ← ⌊i/(s * t)⌋
3:     rankB ← sumR[sb]
4:     b ← sb * s, q ← ⌊i/t⌋
5:     pos ← PosS[sb]
6:     for c ← b to q − 1 do
7:         k_c ← R[c]
8:         rankB ← rankB + k_c
9:         pos ← pos + ⌈log (t choose k_c)⌉
10:    end for
11:    k_q ← R[q]
12:    r_q ← ObtenerIndice(S, pos, pos + ⌈log (t choose k_q)⌉ − 1)
13:    return rankB + ReconstruirRank(k_q, r_q, (i mod t))
14: end function
```

Figura 4.14: Algoritmo para responder una consulta *rank* en la estructura para bitmaps dispersos. $ObtenerIndice(S, ini, end)$ obtiene el identificador r del bloque donde está la posición i, leyendo la tabla S entre las posiciones ini y end. Recuerde que los valores $\lceil \log \binom{t}{k_c} \rceil$ están precalculados.

```
1: function SELECT(i)
2:     sb ← BusquedaBinaria(sumR, 0, ⌊n/s⌋, i)
3:     b ← sb * s
4:     posFin ← PosS[sb]
5:     m ← sumR[sb]
6:     c ← b
7:     posLocal ← sb * s * t
8:     while m ≤ i do
9:         k_c ← R[c]
10:        m ← m + k_c
11:        posLocal ← posLocal + t
12:        posFin ← posFin + ⌈log (t choose k_c)⌉
13:        c ← c + 1
14:    end while
15:    k_q ← R[c]
16:    r_q ← ObtenerIndice(S, posFin − ⌈log (t choose k_c)⌉, posFin − 1)
17:    return posLocal − t + ReconstruirSelect(k_q, r_q, i − (m − k_q))
18: end function
```

Figura 4.15: Algoritmo para responder una consulta *select* en la estructura para bitmaps dispersos. Recuerde que los valores $\lceil \log \binom{t}{k_c} \rceil$ están precalculados.

4.3.4. Combinando con el Esquema de Particionamiento Regular en 1s y Select de 1 Nivel

Con el objetivo de optimizar aún más el tiempo de respuesta a una consulta *select*, se modifica la estructura de datos para incorporar la idea del muestreo combinado y Select de 1 Nivel, combinando las ideas de las Secciones 4.1 y 4.2. El problema es que al tener bitmaps dispersos o con runs de 0s muy largos, el muestreo para *rank* obliga a recorrer innecesariamente estas subsecuencias. Por lo tanto, la modificación consiste en agregar un particionamiento regular cada S_s 1s de manera de precalcular las respuestas en los bloques muy largos (dispersos), evitando recorrer los valores precalculados de *rank* en ellos. Además, el particionamiento regular en 1s evita la búsqueda binaria en el arreglo completo. Con el particionamiento regular en 1s se obtienen m bloques de largo variable según la densidad del bitmap, es decir, cada bloque m_k tiene largo l_k, y posición inicial pos_k.

Durante la construcción de la estructura, se clasifican los m bloques como densos o dispersos, según un parámetro $d \in [0, 1]$. Si $S_s/l_k > d$, entonces el bloque m_k es denso. En otro caso el bloque es disperso.

En una tabla *SampleSelect*, para cada bloque se almacena cierta información según la clasificación del bloque. Si el bloque m_k es denso, entonces se almacena en $SampleSelect[k] = \lfloor pos_k/l_{sb} \rfloor \cdot l_{sb}$, que equivale a la posición inicial del superbloque donde se encuentra el k-ésimo 1, donde l_{sb} es el largo del superbloque del muestreo para *rank* que contiene la posición pos_k. En cambio, si el bloque es disperso, entonces $SampleSelect[k] = y + y'$, donde y es una constante tal que $y > |B|$, e y' es un contador que comienza en 1 y se incrementa para cada bloque disperso.

Finalmente, para cada bloque disperso se almacena en una tabla *sparseBlocks* las respuestas *select* respecto del bitmap B. Si el bloque m_k es disperso y tiene asociado el contador y'_k, entonces sus respuestas en *sparseBlocks* comienzan en la posición $y'_k \cdot S_s$.

Para responder a una consulta $select(B, i)$ se realiza lo siguiente:

1. Sea $m = \lfloor i/S_s \rfloor$ el bloque del muestreo para *select* que contiene el i-ésimo bit.
2. Consideramos el valor de $SampleSelect[m]$
3. Si $SampleSelect[m] > y$ entonces el bloque es disperso, y su contador asociado es $y' = SampleSelect[m] - y$. Por lo tanto, la respuesta a la consulta *select* está dada por $sparseBlocks[(y' \cdot S_s) + (i \bmod S_s)]$.
4. En otro caso, sea $j \leftarrow SampleSelect[m]$ la posición inicial del superbloque que contiene el i-ésimo 1.
5. Sea $r \leftarrow \lfloor j/l_{sb} \rfloor$ es el índice en *sumR* del superbloque que comienza en la posición j.
6. Ahora se trata de explotar *sumR* para obtener una mejor posición de partida.
7. Se ejecuta $while(sumR[r+1] < i)\ r \leftarrow r+1$ para obtener el índice del superbloque más cercano a i.
8. Ahora se considera el $\max(j, r \cdot l_{sb})$, sea *ini* la posición más cercana a i según el muestreo combinado, y sea *bitsRestantes* la cantidad de 1s por encontrar.

(a) Si j es mayor a $r \cdot l_{sb}$, se fija $ini \leftarrow \lfloor j/l_{sb} \rfloor \cdot l_{sb}$, y $bitsRestantes \leftarrow i - sumR[\lfloor j/l_{sb} \rfloor]$.

(b) De otra forma, se fija $ini \leftarrow r \cdot l_{sb}$, por lo que $bitsRestantes \leftarrow i - sumR[r]$.

9. Se decrementa $bitsRestantes$ restando los k_c 1s de cada bloque c en el superbloque $\lfloor ini/l_{sb} \rfloor$, hasta que se cumple $bitsRestantes \leq 0$. Sea q este último bloque y p la cantidad de bloques recorridos sin incluir el bloque q.
10. La posición del i-ésimo 1 está dada por $ini + p \cdot t + ReconstruirSelect(k_q, r_q, bitsRestantes + k_q)$.

Capítulo 5

Resultados Experimentales

En este capítulo se presentan los resultados experimentales obtenidos con las nuevas estructuras de datos, y se comparan con las implementaciones existentes presentadas en el Capítulo 3. Los resultados se exponen en el mismo orden en que fueron presentadas las soluciones en el Capítulo 4.

Todos los experimentos fueron realizados en un computador que tiene 2 procesadores Intel Core2 Duo, con 3 Ghz y 6 MB de caché y 8 GB de RAM. El sistema operativo es Ubuntu 8.04.04. Se utilizó el compilador `gcc` con todas las optimizaciones. Los experimentos se realizaron sobre bitmaps de entrada de distintas densidades (cantidad de 1s del bitmap).

5.1. Resultados de Estructura de Muestreo Combinado

Con el objetivo de evaluar el rendimiento de esta nueva estructura de datos, se compara con las implementaciones de las estructuras de datos planas existentes[1]:

- `Clark` : estructura de Clark (Sección 3.1.3), que agrega un 60 % extra al bitmap de entrada (implementado por R. González).
- `RG 37%`: implementación de González *et.al* (Sección 3.1.2) con dos niveles de directorios, bloques y superbloques, que utiliza un 37 % de espacio extra.
- `RG 1 Nivel`: implementación de González *et. al.* (Sección 3.1.2) con un nivel de directorio. El espacio utilizado depende de la configuración de sus parámetros.
- `Vigna rank9sel`: implementación de Vigna (Sección 3.1.5) que usa `rank9` como estructura de datos básica y agrega `select9` para responder *select* en tiempo constante.
- `Vigna rank9b`: implementación de Vigna que usa `rank9` como estructura de datos básica y ejecuta *select* usando *hinted binary search* (Sección 3.1.5).
- `Vigna Simple select`: implementación de Vigna de la estructura de datos *Simple select* (Sección 3.1.5).

[1]Las implementaciones de González *et.al* se obtuvieron mediante comunicación personal con Rodrigo González. Las implementaciones de Vigna se encuentran disponibles en `http://sux.di.unimi.it/sux-0.7.tar.gz`. `Darray`, adaptado a la librería libcds, se encuentra en `http://libcds.recoded.cl/`.

- `darray`: implementación de la estructura de datos *darray* de Okanohara y Sadakane (Sección 3.1.4)

En los experimentos se considera la estructura de muestreo combinado considerando sus parámetros S_r y S_s, que definen los muestreos para *rank* y *select*, respectivamente. Recuerde que en esta versión las consultas *rank* y *select* sacan provecho de los dos muestreos.

Junto con esto, para estudiar la influencia de responder las consultas utilizando ambos muestreos, se crea una versión modificada de la estructura de muestreo combinado donde cada tipo de consulta sólo utiliza su muestreo correspondiente. En cada caso, según corresponda, se evalúan diferentes valores para S_r y S_s, en el rango de $[2^8, 2^{13}]$.

Se hicieron dos experimentos: en el primero, se ejecutaron consultas *select* y *rank* de forma independiente para evaluar su desempeño, con diferentes densidades del bitmap de entrada. En el segundo experimento, se emula una aplicación real que requiera los dos tipos de consulta, por lo que se ejecutan consultas combinadas variando la proporción de consultas *rank* y *select*.

A continuación se detallan las configuraciones de los experimentos realizados y los resultados obtenidos en ellos.

5.1.1. Consultas Independientes

En estos experimentos se generaron bitmaps aleatorios de densidad (proporción de 1s) 10 %, 50 % y 90 %. Luego se ejecutaron 1.000.000 de consultas aleatorias *rank* y *select*. Se realiza un experimento para cada valor S_r en $[2^8, 2^{13}]$, donde S_r queda fijo y S_s varía entre $[2^8, 2^{13}]$.

Como se muestra en las Figuras 5.1, 5.2 y 5.3, la nueva técnica provee un continuo de desempeño en espacio y tiempo según se varía S_r y S_s. La zona más interesante es donde el espacio extra es inferior al 5 % y los tiempos son menores a 0.2 microsegundos (para *rank*) y 0.3 microsegundos (para *select*). En particular, usando $S_r = 1024$ y S_s=8192 se alcanza un espacio extra de alrededor de 3 % y tiempos en el rango 0.1-0.2 microsegundos (para *rank*) y 0.2-0.25 (para *select*). Este tradeoff entre espacio y tiempo no había sido logrado anteriormente.

Al considerar el desempeño sin utilizar S_s se observa que el uso del muestreo de *select* no genera una gran diferencia para *rank* (por lo que se podría usar la técnica básica que no utiliza *sampleSelect*). Sin embargo el desempeño sin utilizar S_r muestra que *select* se acelera considerablemente usando el muestreo de *rank*, aunque la diferencia disminuye a medida que aumenta la densidad del bitmap. Como resultado, la nueva estructura de datos combinada es mejor que juntar dos estructuras independientes, como ocurre en la mayoría de las estructuras de datos en la literatura.

Por otro lado, para *rank*, estructuras de datos como `rank9b`, `rank9sel` y `darray` ocupan significativamente más espacio y no son más rápidas que la nueva combinación de 3 % en espacio. Sólo la estructura `RG` 37 % [16] es considerablemente más rápida y aún así requiere 10 veces más espacio. La estructura `RG 1 Nivel` alcanza un desempeño muy similar al de la

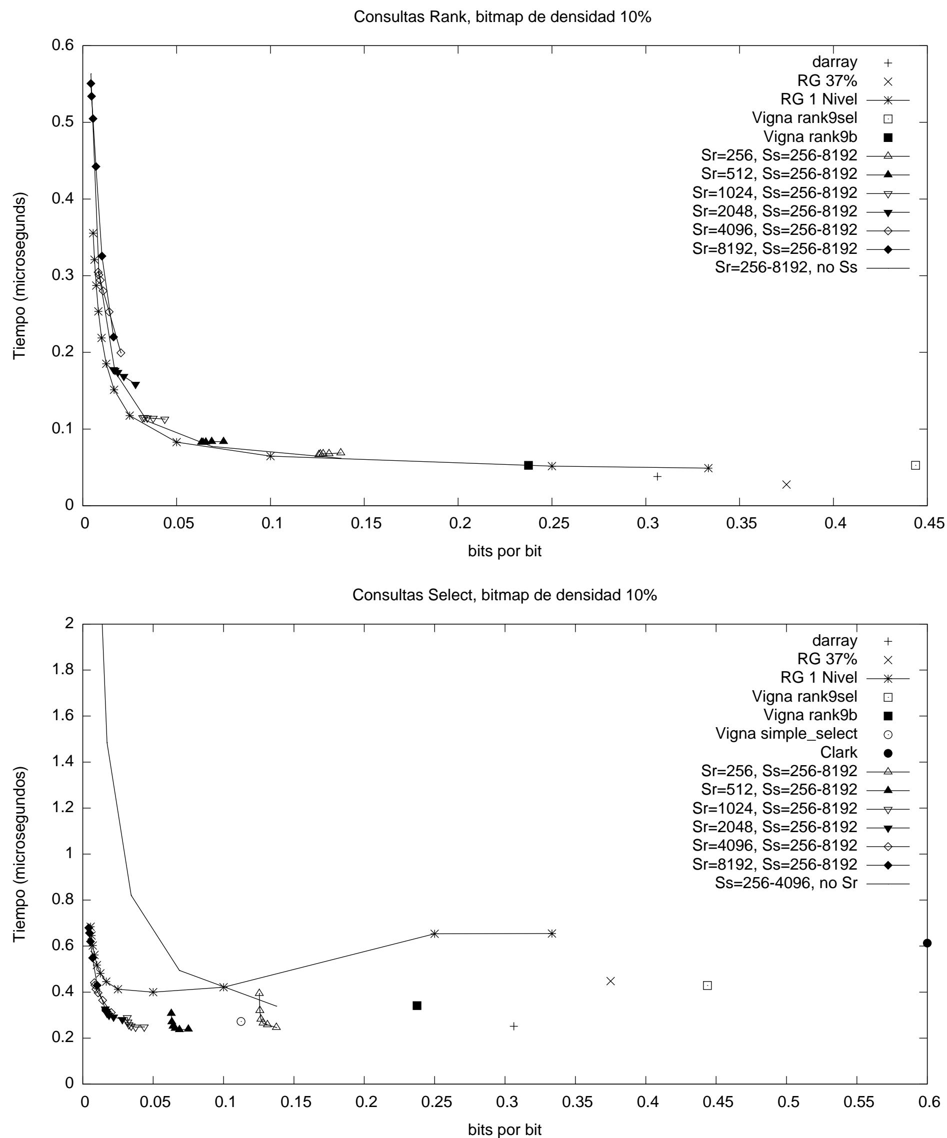


Figura 5.1: Consultas *rank* y *Select* para bitmaps planos con densidad de 10 %. El espacio corresponde al espacio extra sobre el bitmap.

nueva estructura.

Sin embargo, para *select* `RG 1 Nivel` tiene un peor desempeño. Otras estructuras de datos que no alcanzan mejor tiempo que la nueva combinación (que utiliza 3 % espacio extra) pero que usan más espacio son: `rank9b`, `rank9sel`, `RG` 37 %, `darray` y `Clark`. El mejor desempeño, aún así no competitivo con el trabajo presentado en esta tesis, es el de `Simple select`. Este logra tiempo similar pero utiliza 2 a 3 veces más espacio que nuestra combinación de 3 % de espacio.

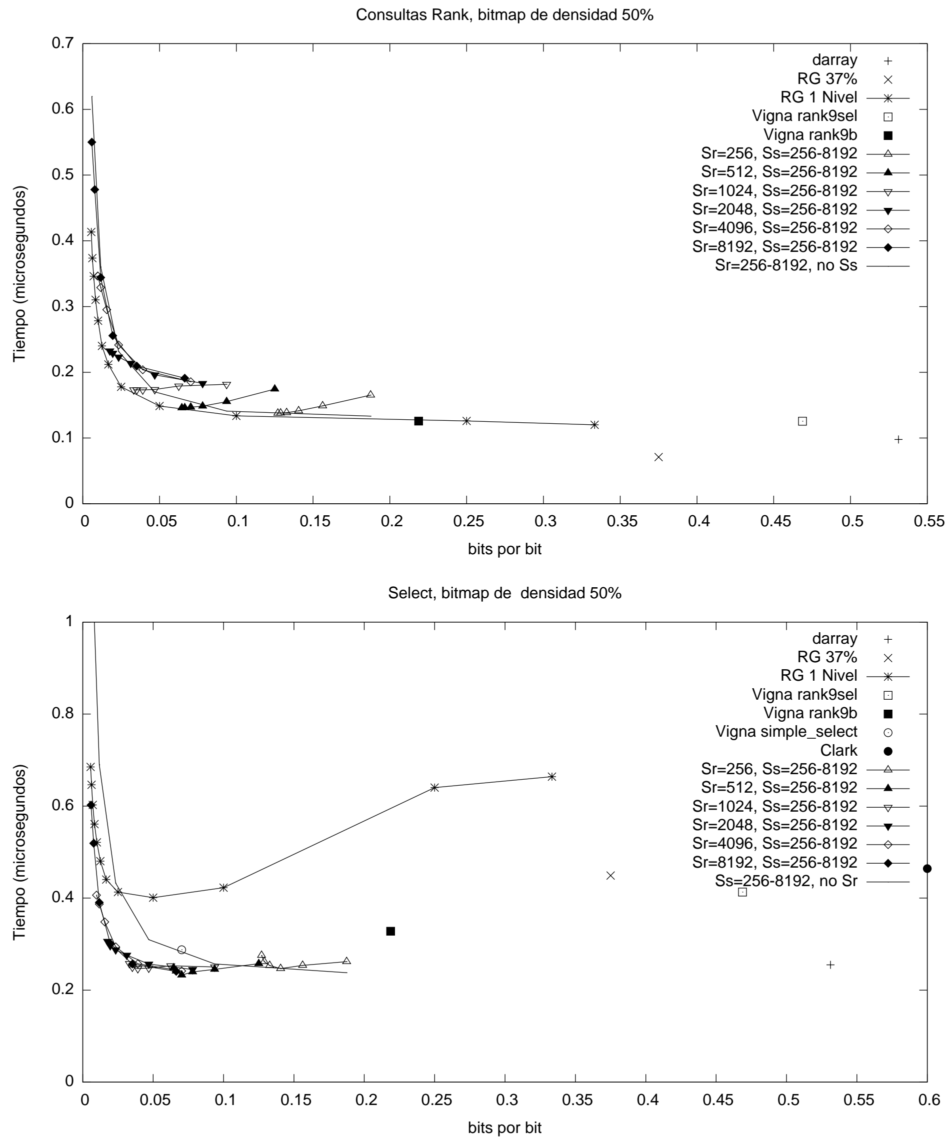


Figura 5.2: Consultas *rank* y *Select* para bitmaps planos con densidad de 50 %. El espacio corresponde al espacio extra sobre el bitmap.

5.1.2. Consultas Mezcladas

En este experimento se generó un bitmap aleatorio con densidad de 50 %. Luego, al igual que en las pruebas anteriores, se ejecutaron 1.000.000 de consultas aleatorias con las siguientes proporciones de consultas *select*/*rank*: 10 %/90 %, 30 %/70 %, 50 %/50 %, 70 %/30 % y 90 %/10 %. Además se incluye una variante de la estructura presentada en este trabajo con el mismo valor para ambos muestreos, $S_r = S_s$.

Para aleatorizar el tipo de consulta, se construye un arreglo de tamaño 1.000.000 que se completa con 1s y 0s, donde 1 indica que se debe ejecutar una consulta de tipo *rank* y 0 una consulta de tipo *select*, luego se recorre este arreglo en tiempo de ejecución para ejecutar la consulta respectiva.

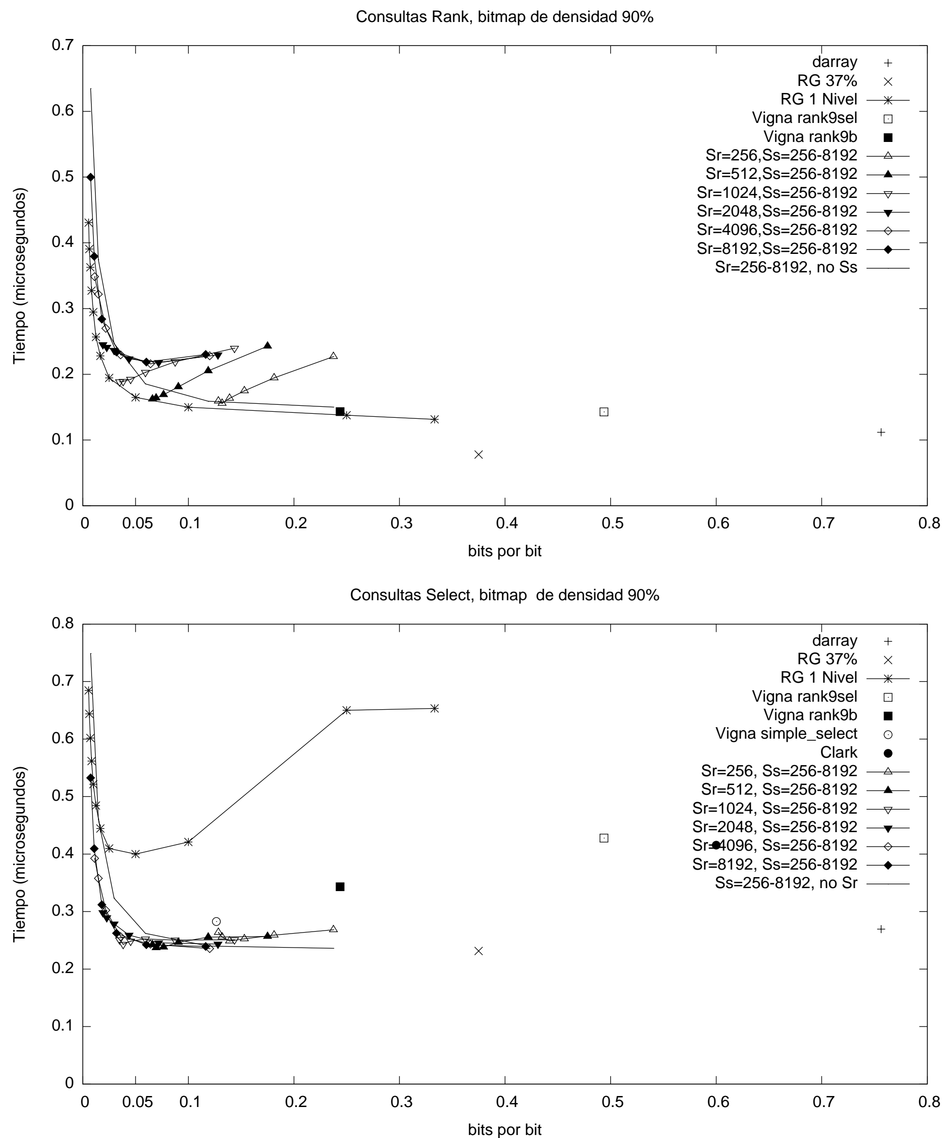


Figura 5.3: Consultas *rank* y *Select* para bitmaps planos con densidad de 90 %. El espacio corresponde al espacio extra sobre el bitmap.

Los resultados se muestran en las Figuras 5.4, 5.5 y 5.6. Se puede observar que al tener más de 10 % de consultas *select* la nueva estructura no tiene paralelo. Esto es porque la operación *select* en las otras estructuras de datos es mucho más lenta que *rank* y por lo tanto una pequeña fracción de esas consultas afecta el tiempo total. En cambio, la nueva estructura mantiene un desempeño cercano a los 0.2 microsegundos utilizando alrededor de 3 % de espacio extra, lo que se alcanza con $S_r = 1024$ y $S_s = 8192$. La variante que utiliza el mismo valor para ambos muestreos es generalmente subóptima; es conveniente gastar más espacio en el muestreo de *rank*.

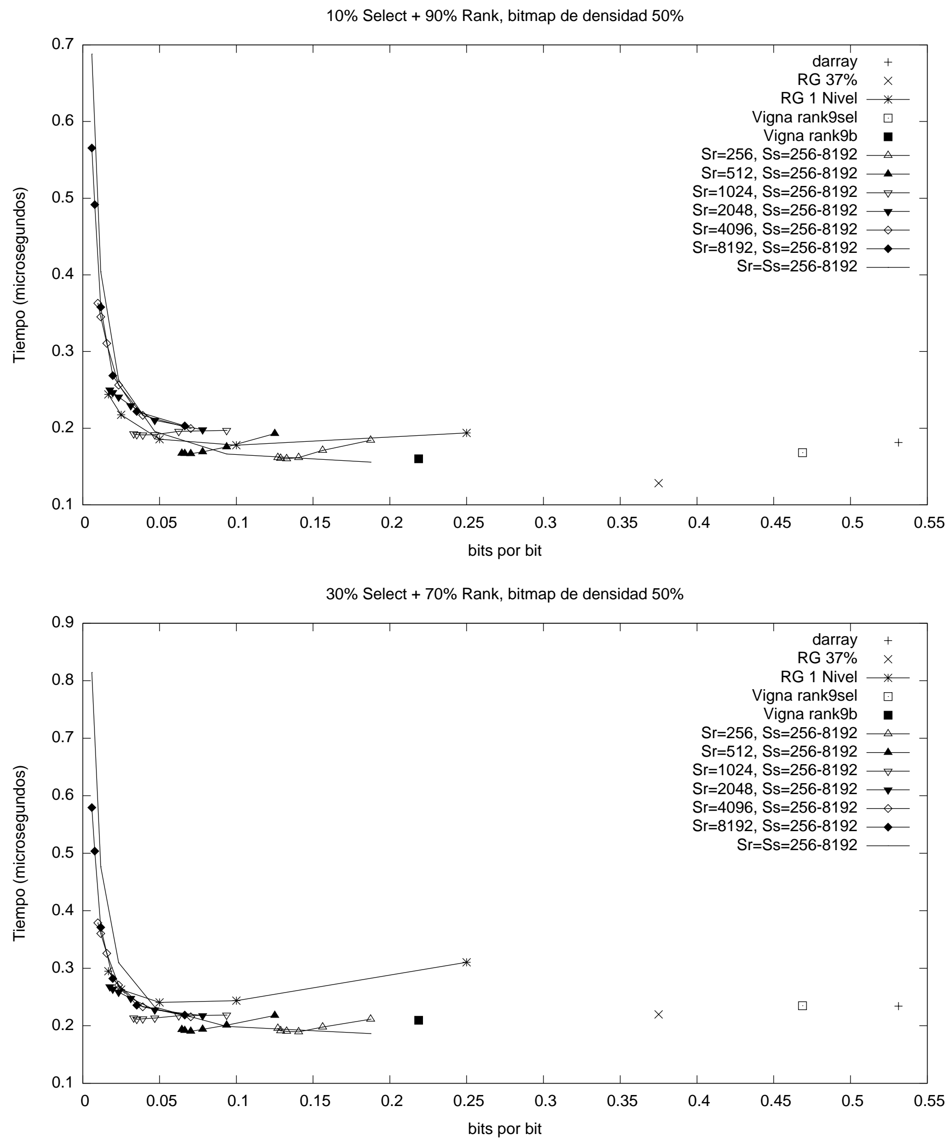


Figura 5.4: Resultado de experimentos de consultas *select*/*rank* mezcladas para las proporciones *select*/*rank* de 10 %/90 % y 30 %/70 %.

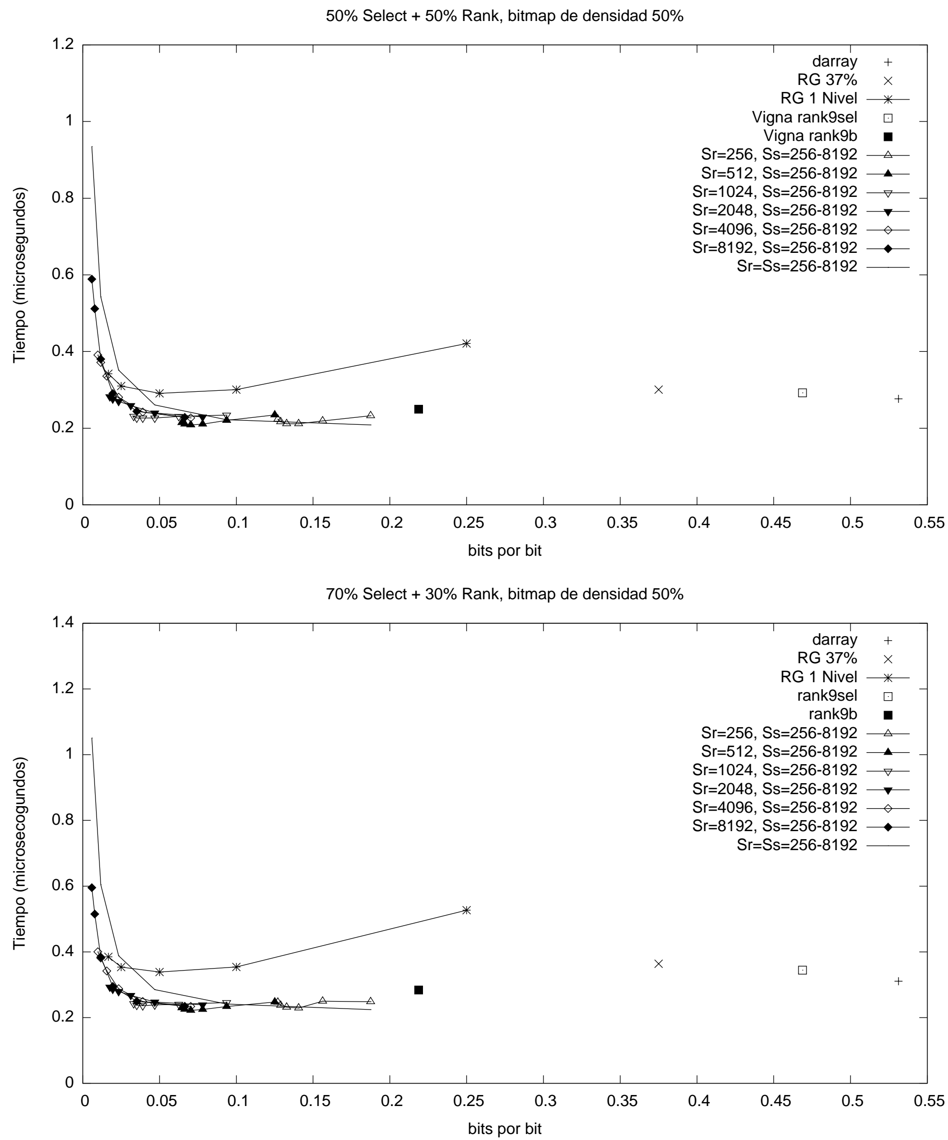


Figura 5.5: Resultado de experimentos de consultas *select*/*rank* mezcladas para las proporciones *select*/*rank* de 50 %/50 % y 70 %/30 %.

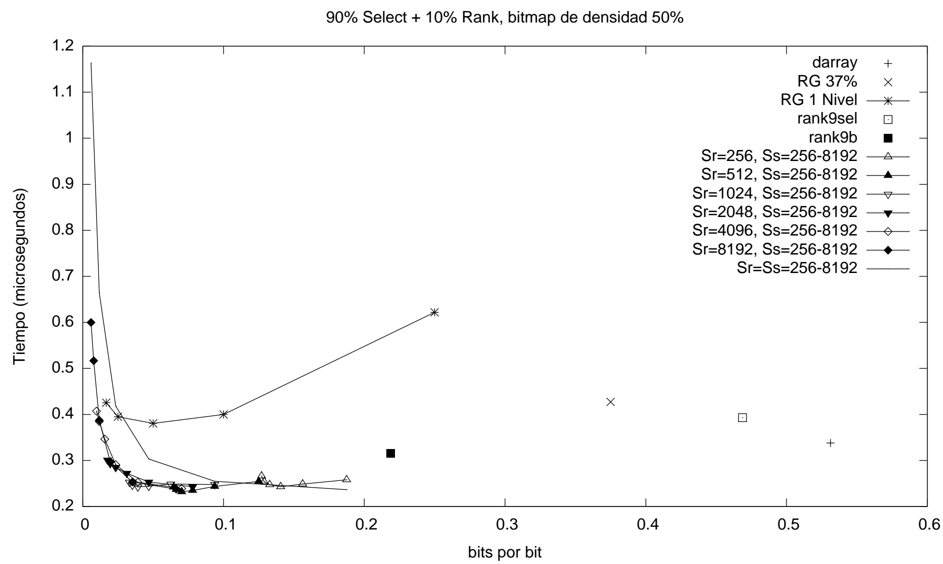


Figura 5.6: Resultado de experimentos de consultas *select*/*rank* mezcladas para la proporción *select*/*rank* de 90 %/10 %.

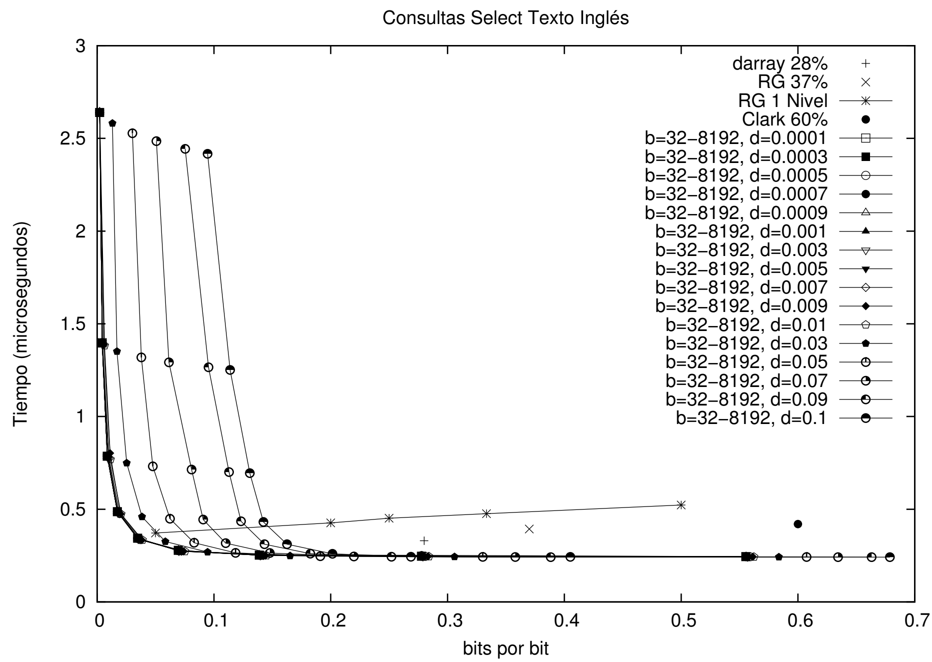


Figura 5.7: Relación tiempo y espacio por consulta *select* utilizando el texto Inglés.

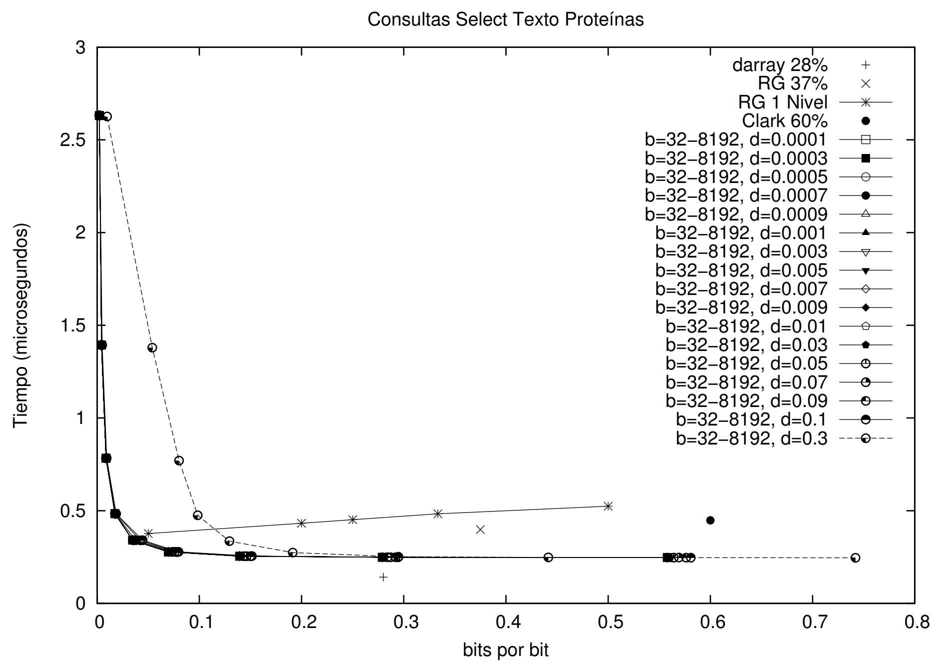


Figura 5.8: Relación tiempo y espacio por consulta *select* utilizando el texto Proteínas.

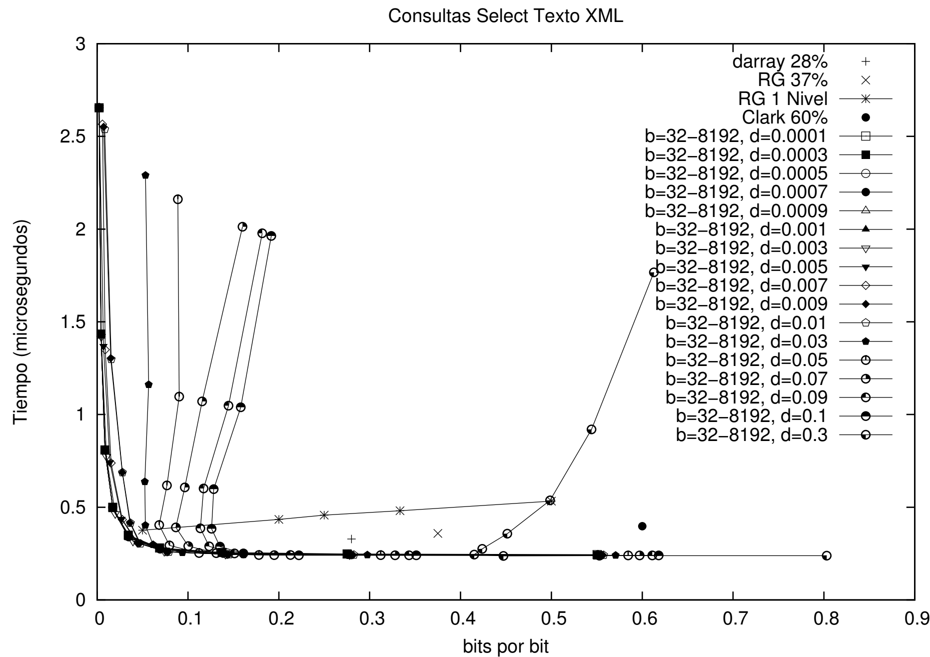


Figura 5.9: Relación tiempo y espacio por consulta *select* utilizando texto XML.

5.2. Estructura Select 1 Nivel

Con el objetivo de verificar el desempeño de esta estructura en tiempo y espacio, se realizaron los experimentos que se describen a continuación. Para estas pruebas se considera que un bloque es disperso si la cantidad de 1s del bloque dividido por el largo del bloque es menor a un parámetro d con los valores {0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, 0.003, 0.005, ..., 0.1, 0.3 }. Esta condición permite garantizar espacio extra utilizado para bloques dispersos, pero no así el tiempo.

Para realizar las pruebas, se utilizaron los bitmaps extraídos del wavelet tree de la transformada de Burrows-Wheeler de los textos de Inglés, Proteínas y XML, cada uno de 50 MB, disponibles en `http://pizzachili.dcc.uchile.cl`. Se escogieron estos bitmaps debido a que no poseen una distribución uniforme de 1s y 0s, sino que poseen largos runs de estos valores. Para cada prueba se crearon bloques de tamaño $b \in [2^5, 2^{13}]$. Como puntos de comparación se utilizan las implementaciones `darray`, `Clark`, `RG 37%`, y `RG 1 Nivel`.

Para el texto Inglés, Figura 5.7, se observa que a medida que el parámetro d aumenta, se encuentran más bloques dispersos, lo que lleva a un mayor espacio utilizado por la estructura, ya que para estos bloques se almacenan las respuestas para las consultas *select*. Con $d = 0.1$ los bloques dispersos equivalen al 1 % de todos los bloques creados, utilizando un espacio extra al bitmap de 16 % con $b = 512$, y logrando un tiempo de 0.31 microsegundos para cada consulta. Para $b = 256$ se obtiene un espacio extra de 20 %, logrando un tiempo de 0.25 microsegundos por consulta. Por otro lado, cuando $b = 32$ se utiliza alrededor de un 65 % de espacio extra con tiempos muy similares a los obtenidos con $b = 256$. En general según la gráfica conviene utilizar valores de $b \in \{256, 512, 1024, 2048\}$. Con valores muy pequeños para d, como 0.0001, no se encuentran bloques dispersos. Y para valores de d muy grandes

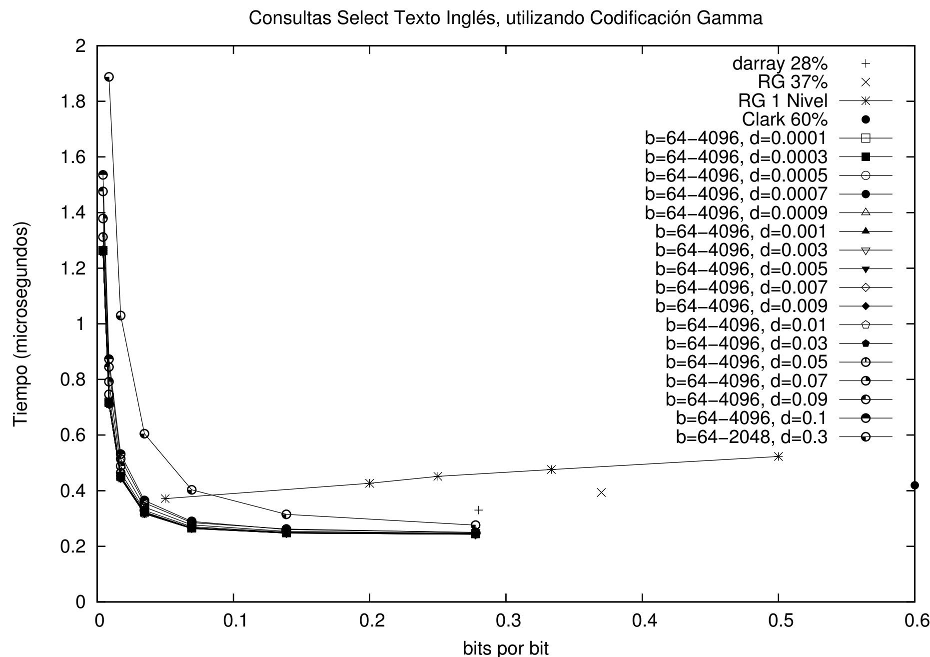


Figura 5.10: Resultados para bitmap Inglés, Select 1 Nivel utilizando codificación Gamma.

no se obtienen resultados competitivos.

Para el texto Proteínas, Figura 5.8, se observa que a medida que el valor de b aumenta se consideran menos bloques dispersos, por lo que se utiliza menos espacio extra al bitmap de entrada en contraste del aumento de tiempo. Con $d = 0.3$ y $b = 512$ se tienen 1.2 % de bloques dispersos y se utiliza 12 % extra de espacio, en contraste a cuando no considera bloques dispersos, en que utiliza un 3 % de espacio extra, obteniendo, en ambos casos, tiempos muy similares en las consultas. Es importante mencionar que para valores $d = \{0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, 0.003, 0.005, \ldots, 0.1\}$ el comportamiento es muy similar. Valores óptimos para b son 256 y 512, obteniendo buena relación tiempo y espacio.

Finalmente, para el texto XML, Figura 5.9, se observa que para todos los valores del parámetro d se encuentran bloques dispersos, y que estos bloques se encuentran en mayor cantidad cuando d aumenta, aumentando a la vez el espacio extra utilizado. Los valores para b interesantes son {256,512,1024}, obteniendo entre 0.25 y 0.28 microsegundos por consulta con $d = 0.0001$ y $d = 0.1$ respectivamente, utilizando entre un 6 % y 16 % de espacio extra, respectivamente.

5.3. Estructura Select 1 Nivel con Codificación

Como se mencionó en la Sección 4.2.3, se utilizaron dos tipos de codificación, *Gamma* y *Simple9*, para codificar las respuestas a bloques dispersos. Para los experimentos se realizaron 1.000.000 de consultas *select* aleatorias sobre los bitmap de Inglés, Proteínas y XML. Al igual que en la sección anterior, como puntos de comparación se utilizan la implementaciones `darray`, `Clark`, `RG 37%`, y `RG 1 Nivel`.

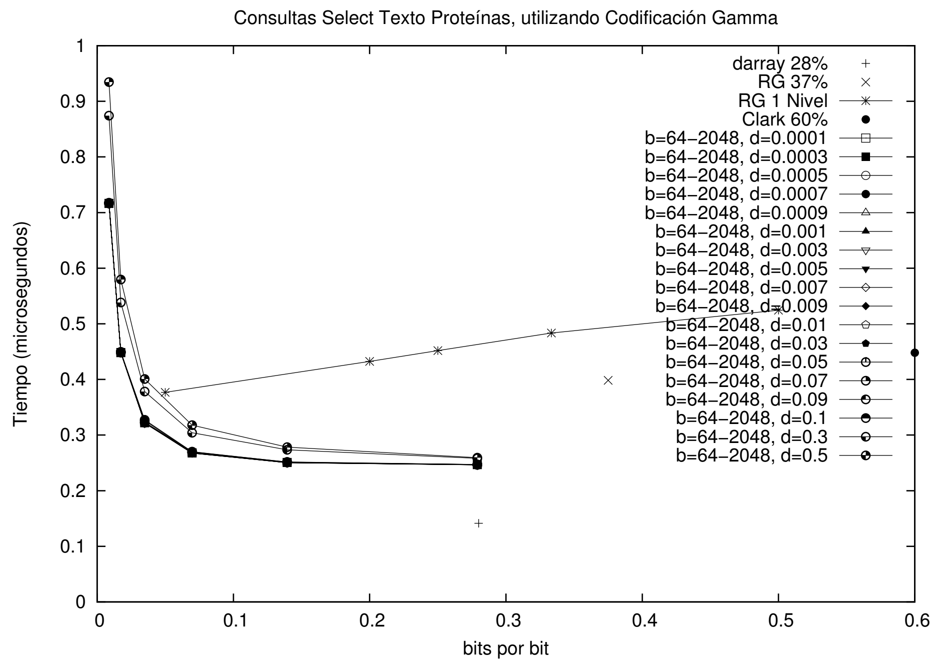


Figura 5.11: Resultados para bitmap Proteínas, Select 1 Nivel utilizando codificación Gamma.

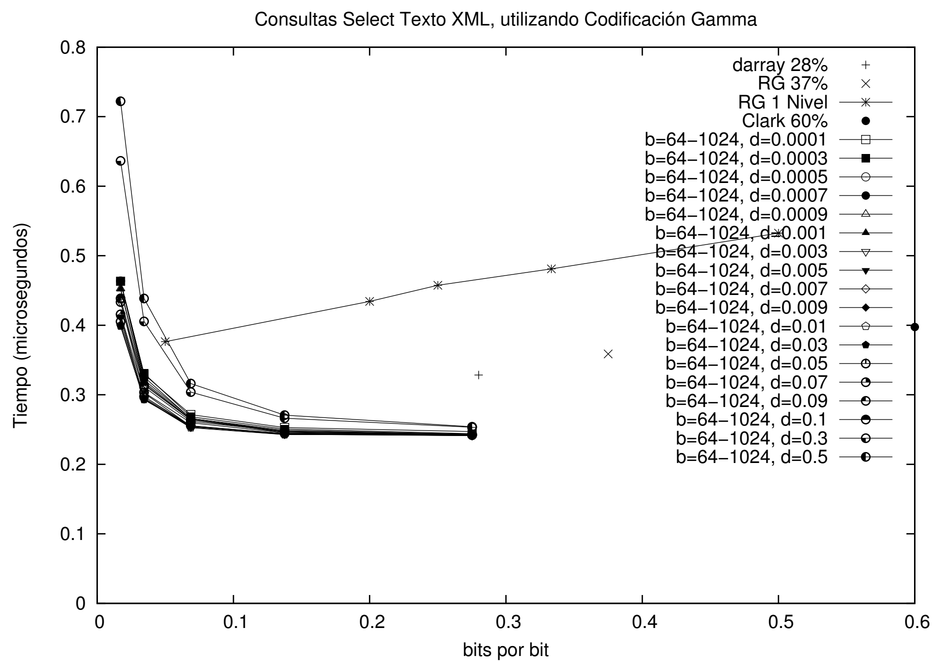


Figura 5.12: Resultados para bitmap XML, Select 1 Nivel utilizando codificación Gamma.

5.3.1. Utilizando Codificación Gamma

En esta variante se considera un bloque como disperso según dos criterios. Primero, de acuerdo a la relación entre la densidad del bloque con el parámetro d, descrito anteriormente, y segundo, el bloque es disperso solamente si el segmento del bitmap de entrada correspondiente al bloque tiene tamaño suficiente para almacenar las respuestas codificadas.

Para el bitmap del texto Inglés, se realizaron pruebas para valores de $b \in [2^5, 2^{13}]$, sin embargo en la Figura 5.10 sólo se muestran los valores en el rango $[2^6, 2^{12}]$ ya que para valores mayores a 4096 el tiempo no es competitivo y para $b = 32$ el espacio no es competitivo. Para los valores de d menores o iguales a 0.1 y $b = 512$ se utiliza un espacio extra alrededor de 0.034 % y el tiempo de consulta es de 0.3 microsegundos por consulta, considerando a lo más un 1 % de bloques dispersos. Con $d = 0.3$ y $b = 256$, se consideran 5 % de los bloques dispersos, logrando un espacio extra de 0.06 % y 0.4 microsegundos por consulta.

Para el bitmap de Proteínas, Figura 5.11, se muestran los resultados para $b \in [2^5, 2^{11}]$ (para valores de b mayores a 2048 el tiempo no es competitivo y para $b = 32$ utiliza un espacio extra no competitivo). Los puntos interesantes son aquellos donde $b = 256$ y $b = 512$, para todos los valores de d. En estos casos se obtiene entre 0.28 y 0.4 microsegundos por consulta con un espacio extra de 0.03 % y 0.07 % al bitmap de entrada, respectivamente. Notar que los resultados son casi similares para todos los valores menores o iguales a $d = 0.1$.

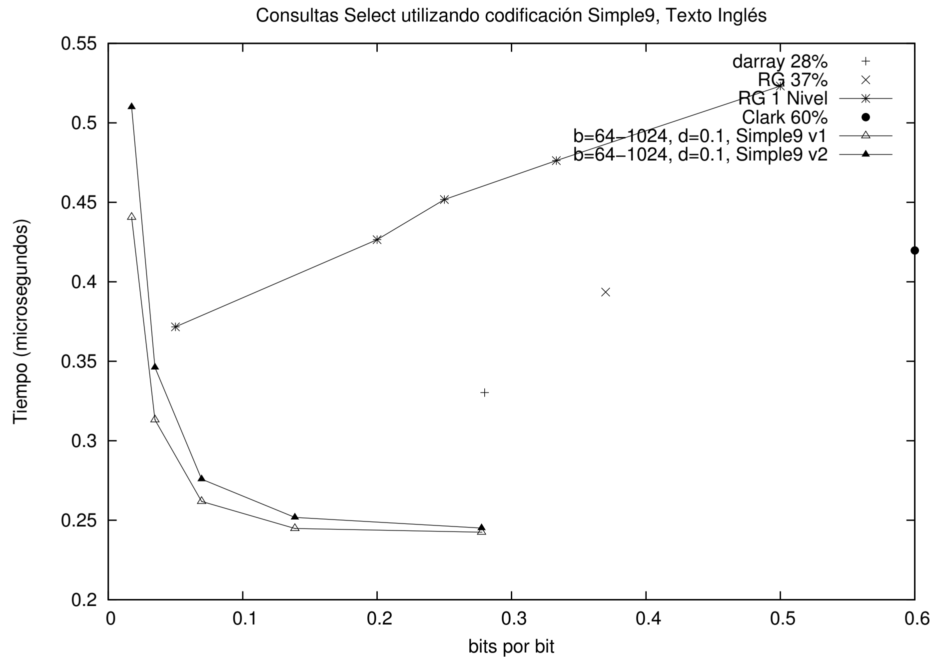


Figura 5.13: Resultados para bitmap Inglés, Select 1 Nivel utilizando codificación Simple9.

Para el bitmap de XML, Figura 5.12, con valores de $b \in [2^6, 2^9]$ y d menor a 0.1 se consigue tener un un tiempo de respuesta menor a 0.35 microsegundos por consulta, con un espacio que no supera el 28 % extra al bitmap de entrada. Por otro lado, con $d = 0.3 - 0.5$ y con b

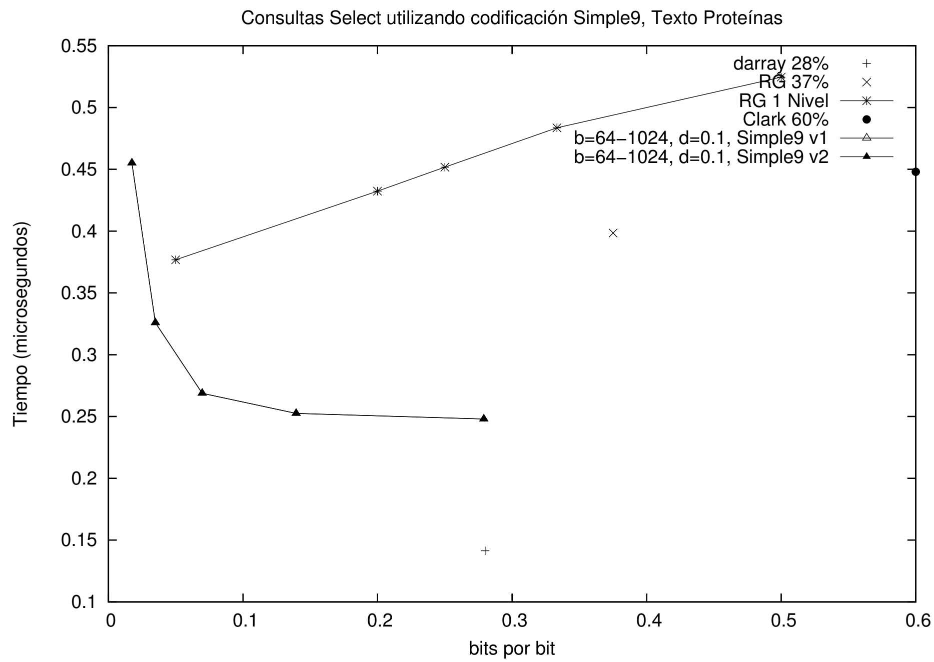


Figura 5.14: Resultados para bitmap Proteínas, Select 1 Nivel utilizando codificación Simple9.

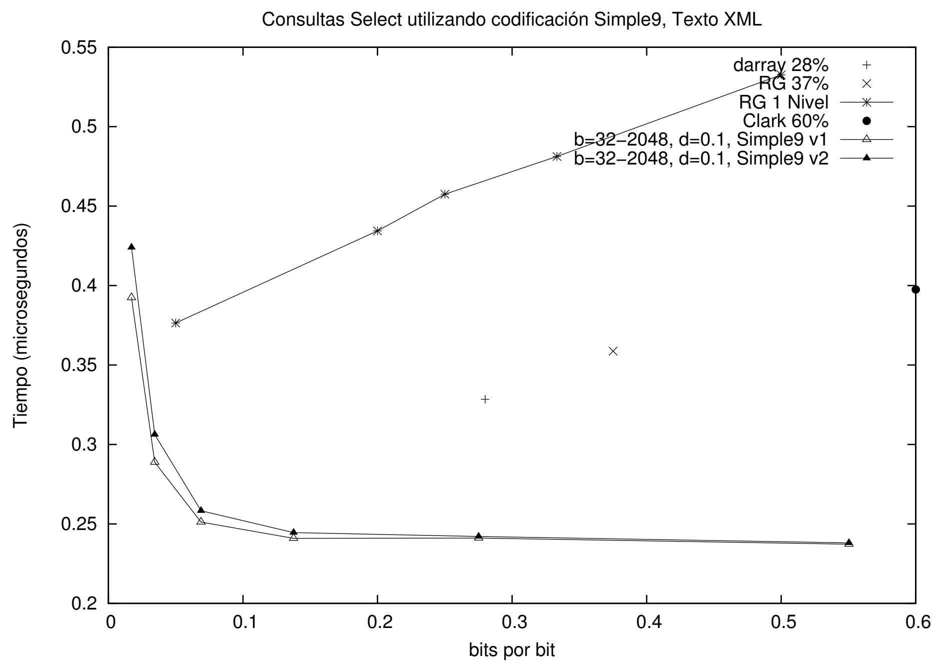


Figura 5.15: Resultados para bitmap XML, Select 1 Nivel utilizando codificación Simple9.

$= \{64, 128, 256\}$ se consigue un espacio con una diferencia muy pequeña a lo descrito para los otros valores de d, pero con mayor tiempo de respuesta.

5.3.2. Utilizando Codificación Simple9

Para esta codificación se crearon dos implementaciones: `Simple9 v1`, que almacena las posiciones absolutas de las respuestas *select*, y `Simple9 v2`, que guarda las diferencias de estas posiciones. Para estas implementaciones se ejecutaron 1.000.000 de consultas aleatorias, variando el parámetro d en los valores $\{0.0003, 0.0005, 0.0007, ..., 0.1\}$. Se usan los mismos criterios que para la codificación Gamma para determinar si un bloque es o no disperso.

Para el bitmap de Inglés, Figura 5.13, se presenta sólo el resultado para $d = 0.01$, ya que en cada versión para los otros valores del parámetro d se obtienen los mismos resultados, debido a la densidad y distribución de los 1s y 0s en el bitmap. Para $b \leq 512$ se obtiene un buen desempeño relación tiempo y espacio, considerando un 0.2 % de los bloques dispersos. Se observa que `Simple9 v1` tiene un mejor desempeño en tiempo de respuesta que `Simple9 v2`.

Para el bitmap de Proteínas, Figura 5.14, se presentan los resultados para $b \in [2^6, 2^{10}]$ y $d = 0.1$ (al igual que con el bitmap anterior, para los otros valores del parámetro d se obtienen los mismo resultados) obteniendo aproximadamente 0.7 % de bloques dispersos. El área competitiva para valores de b es cuando éste toma un valor menor a 1024 obteniendo entre 0.25 y 0.35 microsegundos por consulta. Notar que en ambas versiones se obtienen prácticamente los mismos resultados.

Para el bitmap XML, Figura 5.15, se presentan los resultados obtenidos para $b \in [2^5, 2^{11}]$. Al igual que en los bitmap anteriores se presentan los resultados con $d = 0.1$. El área interesante es cuando $b \in \{64, 128, 256, 512\}$ obteniendo entre 0.31 y 0.24 microsegundos por consulta, y entre 0.04 % y 0.27 % de espacio extra, respectivamente. Al igual que en el bitmap de Inglés, `Simple9 v1` tiene mejor desempeño.

5.3.3. Comparación para Consultas *Select* entre Muestreo Combinado y Select 1 Nivel

A continuación se compara la estructura de muestreo combinado con todas las variantes de Select 1 Nivel, y con las implementaciones existentes usadas en la sección anterior. Las Figuras 5.16 a la 5.18 muestran los resultados para los bitmaps Inglés, Proteínas y XML. En todos los casos analizados se observa que la estructura de muestreo combinado ofrece el mejor tradeoff en espacio y tiempo. Sin embargo, Select 1 Nivel permite reducir aún más el espacio, con una correspondiente penalización en tiempo de respuesta. En conclusión, utilizando estas estructuras es posible escoger el tradeoff de acuerdo a los requerimientos en espacio y tiempo de consulta, obteniendo en todos los casos menos espacio que en las soluciones actuales.

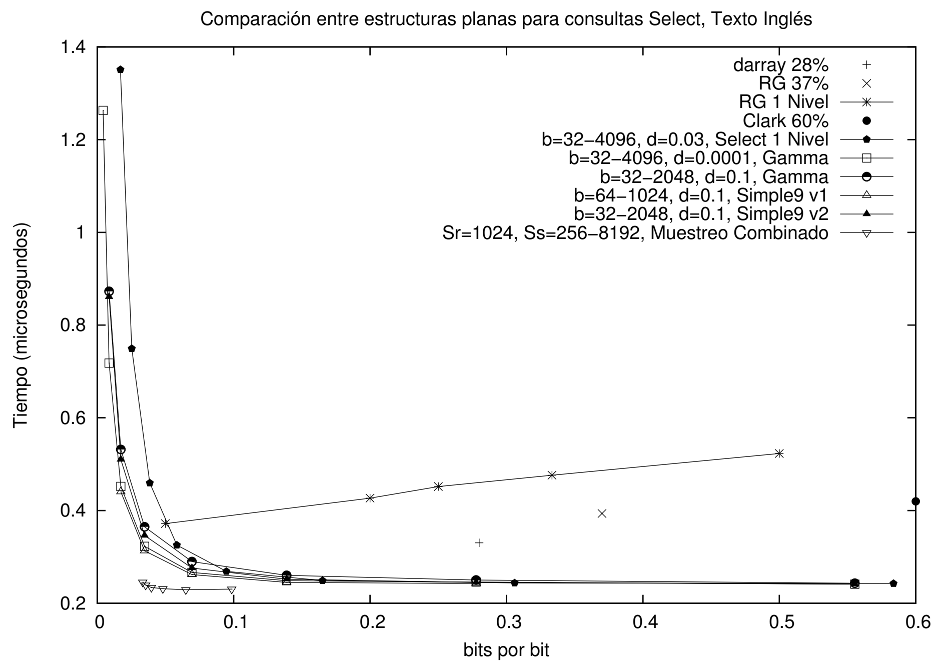


Figura 5.16: Resultados comparación consultas *select* para bitmap Inglés.

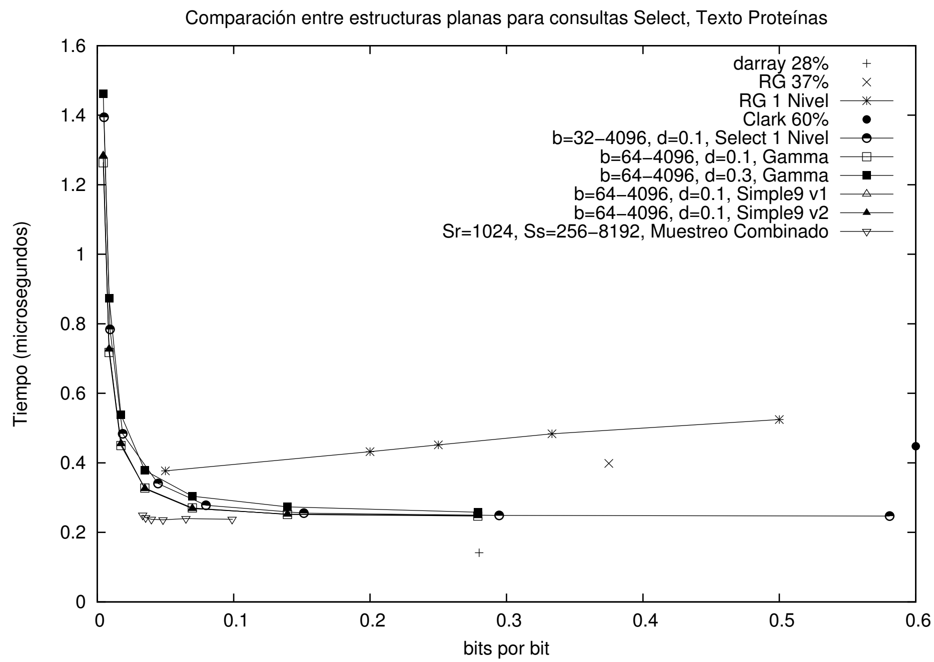


Figura 5.17: Resultados comparación consultas *select* para bitmap Proteínas.

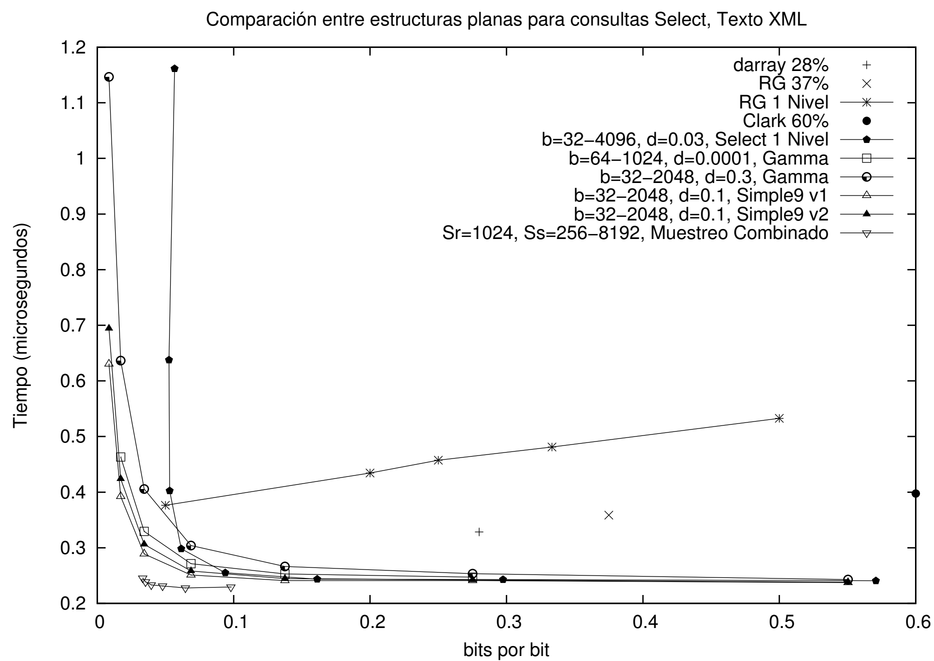


Figura 5.18: Resultados comparación consultas *select* para bitmap XML.

bloque	*superbloque*	*bloque*	*superbloque*	*bloque*	*superbloque*
15	480	31	992	63	2016
	960		1984		4032
	1920		3968		8064

Tabla 5.1: Combinación de largos de bloques y superbloques.

5.4. Estructuras para Bitmaps Dispersos

Para esta estructura de datos se ejecutaron cuatro experimentos. En el primero se realizan consultas independientes para comparar el rendimiento de las operaciones *rank* y *select* con la implementación `RRR` descrita en la Sección 3.2.2, de la estructura original de Raman *et al.* [9] descrita en la Sección 3.2.1, y la implementación `sarray` para bloques dispersos de Okanohara y Sadakane [25] descrita en la Sección 3.2.3.

En el segundo experimento se crea una variante de la estructura que incorpora un muestreo regular en 1s, solamente para *select*, incorporando las ideas de las Secciones 4.1 y 4.2, y se compara con la estructura para bitmaps dispersos original (sin muestreo para 1s), y con la implementación `RRR`.

En el tercer experimento, se compara la implementación original de `RRR` (Sección 3.2.2), con todas las variantes realizadas a esta estructura (Sección 4.3).

Finalmente, en el cuarto experimento, se construye el FM-index [11] utilizando la estructura de datos para bitmaps dispersos, ejecutando posteriormente consultas de tipo *count*, que cuentan las ocurrencia de un patrón en el texto indexado, y se compara el desempeño con el FM-index original.

5.4.1. Consultas Independientes

Para el experimento se combina el largo de bloques y superbloques como se muestra en el Cuadro 5.1. Para cada una de estas combinaciones se generaron bitmaps aleatorios, con densidad 5 %, 10 % y 20 % que tienen una entropía de orden cero de 0.286, 0.469 y 0.772 respectivamente. Luego, considerando los parámetros descritos, de forma independiente se ejecutaron 1.000.000 de consultas para *rank* y *select*. El mismo experimento se realizó con las otras implementaciones (notar que `RRR` puede utilizar sólo bloques de tamaño 15).

En las Figuras 5.19, 5.20 y 5.21, se presentan los resultados obtenidos en el experimento, donde se observa que al incrementar el tamaño del bloque de 15 a 63 se reduce ampliamente el overhead en espacio de la representación del bitmap de entrada, alrededor de un 40 %-50 % de su valor.

Los resultados de las Figuras 5.19, 5.20 y 5.21 se describen a continuación de acuerdo al tipo de consulta.

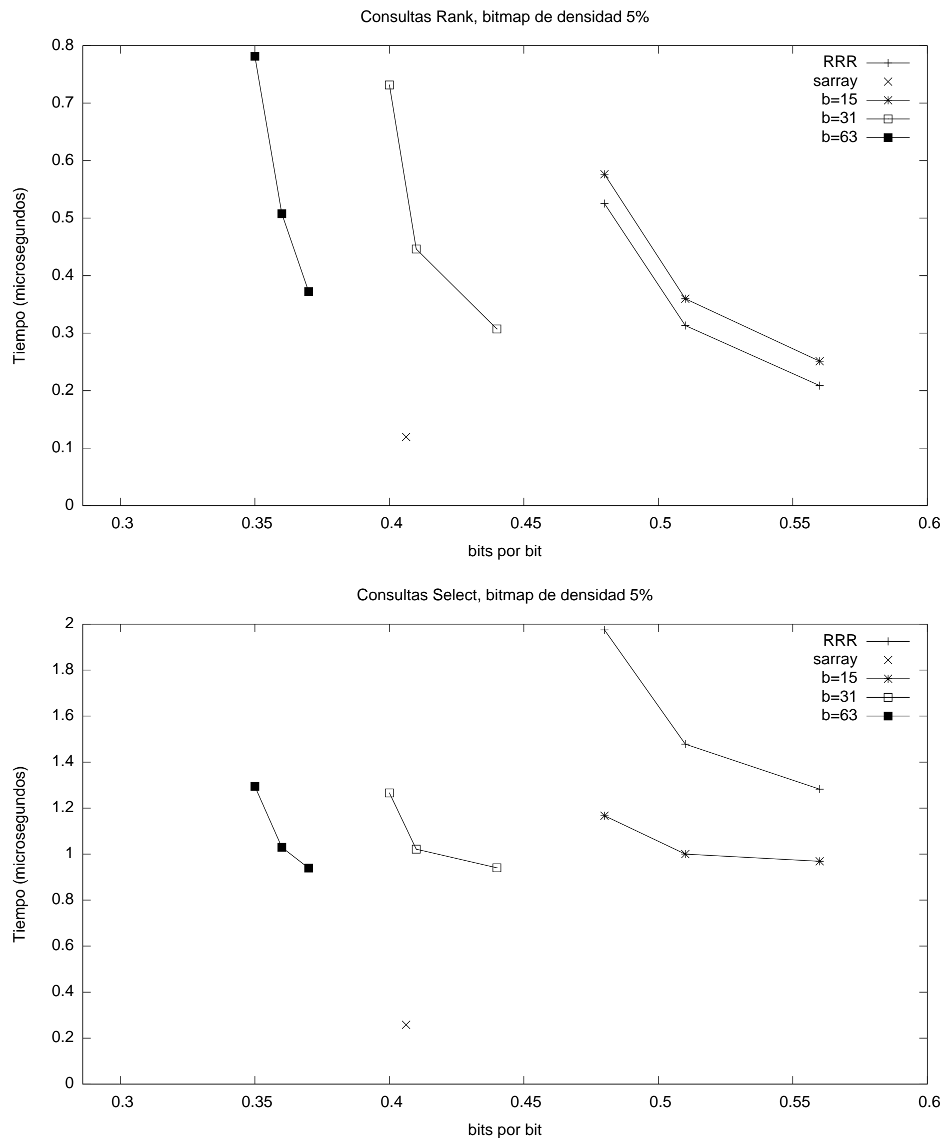


Figura 5.19: Resultados para bitmaps comprimidos con densidad 5 %. La coordenada x parte desde la entropía de orden cero del bitmap, con lo que la redundancia se puede apreciar con mayor claridad.

Para la operación *select* se observa que la nueva implementación es un 30 %-50 % más rápida que `RRR` incluso para el mismo tamaño de bloque $b = 15$. Los nuevos tiempos para *select* son casi insensibles al tamaño del bloque. Esto muestra que el tiempo para reconstruir un bloque es despreciable comparado con el de la búsqueda binaria. Por otro lado, superbloques más pequeños hacen la operación más rápida, por lo que se recomienda un superbloque de 32 o 64 bloques.

Por otro lado, para la operación *rank* `RRR` es ligeramente más rápido que la implementación de la nueva estructura. Se observa que utilizando bloques más largos se requiere más tiempo para responder, sin embargo sigue siendo un tiempo aceptable con respecto a los bloques de largo 15. En conjunto, el precio es bajo comparado con la gran ganancia en espacio.

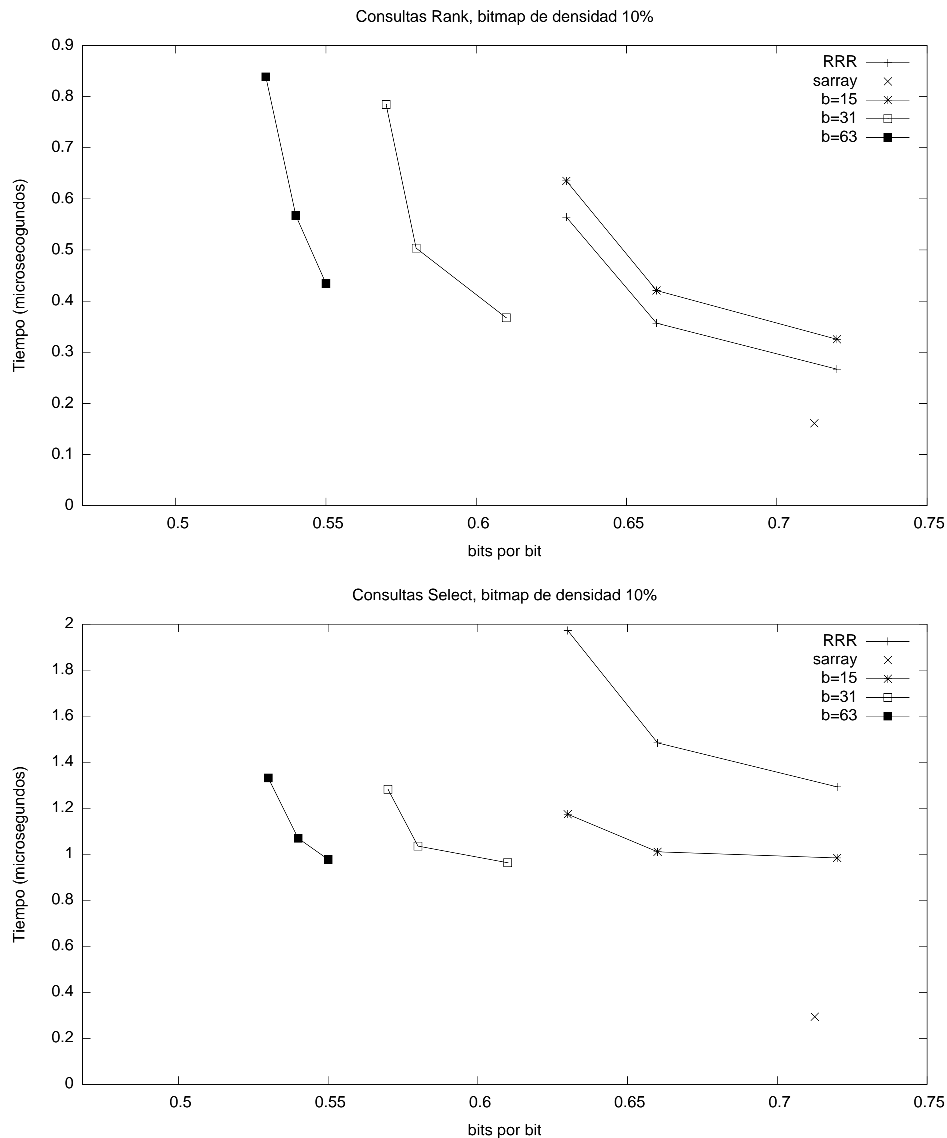


Figura 5.20: Resultados para bitmaps comprimidos con densidad 10 %. La coordenada x parte desde la entropía de orden cero del bitmap, con lo que la redundancia se puede apreciar con mayor claridad.

Finalmente, `sarray` es significativamente más rápido que todas las otras implementaciones, especialmente para *select*. Sin embargo su espacio es competitivo sólo para densidades muy bajas.

En resumen, usando un tamaño de bloque $b = 63$ y superbloques de 32 o 64 bloques, se observa que la estructura de datos propuesta en el presente trabajo calcula *rank* en un tiempo de alrededor de medio microsegundo, *select* en cerca de un microsegundo, y requiere un costo en espacio superior, pero muy cercano, a la entropía. Esta es una alternativa muy conveniente para bitmaps dispersos (con densidad hasta 20 %, ya que luego la entropía se acerca demasiado a 1.0). Para bitmaps muy dispersos (densidades bastante menores al 5 %), la técnica `sarray` se vuelve la mejor alternativa. Sin embargo, es importante destacar que estos resultados mejorarían inmediatamente en procesadores de 128 bits, pues se podría trabajar

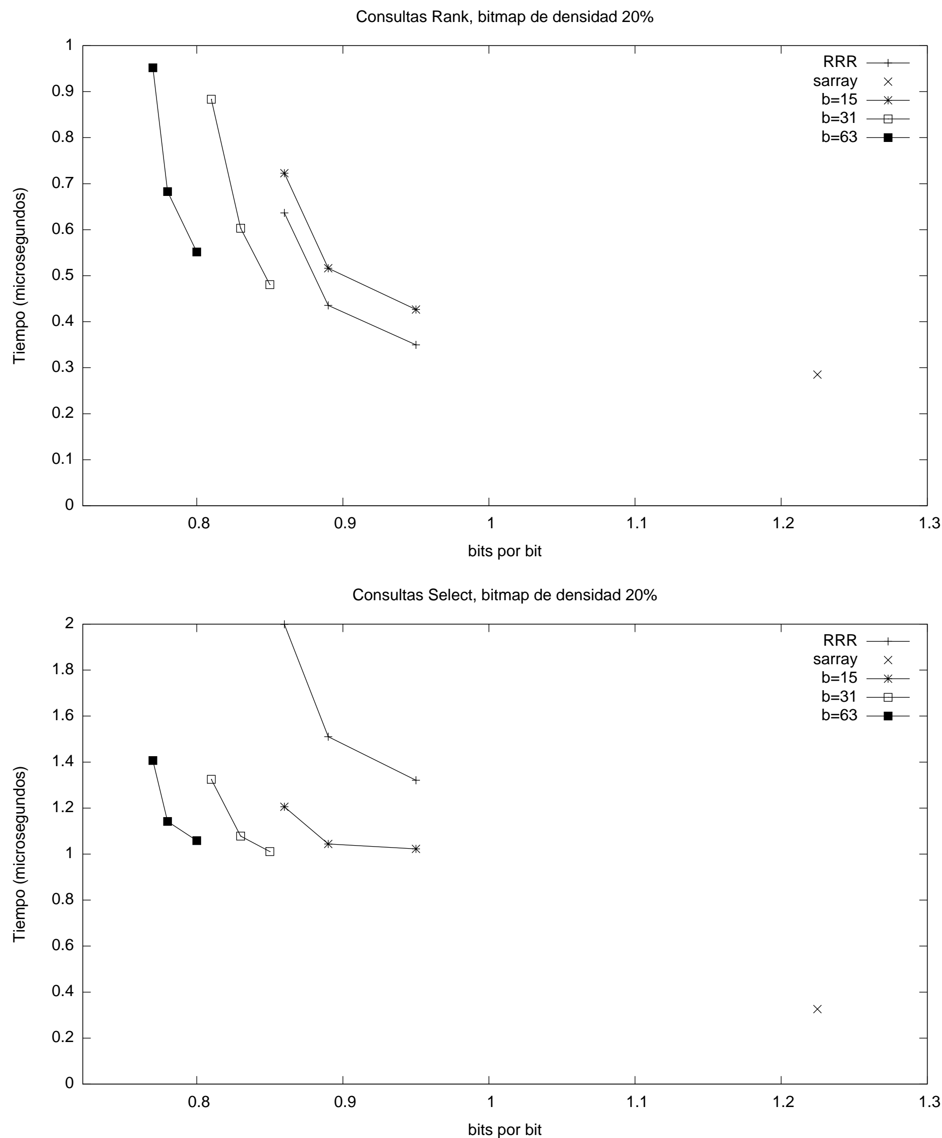


Figura 5.21: Resultados para bitmaps comprimidos con densidad 20 %. La coordenada x parte desde la entropía de orden cero del bitmap, con lo que la redundancia se puede apreciar con mayor claridad.

con bloques de tamaño $t = 128$ directamente en una palabra de memoria (Sección 4.3).

5.4.2. Resultados al Combinar con el Esquema de Particionamiento Regular en 1s para *Select*

Con el objetivo de optimizar aún más las consultas *select*, en esta sección se presentan dos variantes de la estructura de datos para bitmaps dispersos, que incorporan un particionamiento regular en 1s, determinado por el parámetro S_s. Las implementaciones se diferencian en que la primera no almacena respuestas *select* para bloques dispersos y la segunda sí las almacena. Al igual que la estructura de Select 1 Nivel (Sección 4.2), un bloque es considerado disperso si la cantidad de 1s del bloque dividido por el largo del bloque es menor a un parámetro

$d \in [0, 1]$. En otro caso el bloque se considera denso.

La primera variante está diseñada para bitmaps dispersos en general, sin considerar la distribución de los 1s y 0s. En cambio, la segunda variante está diseñada específicamente para los casos en que existen runs de 0s en el bitmap. El segundo enfoque utiliza un mayor espacio a los ya presentados al inicio de la Sección 5.4.1, pero sin embargo tiene mejor tiempo de respuesta.

Resultados Sin Almacenar Respuestas

Se realizaron dos experimentos con la variante que no almacena respuestas *select* de los bloques dispersos. El primero consiste en realizar consultas independientes de tipo *select* y el segundo en realizar consultas mezcladas *rank*/*select*. A continuación se muestran sólo los resultados del primer experimento, pues los resultados del segundo no muestran grandes diferencias con los primeros.

Para el experimento de consultas independientes se ejecutaron 1.000.000 de consultas sobre bitmaps con densidad 2 %, 3 %, 5 %, 10 % y 20 %. Tanto las consultas como los bitmaps se generaron aleatoriamente. Se utiliza el parámetro $S_s = [2^8, 2^{14}]$ y los valores de bloque y superbloque descritos en el Cuadro 5.1. Observe que en vez de mostrar las curvas para cada combinación b, sb, S_s, se presentan las curvas con los puntos dominantes[2] de S_s para cada combinación b, sb. Estos valores de S_s pueden coincidir para cada combinación, como por ejemplo en la Figura 5.22 para $b = 15$ y $b = 63$.

De acuerdo a los gráficos presentados en la Figuras 5.22 hasta la Figura 5.26, se observa que para densidades mayores o iguales a 3 % del bitmap de entrada y con $b = \{31, 63\}$ se obtiene un mejor desempeño en espacio con respecto a los resultados de `sarray`, y que en todos los casos el desempeño sigue siendo mejor que el de `RRR`. Observe que cuando $b = 15$, el largo de los bloques es igual que en `RRR`. Además, esta variante ocupa una pequeña cantidad adicional de espacio que la estructura original, pero mejora considerablemente el tiempo de respuesta. Finalmente, es importante mencionar que todas las curvas presentadas en las figuras muestran los puntos dominantes (considerando la combinación superbloque y S_s), lo que permite comparar con mayor claridad los resultados obtenidos.

Resultados que almacenan respuestas para bloques dispersos

Como fue descrito en la Sección 4.3.4, en caso de bitmaps muy dispersos o con runs de ceros muy largos, se recorren innecesariamente los muestreos para *rank* asociados a estas subsecuencias. Este experimento busca medir el impacto de almacenar las respuestas para estos bloques, considerando el costo extra en espacio contra la mejora en el tiempo de consulta.

Para el experimento se utilizó como bitmap de entrada el primer nivel del wavelet tree de la transformada de Burrows-Wheeler del texto Inglés. Se eligió este bitmap por su distribución de 1s y densidad. Se realizaron experimentos con valores del parámetro $d = \{0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, \ldots, 0.09, 0.1, 0.3\}$, y del parámetro $S_s = [2^5, 2^{14}]$. Los valores

[2] Un punto es dominante si no existe otro que use menos espacio y menos tiempo que él.

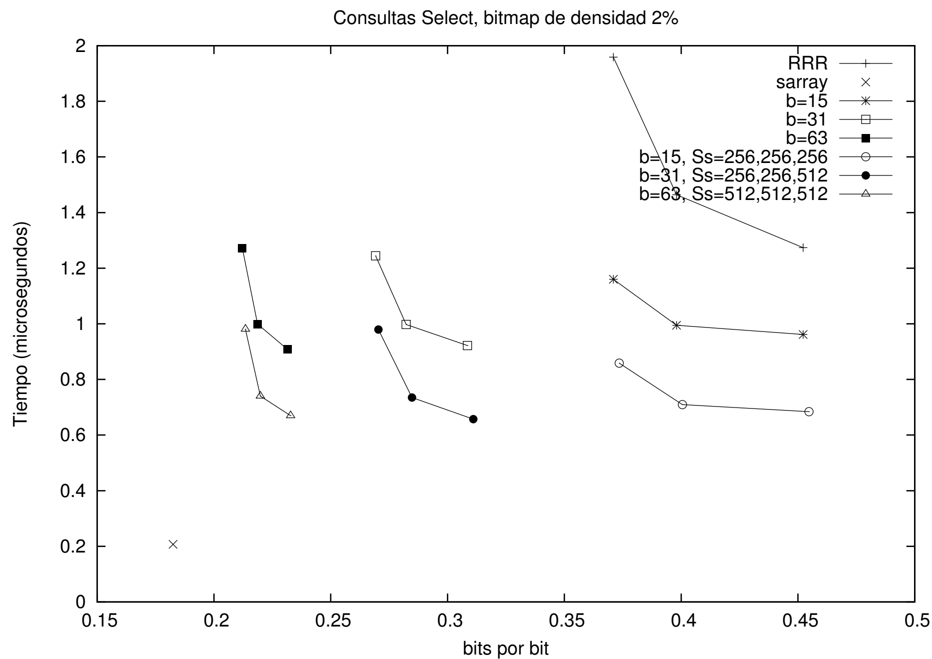


Figura 5.22: Resultados para consultas *select* para un bitmap de densidad 2 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb.

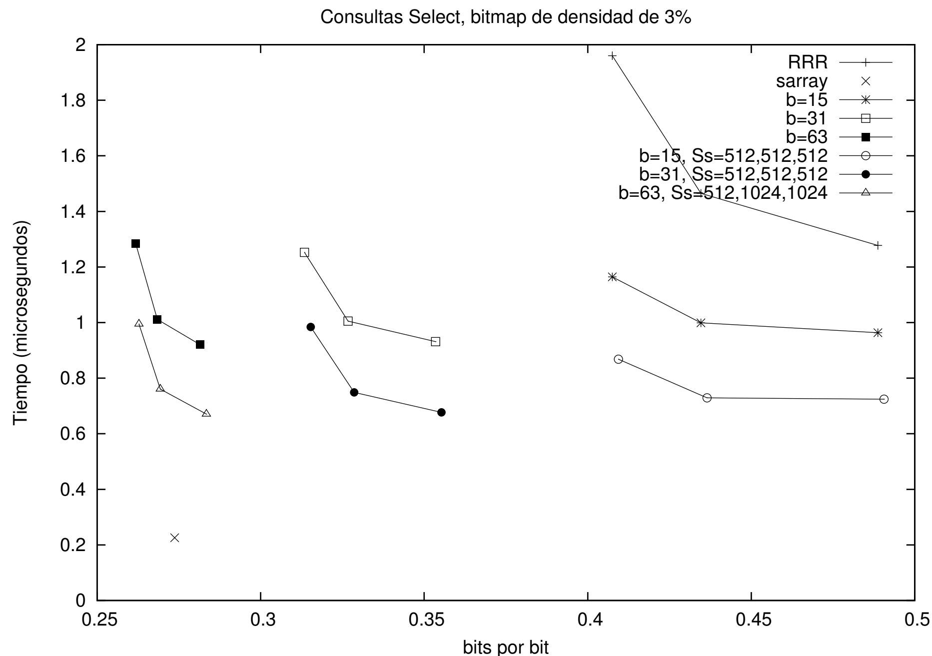


Figura 5.23: Resultados para consultas *select* para un bitmap de densidad 3 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb.

de bloques y superbloques son los descritos en el Cuadro 5.1. Se presentan solamente los resultados para el muestreo $S_s = [2^7, 2^{14}]$. Los valores $S_s = \{32{,}64\}$ no se presentan ya que

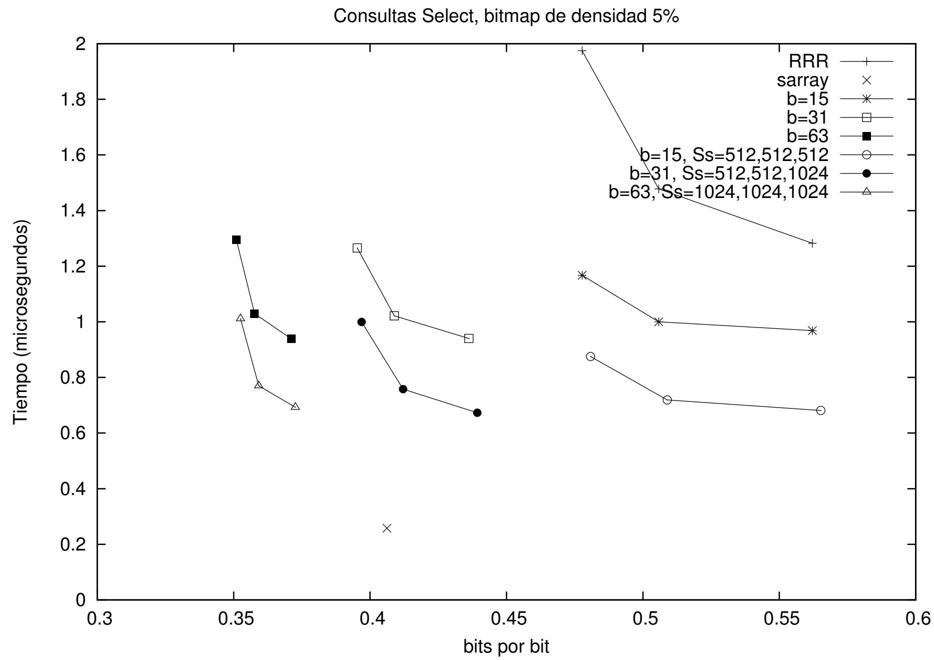


Figura 5.24: Resultados para consultas *select* para un bitmap de densidad 5 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb.

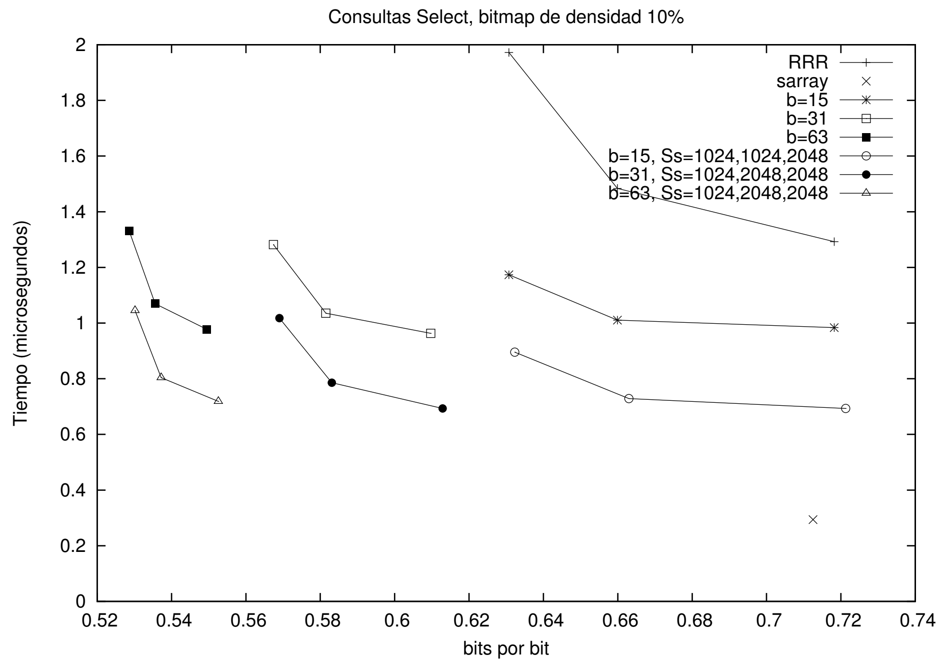


Figura 5.25: Resultados para consultas *select* para un bitmap de densidad 10 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb.

utilizan demasiado espacio y los valores S_s entre 256 y 4096 (sin considerar los extremos) se omiten ya que sus resultados en tiempo y espacio se encuentran entre los obtenidos con

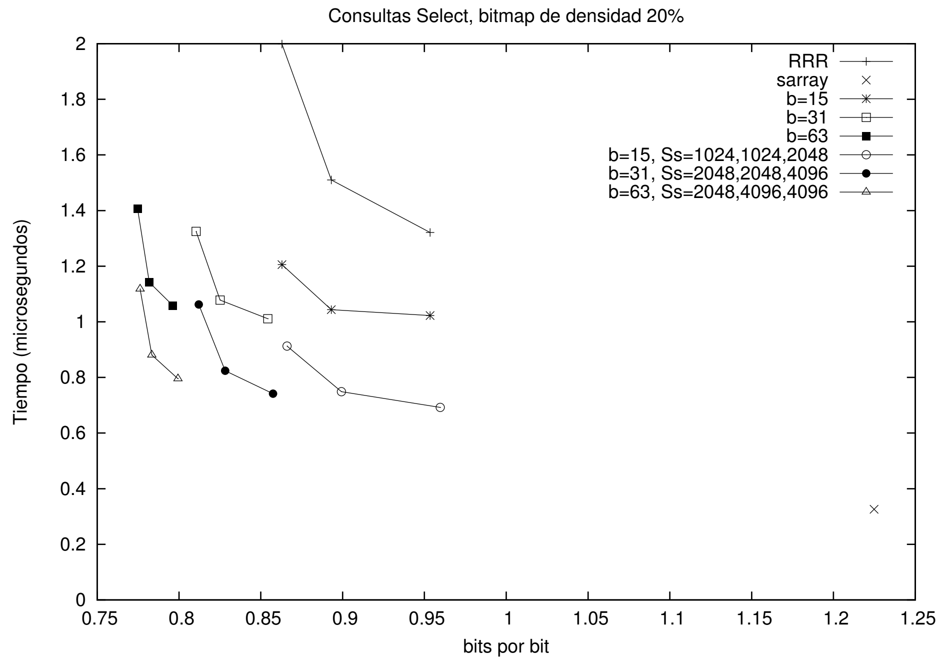


Figura 5.26: Resultados para consultas *select* para un bitmap de densidad 20 %, utilizando sampling de 1s. En cada curva se usan los valores dominantes de S_s para cada combinación b y sb.

los valores que se presentan. De acuerdo a los experimentos, se observa que la estructura de datos comienza a almacenar respuestas para bloques dispersos desde $d = 0.003$, los que van aumentando a medida que aumenta el valor de d.

En la Figura 5.27 se presentan los resultados obtenidos, donde se observa que se obtienen mejores resultados que la implementación de `RRR` en tiempo y espacio para los valores de S_s mayores a 128. Un punto importante de comparación es con $b = 63$, donde se observa que utilizando $S_s = \{4096, 8192, 16384\}$ se utiliza un poco más de espacio pero se consigue obtener mejor tiempo de respuesta que la implementación original (sin muestreo de 1s) que utiliza $b = 63$.

En la Figura 5.28 se presentan los resultados donde se consideran más bloques dispersos para los cuales se almacenan sus respuestas. Desde $b = 15$ y $S_s = 16384$ se empiezan a tener resultados competitivos en tiempo y espacio con respecto a `RRR`. Se observa que con $b = 63$ y $S_s = \{8192, 16384\}$ se utiliza más espacio (alrededor de 0.6 bits por bit en ambos casos) pero se consigue tener mejor tiempo que la implementación que sólo utiliza $b = 63$.

5.4.3. Comparación Variantes Estructura de Raman *et. al* para Consultas *Select*

A continuación se presenta una comparación entre la implementación de la solución original de la estructura de Raman *et. al*, `RRR`, por Claude y Navarro [9], la implementación, `EBD`, de la estructura para bitmaps dispersos (Sección 4.3) y las variantes de esta última que utilizan

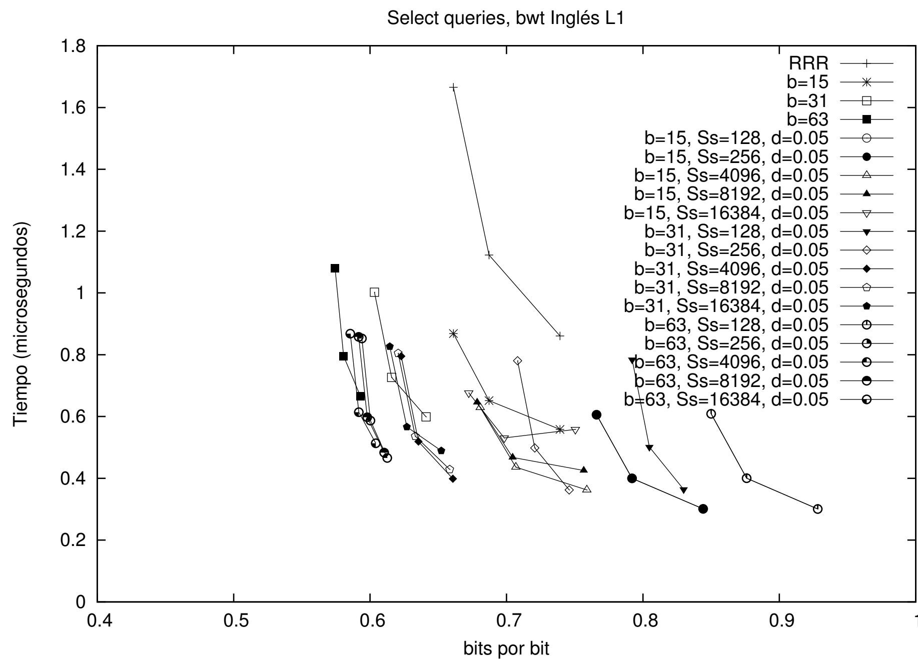


Figura 5.27: Resultados para consultas *select*, para el primer nivel del wavelet tree de la transformada de Burrows-Wheeler del texto Inglés. Se utilizan diferentes valores de bloques, superbloques y S_s, y $d = 0.05$.

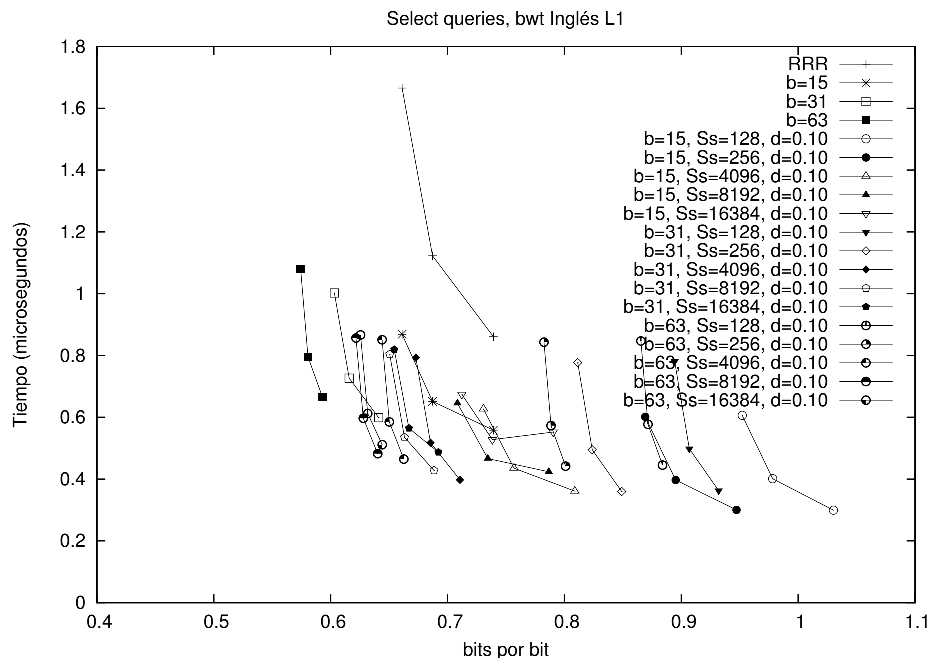


Figura 5.28: Resultados para consultas *select*, para el primer nivel del wavelet tree de la transformada de Burrows-Wheeler del texto Inglés. Se utilizan diferentes valores de bloques, superbloques y S_s, y $d = 0.1$.

el muestreo regular en 1s (Sección 4.3.4, `EBD1` que no almacena respuestas y `EBD2` que sí almacena respuestas para bloques dispersos). Para este experimento se usa el wavelet tree de

la transformada de Burrows-Wheeler de los textos Inglés, Proteínas y XML.

Como se observa en la Figura 5.29, para el texto Inglés se obtiene un mejor espacio que en `RRR`, a excepción de `EBD2` cuando $b \in \{15, 31\}$ y $d = 0.1$. En todos los casos (excepto en dos configuraciones) se supera el mejor tiempo de `RRR`. La máxima disminución de espacio es alrededor de 10 %, en la estructura `EBD1` con $b = 63$ y $S_s = 8192$.

Para el texto Proteínas, Figura 5.30, en todos los casos se tiene mejor desempeño que en `RRR`, tanto en tiempo como en espacio, obteniendo una disminución máxima en el espacio utilizado de alrededor de 4 %, en la estructura `EBD1` con $b = 63$ y $S_s = 8192$, al igual que en el texto Inglés.

Para el texto XML, Figura 5.31, se observa una mejora en tiempo y espacio de las nuevas implementaciones con respecto a la original `RRR`, logrando una disminución máxima en el espacio de 18 % cuando $b = 63$ con la estructura `EBD1`. Al igual que en el texto Inglés, sólo dos casos no superan el tiempo de `RRR`. Con `EBD1`, cuando $b = 63$ y $sb = 2016$ se consigue el mejor tiempo de respuesta de 0,42 microsegundos.

Además, se observa que para los tres textos existen diversas configuraciones de las estructuras para bitmaps dispersos, las que ofrecen un variado conjunto de tradeoffs espacio tiempo. En conclusión, la estructura de bitmaps dispersos es claramente superior a la implementación de la solución original para consultas *select*.

Finalmente, se observa que al incorporar la idea de la estructura de muestreo combinado a la estructura para bitmaps dispersos se obtiene un excelente desempeño en tiempo y espacio. Sin embargo, en este caso almacenar respuestas para los bloques dispersos no impacta notoriamente en el tiempo de consulta, pero incurre en un considerable costo en espacio. Esto muestra que no vale la pena explorar la variante que codifica las respuestas *select* codificadas (Sección 4.2.3) en la variante para bitmaps dispersos, ya que sería aún mas lenta.

5.4.4. Integrando la Estructura al FM-index

Con el objetivo de ilustrar las muchas aplicaciones de estructuras de datos para *rank*/*select* se utiliza un ejemplo basado en el FM-index [11]. Para ello, se utiliza un FM-index con la nueva estructura para bitmaps dispersos descrita en la Sección 4.3. El FM-index representa una colección de texto de forma comprimida y ofrece capacidad de búsqueda en ella, como se explicó en la Sección 2.5.3. Su implementación más práctica y compacta [9] utiliza un wavelet tree con forma de Huffman que almacena a lo más $n(H_0(T) + 1)$ bits, y esos bits se representan de forma comprimida. El FM-index puede contar el número de ocurrencias de un patrón P en T con un número de operaciones *rank* sobre los bitmaps. La implementación utilizada de FM-index es la que está disponible en `http://libcds.recoded.cl`. Esta implementación se compara con una donde cambiamos los bitmaps comprimidos por la nueva implementación presentada en la Sección 4.3. Para las pruebas se indexó el texto en Inglés de 50MB de `http://pizzachili.dcc.uchile.cl`, y luego se buscaron 50.000 patrones de largo 20 extraídos al azar desde el texto.

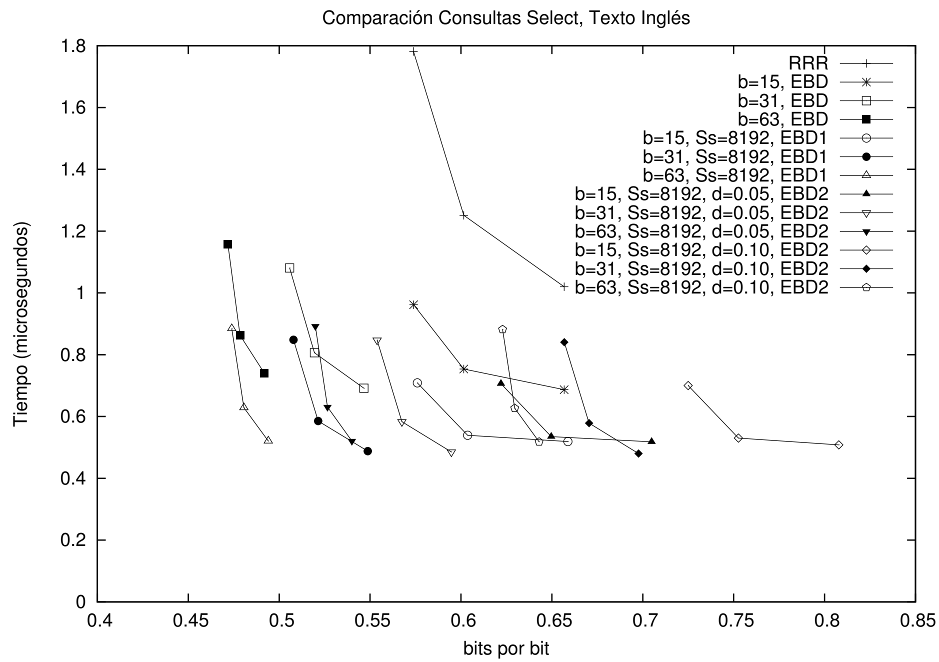


Figura 5.29: Resultados comparación variantes estructura de Raman *et. al* para consultas *select* en el texto Inglés.

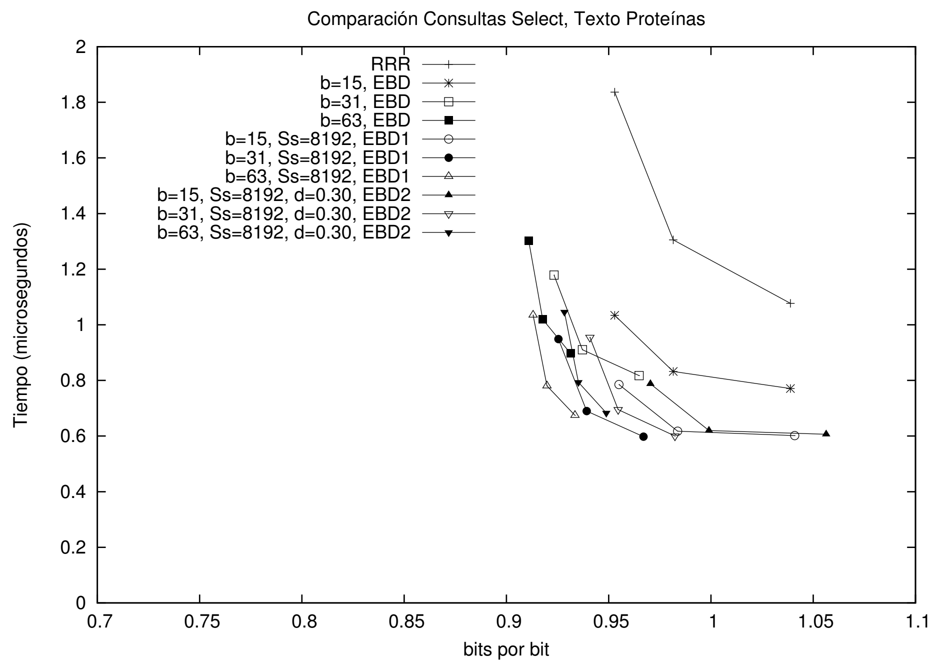


Figura 5.30: Resultados comparación variantes estructura de Raman *et. al* para consultas *select* en el texto Proteínas.

Como se visualiza en la Figura 5.32, el nuevo índice con bloques $b = 63$ y superbloques de 32 bloques es ligeramente más lento que el original, pero aún así cuenta en 90 microsegundos usando 0.28 bits por bit, donde el original necesita 0.335 bits por bit para alcanzar el mismo tiempo. Es decir, la nueva implementación reduce el espacio alrededor de 16 %. Con esto se

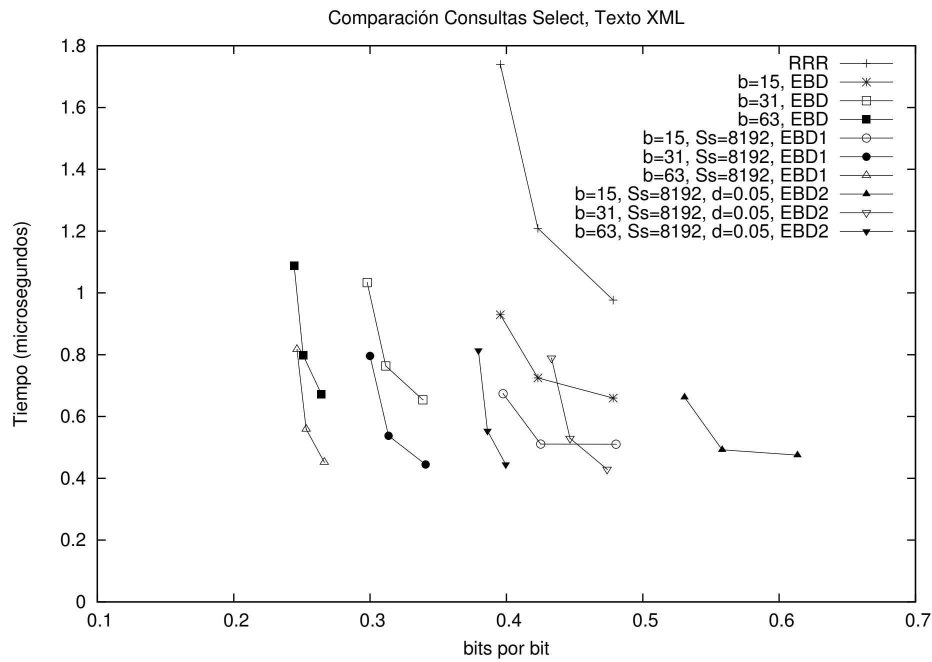


Figura 5.31: Resultados comparación variantes estructura de Raman *et. al* para consultas *select* en el texto XML.

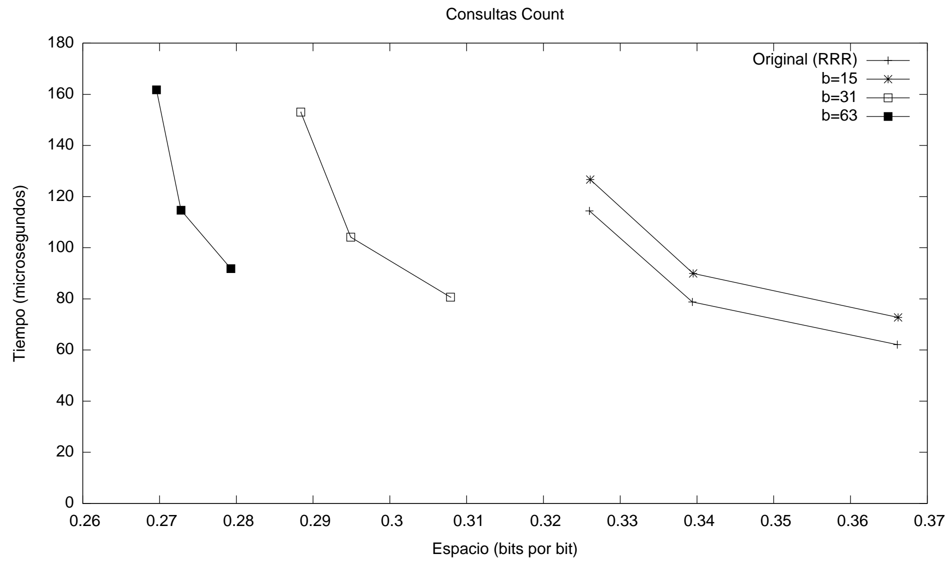


Figura 5.32: Consultas *Count* en el FM-index sobre texto Inglés de 50 MB.

ha alcanzado el menor espacio que se haya reportado para un FM-index para este texto (los resultados en otros textos fueron similares).

Capítulo 6

Conclusión y Trabajos Futuros

Este trabajo contribuye a mejorar el desempeño de las consultas *rank* y *select* sobre secuencias binarias, mejorando notablemente los resultados de las implementaciones existentes. En el caso particular de consultas *select*, nuestros resultados permiten su uso práctico en estructuras de datos, a diferencia de resultados anteriores de la literatura, donde el desempeño de *select* en tiempo y/o espacio desincentiva su uso. Asimismo, nuestras estructuras de datos reducen notablemente la redundancia sobre la entropía de los bitmaps, tanto si son compresibles como si no.

Hemos desarrollado tres estructuras de datos: la primera es la Estructura de Muestreo Combinado, donde se desarrolla la idea de complementar los muestreos para *rank* y *select* para la respuesta de consultas, en vez de usarlos de manera independiente como en otras estructuras. Esta estructura de datos alcanza sólo 3 % de espacio extra y resuelve ambas operaciones en alrededor de 0.2 microsegundos, mucho más eficientemente que las implementaciones actuales.

La segunda estructura de datos, Select 1 Nivel, busca reducir el sobrecosto de espacio de las estructuras de datos existentes para *select*, utilizando sólo un nivel de directorio para guardar respuestas de bloques dispersos, y únicamente un muestreo regular de 1s. En cambio, estructuras de datos similares como la de Clark [8] y darray de Okanohara y Sadakane [25], utilizan tres y dos niveles de directorio para almacenar respuestas a bloques dispersos, ocupando mucho más espacio que en nuestra solución. A pesar de que darray responde consultas más rápidamente, nuestra estructura de datos provee un continuo de tradeoffs de espacio y tiempo que utilizan mucho menos espacio que darray.

La tercera es la Estructura para Bitmaps Dispersos, que mejora la solución de Raman *et. al* [29] reemplazando una tabla universal que está limitada por su tamaño exponencial respecto al tamaño de los bloques, con algoritmos que la simulan en tiempo de consulta. Esta estructura de datos retiene el desempeño de la solución de Raman *et. al* pero reduce su sobrecosto en espacio en 50 %-60 %, alcanzando sólo 0.1 bit de sobrecosto de espacio sobre la entropía y resolviendo *rank* en alrededor de 0.4 microsegundos y *select* en alrededor de 1 microsegundo.

Dado lo fundamental de las operaciones *rank* y *select*, nuestras implementaciones tienen aplicación inmediata en estructuras de datos para representar árboles, grafos, permutaciones, strings, relaciones binarias, y colecciones de texto, entre muchas otras aplicaciones. Aún más, nuestros resultados para *select* permiten el desarrollo de nuevas estructuras de datos basadas en esta operación, y que pueden ser eficientes en la práctica.

Lo anterior sugiere directamente una línea de trabajo futuro, que consiste en medir experimentalmente el desempeño de otras estructuras de datos compactas utilizando nuestras estructuras de datos para *rank* y *select*, tal como se realizó en el caso del FM-index.

Por otro lado, una nueva idea para mejorar las consultas *select* de la implementación de González *et. al* [16] consiste en acotar el intervalo para búsqueda binaria, de manera similar a hinted bsearch de Vigna, mediante el uso de un muestreo regular en 1s en la implementación basada en esta técnica. Se debe investigar si se justifica el costo del muestreo de 1s en relación a los tiempos de consulta.

Otra idea para mejorar el espacio de las estructura de Select 1 Nivel con codificación consiste en verificar si las respuestas codificadas que se almacenan en el bitmap de entrada utilizan menos espacio que el bitmap plano donde se almacenan. En estos casos el bitmap original puede acortarse, reduciendo así el espacio utilizado, pero con un nuevo costo para los punteros necesarios para mantener la correspondencia de las posiciones del bitmap modificado con el bitmap original. Se debe determinar experimentalmente en qué casos, según la densidad del bitmap y el muestreo de 1s, esta variante obtiene mejores resultados que la estructura de datos original.

Bibliografía

[1] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, January 2005.

[2] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 84–97. SIAM Press, 2010.

[3] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. 9th Latin American Symposium on Theoretical Informatics (LATIN 2010)*, LNCS 6034, 2010.

[4] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. 26th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 111–122, 2009.

[5] Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms (TALG)*, 7(4), 2011.

[6] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. In *Technical Report 124, Digital Equipment Corporation*, 1994.

[7] S. Büttcher, C.L.A. Clarke, and G.V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. Mit Press, 2010.

[8] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

[9] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[10] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194 – 203, 1975.

[11] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[12] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of

sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[13] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proc. 15th ESA*, LNCS 4698, pages 371–382, 2007.

[14] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[15] Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007.

[16] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. Posters of 4th Workshop on Efficient and Experimental Algorithms*, pages 27–38, 2005.

[17] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[18] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[19] Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. In *Proc. of the 4th International Conference on Experimental and Efficient Algorithms (WEA'05)*, 2005.

[20] I. Munro, R. Raman, V. Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 345–356, 2003.

[21] J. Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[22] J. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[23] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[24] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, LNCS 7276, pages 295–306, 2012.

[25] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[26] R. Pagh. Low redundancy in dictionaries with o(1) worst case lookup time. In *Proc. 26th*

International Colloquium on Automata, Languages and Programming (ICALP), pages 595–604, 1999.

[27] M. Pătraşcu. Succincter. In *Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

[28] Mihai Pătraşcu and Emanuele Viola. Cell-probe lower bounds for succinct partial sums. In *Proc. 21st ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 117–122, 2010.

[29] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4), November 2007.

[30] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.

[31] Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proc. 7th International Conference on Experimental Algorithms (ALENEX)*, pages 154–168, 2008.